Оглавление

Содержание

Об авторах

Основные авторы

Джарод Холингвэрт (Jarrod Hollingworth)

Джарод профессионально занимается программированием с 1993 года. В настоящее время он владеет фирмой Backslash (http://www.backslash.com.au), которая создает программное обеспечение для работы в сети Internet и различных секторах экономики и оказывает консалтинговые услуги по созданию программного обеспечения. Он обладает солидными навыками программирования на языке C/C++ в области телекоммуникаций и способствовал созданию первой в мире системы коротких сообщений (short-messaging system) для цифровой мобильной связи стандарта GSM.

Джарод начал свою карьеру программиста в 1985 году как любитель-самоучка, программируя на языках BASIC и Assembler, а затем, получив в университете Декина, Австралия (Deakin University, Australia), степень бакалавра информатики, перешел к программированию на языках Pascal и C/C++. Его профессиональные обязанности в процессе создания программного обеспечения варьировались от программиста до менеджера отдела программного обеспечения.

Обладая многолетним опытом работы с C++Builder и Delphi и опытом коллективной работы с Microsoft Visual C++, он считает C++Builder наилучшим инструментом для создания приложений для Windows.

Джарод живет в Мельбурне, Австралия (Melbourne, Australia), со своей женой Линдой. Его любимые занятия на досуге — путешествия и езда на велосипеде. Ему можно написать письмо по адресу jarrod@backslash.com.au.

Дэн Баттерфилд (Dan Butterfield)

В настоящее время Дэн занимается созданием программного обеспечения для математического моделирования и геоинформационных систем (GIS), используемых для контроля за окружающей средой в Центре исследований водной окружающей среды (Aquatic Environments Research Centre — AERC) при Редингском университете, Великобритания (University of Reading, UK). Как единственный программист в этом исследовательском центре Дэн тесно сотрудничает со своими коллегами над созданием программного обеспечения для крупномасштабного моделирования качества воды и визуализации данных. Программа INCA (Integrated Nitrogen in Catchments) является новейшей реализацией модели циркуляции азота. Она широко используется во всем мире и постоянно совершенствуется. Дэн начал свою профессиональную карьеру с программирования на языке BASIC в возрасте 12 лет, а теперь обладает навыками профессионального программирования на языках REXX, Fortran, Pascal, C и C++. В течение двух последних лет Дэн работал исключительно с Borland C++Builder.

Боб Сворт (Bob Swart)

Боб Сворт (или Dr. Bob — http://www.drbob42.com) является старшим научным консультантом по использованию Delphi, C++Builder и JBuilder (а также Kylix) в компании TAS Advanced Technologies (http://www.tas-at.com), Бест, Нидерланды (Best, The Netherlands). Боб

также публикует свои статьи в журналах *The Delphi Magazine*, *Delphi Developer*, *UK-BUG NewsLetter* и *SDGN Magazine*. Он соавтор книг *The Revolutionary Guide to Delphi 2*, *Delphi 4 Unleashed* и *C++Builder 4 Unleashed*. Он часто выступает на конференциях Borland, которые проводятся по всему миру.

Джэйми Оллсоп (Jamie Allsop)

Джэйми живет в Северной Ирландии. Он изучил множество языков, но отдает предпочтение C++. Он использовал C++Builder с момента выпуска, и по сей день этот пакет остается для него любимым инструментом создания программного обеспечения. Джейни имеет ученую степень в области электроники и в настоящее время занят научной работой. Кроме того, он вместе со своей женой владеет фирмой по созданию программного обеспечения, Shiying, и создает компоненты для коммуникационных и DSP-приложений реального времени. Среди его увлечений стоит упомянуть европейский футбол и бадминтон.

Другие авторы

Роб Аллен (Rob Allen)

Роб живет в Лондоне, Великобритания (London, UK). Имеет ученую степень в области электроники, но последние пять лет посвятил программированию с помощью Borland C++ 4.5, 5 и C++Builder 3. У него большой опыт создания тестовых и измерительных инструментов для мобильной связи.

Халид Алманай (Khalid Almannai)

Халид живет в Бахрейне. Начинал программировать с помощью Borland C++ 4.5, а теперь имеет опыт работы на языках C, C++, BASIC и Delphi. Два года занимается программированием с использованием C++Builder.

Дру Эйвис (Drew Avis)

Дру работает технически писателем, специализируясь на создании интерактивной документации. Живет в Мериквилле, Онтарио (Merrickville, Ontario) и в настоящее время работает в компании Nortel Networks. Дру имеет опыт работы в больших и малых фирмах, преподавал подготовленный им курс лекций о создании технической документации для инженеров в университете Калгари (University of Calgary). На досуге Дру любит варить пиво, играть в хоккей и работать над созданием программы для пивоваров StrangeBrew.

Джей Банкс (Jay Banks)

Джей живет в Великобритании. Программирует с 11 лет и теперь обладает опытом программирования во многих областях, включая бухгалтерский учет, автоматизированное управление, безопасность, игры, образовательные программы и работа с базами данных. Он имеет разнообразный опыт работы с C++Builder.

Эдуардо Безерра (Eduardo Bezerra)

Эдуардо живет в Рио-де-Жанейро, Бразилия (Rio de Janeiro, Brazil). Он имеет 19-летний опыт программирования на языке C++. Эдуардо начинал программировать с помощью Borland Turbo C и работал со всеми компиляторами Borland C/C++ вплоть до C++Builder. Владеет небольшой компанией, оказывающей консультационные услуги и создающей программное обеспечение для работы в сети Internet и с телекоммуникациями. У него четырехлетний опыт работы с технологией COM, в области системной интеграции и создания распределенных сред.

Филипп Х. Блантон II (Phillip H. Blanton II)

Филипп изучал физику и информатику в университете Северного Колорадо (University of Northern Colorado). Занимается программированием для персональных компьютеров с помощью инструментов Borland начиная с 1987 года. Последние 13 лет он выполнял обязанности системного администратора, директора информационного отдела, разработчика баз данных и специалиста по системной безопасности. В настоящее время занимает должность старшего инженера-программиста фирмы TurboPower Software в Колорадо-Спрингс (Colorado Springs). Он вместе со своей женой Мэри Эн, двумя дочерьми Дэли и Сидни и замечательным псом Харлеем живет в горах, окружающих Вудлэнд Парк, Колорадо (Woodland Park, Colorado). Ему можно написать письмо по адресу `phillipb@turbopower.com`.

Джо Бонавита (Joe Bonavita)

Джо живет в Коннектикуте (Connecticut). Начал программировать в возрасте 14 лет на компьютере Commodore 64 как программист-самоучка. Затем перешел к работе с персональными компьютерами и языками BASIC, Assembler и C++. Свою работу с C++Builder Джо начал с версии 1. Он изучил много других языков и внес весомый вклад в эту книгу. Джо считает, что такие книги особенно полезны тем, кто самостоятельно постигает азы программирования.

Дж. Алан Броуган (J. Alan Brogan)

Алан живет в Кэрри, Ирландия (Kerry, Irland). Он имеет более чем 10-летний опыт программирования и преподавания и в настоящее время возглавляет команду разработчиков программного обеспечения в области телекоммуникаций. Алан владеет многими языками программирования, но предпочитает C и C++ за их простоту и элегантность. Алан всегда предпочитал IDE-среды фирмы Borland и продолжает использовать IDE-среду Turbo C, но все же большую часть времени посвятил программированию с помощью C++Builder.

Марк Кашман (Mark Cashman)

Марк — профессиональный разработчик программного обеспечения с опытом работы в разных областях: от страхового дела до геоинформационных систем для оптовой поставки продукции. Он специализируется на вопросах взаимодействия компьютера и пользователя, реляционных базах данных, компонентном программном обеспечении, объектно-ориентированном подходе, создании программного обеспечения для Web-сайтов, включая его собственный сайт, The Temporal Doorway (`http://www.temporaldoorway.com/programming/index.htm`). На нем можно найти раздел, посвященный C++Builder. Марк работал консультантом, разработчиком, менеджером а теперь является старшим разработчиком компании V-Technologies, LLC, которая поставляет программное обеспечение промежуточного уровня. Является членом Borland TeamB — официальной, но некоммерческой группы рассылки новостей специалистам, имею-

щим соответствующий уровень квалификации и желание помогать другим разработчикам в работе с продуктами фирмы Borland. Марк также известен как удостоенный многих наград художник, автор опубликованных научно-фантастических книг (в настоящее время он работает над своей третьей книгой), а также технических книг (включая несколько статей в журнале *C++Builder Developer's Journal*), композитор, который создает с помощью компьютера музыкальные произведения в стиле фьюжн, а также скалолаз с опытом сложных восхождений.

Дэймон Чандлер (Damon Chandler)

Дэймон занимается программированием более 15 лет. Программист-самоучка, он проделал типичный путь: от программирования на языках GW-BASIC, Pascal перешел к языку C++. Дэймон формально работает инженером, но в душе — программист. В настоящее время он занимается научной деятельностью в лаборатории визуальных коммуникаций (Visual Communications Lab) в Корнельском университете (Cornell University) и в основном работает над проблемой сжатия изображений с помощью волнового преобразования. C++Builder является его любимым инструментом. Недавно Дэймон стал членом группы TeamB.

Джеп Крамон (Jeppe Cramon)

Джеп живет в Копенгагене, Дания (Copenhagen, Denmark). Он имеет 7-летний опыт программирования: начинал с языка Pascal, а затем продолжил работу с языками C++, Visual Basic, Java и C#. Джеп использовал C++Builder в течение последних четырех лет и недавно начал работать с технологиями XML, ASP, DHTML, COM и .NET. Джеп имеет ученую степень в области электроники и информатики. Помимо работы в собственной консультационной фирме, Джеп публикует свои материалы на Web-сайте Bytamin-C (`http://www.bytamin-c.com`). Он также создал программу Jzip, бесплатную версию программы WinZip и аркадную игру B.U.G.S. для операционных систем OS/2 и Windows. В краткие часы досуга Джеп с энтузиазмом занимается горнолыжным спортом и играет на ударных инструментах.

Марк Дэйви (Mark Davey)

Марк живет в Гемпшире, Англия (Hampshire, England). С подросткового возраста занимается профессиональным программированием, а C++Builder использует в своей работе в течение последних двух лет. Его деятельность в основном была связана с созданием систем управления реального времени для радиовещательных приложений в ведущих радиовещательных компаниях Великобритании, а в настоящее время он занимается созданием радиокоммуникационных систем. Марк владеет фирмой, которая занимается созданием систем регистрации данных для мотоциклетного спорта, причем исключительно на основе C++Builder. Интересуется новинками музыки, искусства и техники. Совсем недавно он женился на Саре.

Пол Густавсон (Paul Gustavson)

Пол живет в штате Вирджиния и работает старшим системным инженером Synetics, Inc. (`http://www.synetics.com`), американской компании, которая предоставляет свои услуги в области управления знаниями, системного проектирования и корпоративного управления. Пол прилагает усилия для решения широкого круга задач в области распределенного моделирования среди членов сообщества DoD. Он считается опытным программистом и специализируется в C++Builder. Пол является партнером по бизнесу в новой и быстро развивающейся компании SimVentions (`http://www.simventions.com`), которая занимается созданием и внедрением суще-

ствующих методов и технологий для создания инновационных приложений и решений с интенсивным использованием интеллекта и дополнительных знаний. Пол занимается созданием приложений для Windows начиная с оболочки Windows 3.1 (1993) и опубликовал на эту тему больше десятка технических статей на Web-сайте организации Simulation Interoperability Standards Organization (http://www.sisostds.org), которая занимается стандартизацией моделирования обмена данными между компьютерными системами.

Джон Мак-Суин (John MacSween)

Джон с отличием закончил университет в Глазго (Glasgow University) по специальности корабельная архитектура и кораблестроение. В настоящее время он работает в компании Henry Abram & Sons Ltd, которая занимается проектированием грузовых и пассажирских судов в Глазго, Шотландия (Glasgow, Scotland). Джон начал программировать с 14-летнего возраста на языках BASIC, Fortran и C++. Последние полтора года работает с C++Builder, используя этот инструмент для создания приложений. Ему можно написать письмо по адресу j.a.macsween@henryabram.co.uk или познакомиться поближе, посетив его Web-сайт http://www.redrival.com/mandtsoftware/.

Стефан Махо (Stephane Mahaux)

Стефан руководит собственной консультационной компанией, Hyperian Development Solutions (http://www.hyperian.ab.ca), которая находится в Эдмонтоне, Канада (Edmonton, Canada). Начал программировать еще до появления оболочки Windows 3 и профессионально занимается программированием более 10 лет. Он работает с C++Builder начиная с версии 1. Стефан также жил во Франции, где создавал инструменты для работы с базами данных. Редактировал ежемесячную колонку в журнале *Point DBF* о программировании баз данных с помощью различных инструментов разработки программного обеспечения.

Вильям Моррисон (William Morrison)

Еще в средней школе, познакомившись с Turbo Pascal 7, Вильям полюбил программные продукты фирмы Borland. Он профессионально занимается программированием с помощью C++Builder и Delphi более трех лет. С помощью C++Builder он создал упреждающий номеронабиратель и другие стандартные бизнес-приложения, использовав свой опыт работы с интерфейсами ISAPI, API и технологией проектирования база данных C/S. Он помогает новичкам в освоении C++Builder и Delphi, используя консультационные каналы Undernet IRC, а иногда публикует свои материалы в пользовательских группах Borland Usenet. В настоящее время он является вице-президентом отдела создания программного обеспечения консультационной компании CXC Consulting (http://www.cxcca.com) в Калифорнии. Он выражает благодарность Ясону Вортону (Jason Wharton) из компании IBObjects и Джефу Оверкэшу (Jeff Overcash) из компании IBX за помощь при написании разделов об InterBase, что лишь демонстрирует профессионализм программистов InterBase из его группы поддержки.

Айонел Муньоз (Ionel Munoz)

Айонел живет в Квебеке (Quebec). Он работает старшим системным программистом в компании EXFO, производящей системы для тестирования оптоволоконного оборудования. В его обязанности входит создание архитектуры и каркаса таких систем на основе техноло-

гии COM/DCOM. Айонел начинал программировать с помощью Turbo Pascal в 1990 году, а на языке C++ — в 1992 году, причем C++Builder начал использовать с самой первой версии. Имеет опыт создания приложений во многих областях, включая игры, компоненты, Java-аплеты, базы данных с архитектурой клиент/сервер, системы сбора данных и мультимедиа-приложения. Начиная с 1995 года создавал приложения на основе COM-технологии и библиотеки ATL в пакетах VC++ и C++Builder. Им опубликованы примеры использования COM-технологии в базе данных исходных кодов CodeCentral фирмы Borland. Помимо компьютеров, в сферу его интересов входят традиционное каратэ, книги и музыка. Айонел имеет жену Сильвию и дочь Мариэтту.

Джин Паризье (Jean Pariseau)

Джин работает шеф-поваром в престижном клубе в Новой Англии и учится в Комьюнити-колледже на Род Айленде (Community College, Rhode Island), где скоро получит ученую степень. Джин программирует более 14 лет, начиная с компьютеров Commodore 64 и Amiga и заканчивая компьютерами на основе операционной системы Windows. В основном с помощью самообразования Джин освоил языки C++, Pascal/Delphi, Assembler, Lisp и Java. Среди его профессиональных интересов следует отметить компьютерную алгебру, численные методы, вычислительную гидродинамику и проектирование компиляторов.

Пит Педерсен (Pete Pedersen)

Пит живет в Логане, Юта (Logan, Utah), и профессионально занимается программированием более 30 лет. Он использует продукты фирмы Borland, начиная с C++Builder версии 1. Занимал должность международного консультанта, основал компанию BlueLine Software, публиковал статьи в отраслевых изданиях. В настоящее время работает в фирме Spiricon, которая является лидером в области лазерной диагностики. Свободное время Пит любит проводить со своей семьей. Вместе с девятилетним сыном Камероном они работают над книгой для детей по программированию с помощью пакета C++Builder.

Рууд Ф. Пелс (Ruurd F. Pels)

Рууд работает консультантом в компании TAS Advanced Technologies B.V. (Нидерланды). Он обладает большим опытом работы в создании программного обеспечения, особенно на языке C++. Регулярно публикует заметки в разделе C++Builder Gate на Web-сайте журнала *Dr. Bob* (`http://www.drbob42.com/Cbuilder/`), а также обзоры и статьи в других журналах. Ему можно написать письмо по адресу `ruurd@worldonline.nl`. Имеет жену Ангелину и двух сыновей Рубена и Дэниела.

Шон Рок (Sean Rock)

Шон родился и жил в Сан-Диего, Калифорния (San-Diego, California), но последние пять лет живет в Болтоне, Великобритания (Bolton, UK). Он начал программировать в колледже на языке Pascal и самостоятельно изучил Delphi и C/C++. Занимается программированием в основном как любитель, но иногда выполняет заказы местных компаний. Шон использует C++Builder в течение двух лет и недавно начал работу над созданием Web-сайта Component Writer's Guide, посвященного вопросам создания компонентов с использованием библиотеки VCL, который можно найти по адресу `http://www.componentwriter.co.uk`.

Саймон Ратли-Фрэйн (Simon Rutley-Frayne)

Саймон живет в Девоне, Великобритания (Devon, UK). Он начал программировать с помощью C++Builder только два года назад и вскоре основал фирму Casimo Software (http://www.casimosoftware.co.uk), которая занимается созданием условно-бесплатного программного обеспечения и их компонентов. Кроме того, Саймон руководит собственной консультационной фирмой и является редактором журнала *TheBits* (http://www.thebits.org), где отвечает за публикацию обзоров о компонентах, приложениях и книгах.

Викаш Шах (Vikash Shah)

Викаш живет в Мидлсексе, Великобритания (Middlesex, UK). После получения ученой степени в области математики и информатики он три года занимался профессиональным программированием на языке C++, два из них — на C++Builder.

Малькольм Смит (Malcolm Smith)

Малькольм живет в Австралии. Он самостоятельно овладел методами программирования и работал системным программистом, создавая MIS-решения для полиграфической промышленности, а также драйверы устройств для мониторинговых и следящих систем, предназначенных для обеспечения безопасности. Кроме того, он руководит собственной фирмой, MJ Freelancing, в которой создаются инструменты шифрования, защиты от пиратской деятельности и другие инструменты.

Кит Тернбулл II (Keith Turnbull II)

После окончания Мичиганского государственного университета (Michigan State University) Кит с 1991 года занимался сетевым программированием. Он участвовал в создании разнообразных систем: от интерактивных игр до программ для оформления страховых полисов. В настоящее время он вице-президент отдела по созданию приложений с архитектурой клиент/сервер в компании Intercanvas Design, Inc. Кит живет в Дес-Мойнес, Айова (Des Moines, Iowa), со своей женой, двумя собаками, тремя котами и четырьмя компьютерами.

Крис Винтерс (Chris Winters)

Крис живет в Ричмонде, Вирджиния (Richmond, Virginia). Он занимается программированием в течение 6 лет. Свободно владея языками C/C++, Крис использовал в своей деятельности BASIC, HTML, Java, Assembler, MS Access и Delphi. Он начинал свою карьеру с Turbo C++ (для DOS) и C++Builder версии 1. В настоящее время Крис работает на PSINet — одного из самых крупных и опытных в своем деле провайдеров коммуникационных услуг на основе IP-протокола. Крис также создает системные утилиты и условно-бесплатное программное обеспечение, а также экспериментирует с новыми Win32 API-функциями. У него двое детей: Лорен Эшли и Колин Мэйсон. Ближе с ним можно познакомиться, посетив его Web-сайт по адресу http://www.encomsysware.com.

Вильям Вудбури (William Woodbury)

Вильям начал программировать в возрасте 12 лет. Он владеет многими языками программирования: ADA, BASIC, Pascal, Java, C/C++ и Assembler, но в настоящее время предпочитает использовать C++Builder. У него большой опыт создания графических и сетевых приложений, а также игр.

Сью-Фэн By (Siu-Fan Wu)

Сью-Фэн получил степень бакалавра по физике и доктора наук по электротехнике и электронике в университете графства Суррей, Великобритания (Univerity of Surrey, UK). Занимался научной деятельностью в научно-исследовательском центре фирмы Canon, работая над проблемой визуализации акустических полей, а затем преподавал в политехническом университете в Сингапуре (Singapore Polytechnic). В настоящее время читает лекции о микропроцессорах и инструментах в институте профессионального обучения в Гонконге (Hong Kong Institute of Vocational Education). В круг его научных интересов входит обработка изображений, сжатие и извлечение данных. У него девятилетний опыт программирования с помощью Borland C и C++Builder.

Йото Йотов (Yoto Yotov)

Йото Йотов в настоящее время учится в Монреале (Montreal). Постоянно работая с пакетом C++Builder, он изучает его возможности с момента появления первой версии. Опубликовал множество статей в разных технических журналах и на Web-сайтах.

Зе Ксианг By (Ze Xiang Wu)

Зе Ксианг By имеет более чем 10-летний опыт программирования с помощью Borland C++. Ее последняя работа — программирование интерфейса. Помимо работы, она увлекается туризмом, классическими и народными танцами, а также приготовлением национальных блюд разных стран, например своих любимых блюд Южного Китая, где она родилась.

Посвящения

Джарод Холингвэрт

Я посвящаю эту книгу моей жене Линде. Книга появилась только благодаря ее любви, поддержке, пониманию и снисходительному отношению к моей постоянной просьбе “еще немного, еще 15 минут” (что на самом деле означало еще полтора часа) в течение многих и долгих дней и ночей.

Дэн Баттерфилд

Моей семье.

Боб Сворт

Айвон, Эрику и Наташе.

Джэйми Оллсоп

Венмэй (Зе Ксианг Ву), которая всегда верила в меня.

Введение

Этот чрезвычайно интересный проект имел, казалось бы, очень простую цель — написать новую книгу о C++Builder 5, где были бы представлены не только сведения о новой версии 5, но и другие темы, которые никогда и нигде не поднимались.

Идея создания этой книги появилась в ноябре 1999 года сразу после получения письма от Джарода Холингвэрта в списке рассылки технических материалов по C++Builder — "The Bits" (`http://www.thebits.org`).

> "Я только что вернулся с замечательной конференции разработчиков Australia & New Zealand BorCon99. На ней было представлено большое количество модернизированных версий всех программных продуктов Borland; учебные пособия и семинары были достаточно информативны. В общем все было великолепно, но мне показалось, что книги *C++Builder 5 Unleashed* там не было…"

Это открытие изумило всех подписчиков The Bits, потому что книги серии *Unleashed* имеют репутацию бесценных источников информации, причем довольно часто они являются единственным доступным справочным пособием по C++Builder (помимо руководств для пользователей *Teach Yourself...* и *Developer's Guide* фирмы Borland).

Некоторые подписчики выразили готовность написать статьи о своей области деятельности, и таким образом было положено начало этому проекту. Прежде всего следовало решить, какие темы наиболее интересны сообществу разработчиков C++Builder. Для этого был создан Web-сайт под названием "The C++Builder Book Writers' Guild" ("Гильдия писателей книги по C++Builder") с интерактивным отчетом об исследовании этой проблемы, который был разрекламирован среди большого количества форумов и списков рассылки разработчиков C++Builder. Результаты этого исследования и интерактивных обсуждений (которые можно найти на Web-сайте книги по адресу `http://www.bcb5book.force9.co.uk`) помогли окончательно сформировать основу книги.

Итак, вот что получилось в итоге. В написании книги приняли участие 34 автора из многих стран мира, включая Австралию, Бахрейн, Бразилию, Канаду, Данию, Гонконг, Ирландию, Нидерланды, Великобританию и США. Каждый автор представил статью в своей области, которая приняла окончательный вид после обмена мнениями по электронной почте, обсуждения в чате и на Web-сайте (а также по телефону). Несмотря на активную профессиональную и личную жизнь, которую ведут все авторы, им удалось найти время для работы над книгой. У большинства авторов небольшой опыт написания книг или нет вовсе; одновременно читатель найдет среди них известных авторов книг по C++Builder и Delphi, участников списка рассылки TeamB и создателей Web-сайтов, посвященных C++Builder.

На эту очень сложную работу организаторам проекта пришлось потратить более 600 часов, отправить 2000 и получить более 3500 электронных сообщений, касающихся только организационных вопросов. Мы получили невероятно полезный опыт и надеемся, что история проекта убедит читателя в отсутствии чего-либо загадочного в написании технических книг. Читатель может попробовать и убедиться в этом сам. Авторы надеются, что они создали книгу, которая окажется полезной для пользователей всех версий C++Builder и при разработке всех типов приложений. Мы постарались включить в нее темы и методы, которые не описывались в других книгах, а также представить новые компоненты C++Builder 5. Кроме того, авторы надеются, что, благодаря уникальному способу написания этой книги, сохранен дух C++Builder/Delphi — делиться знаниями с другими.

Важные замечания

Поскольку над книгой работало очень много авторов, различия в их литературных стилях могут быть заметны даже в пределах одной главы. Мы (авторы и группа редакторов издательства Sams) постарались устранить этот недостаток, но в некоторых местах этого сделать не удалось.

Как это всегда бывает с техническими книгами, несмотря на все наши усилия, в тексте и прилагаемом коде неизбежно присутствуют случайные ошибки. Для их устранения мы поддерживаем список найденных ошибок на Web-сайте книги по адресу `http://www.bcb5book.force9.co.uk` и на Web-сайте издательства Sams по адресу `http://www.samspublishing.com`. Сообщения о найденных ошибках, а также вопросы и замечания вы можете послать по адресу `feedback@bcb5book.f9.co.uk`.

Что находится на прилагаемом компакт-диске

Компакт-диск содержит коды всех примеров и проектов C++Builder из этой книги. Они собраны в отдельных папках для каждой главы, а доступ к ним может быть организован с помощью встроенного интерфейса. Некоторые папки (для глав 13, 15, 16, 18 и 19) также содержат файлы `README.TXT` с важной информацией.

Кроме того, не нем собрано большое количество бесплатных, условно бесплатных, демонстрационных и пробных версий компонентов и приложений, которые могут представлять интерес для пользователей C++Builder. Их список можно просматривать также с помощью специальной программы-интерфейса для этого компакт-диска. Убедиться в том, что этот список содержит самую свежую информацию о содержимом компакт-диска, можно, открыв в корневом каталоге компакт-диска файл `README.TXT`.

Благодарности

В ходе работы над этим проектом список авторов постоянно изменялся, что было связано с переключением их деятельности на более приоритетные задачи. Пользуясь возможностью, мы хотим поблагодарить всех, кто выразил желание помочь в создании книги, но не смог продолжить эту работу. Выражаем особую благодарность Рику Малику (Rick Malik), создателю (и владельцу) Web-сайта книги, который затратил огромное количество времени и усилий на начальных стадиях этого проекта.

На кого рассчитана эта книга

Книга не является ни учебным пособием по C++, ни учебником по использованию пакета C++Builder. Скорее всего, она представляет собой руководство по использованию C++Builder для создания более качественных, крупных и сложных приложений, для расширения навыков работы с C++Builder и ознакомления с новыми компонентами C++Builder 5.

Если вы уже имеете опыт разработки приложений с помощью C++Builder, желаете совершить переход от версии 4 к версии 5 или повысить уровень своих знаний, то эта книга предназначена именно для вас. Изложение материала постепенно усложняется в большинстве глав и книге в целом. Поэтому она будет полезна также для начинающих пользователей C++Builder, хотя первоначально предназначалась только для читателей со средним и даже высоким уровнем подготовки. В результате оказалось, что она доступна читателям всех уровней, несмотря на повышенную сложность некоторых обсуждаемых тем.

Системные требования для работы с C++Builder

Книга *C++Builder 5 Developer's Guide* в основном предназначена для пользователей C++Builder версии 5, но большая часть текста и кода примеров в равной степени применима и для версии 4. Применимость излагаемого материала для каждой из версий C++Builder указана в табл. В.1.

Таблица В.1. Применимость книги (в процентах) для разных версий C++Builder

Версия C++Builder	Применимость
C++Builder 5 Enterprise	100%
C++Builder 5 Professional	94%
C++Builder 5 Standard	77%
C++Builder 4 Enterprise	84%
C++Builder 4 Professional	79%

Хотя большая часть кода, содержащегося в книге, должна работать и с C++Builder версии 4 (конечно, за исключением особых компонентов версии 5), многие проекты C++Builder на прилагаемом компакт-диске представлены в формате, пригодном для версии 5. Так как этот формат несовместим с C++Builder версии 4, то для использования их в формате версии 4 пользователям необходимо создать новые проекты, вставить в них код с прилагаемого компакт-диска, а затем добавить формы и соответствующие свойства.

Ниже перечислены минимальные системные требования для C++Builder 5 Enterprise:

- Intel Pentium 90 или выше (рекомендуется Pentium 166),
- Microsoft Windows 2000, Windows 95, 98 или NT4.0 с Service Pack 3 или выше,
- оперативная память 32 Мбайт (рекомендуется 64 Мбайт)
- жесткий диск: 253 Мбайт для минимальной инсталляции, 388 Мбайт для полной инсталляции,
- дисковод компакт-дисков,
- монитор VGA или с более высокой разрешающей способностью,
- мышь или другой манипулятор.

Структура книги

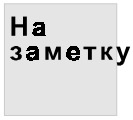
Книга состоит из семи частей. Первые пять частей организованы в виде естественной последовательности тем, начиная с описания основных методов работы с C++Builder и C++, методов обмена данными, приемов работы с базой данных, в Web-среде и распределенного программирования и до более сложных вопросов программирования, включая OpenGL, а также инсталляцию и распространение программного обеспечения. Последние две части содержат подсказки, советы и рекомендации по использованию C++Builder, пример рабочего приложения, а также другие рекомендованные источники информации о C++Builder.

Ниже приводится краткое описание частей книги.

- Часть I. “Основы C++Builder 5”. Эта часть, состоящая из глав с 1-ой по 11-ю, содержит все, что необходимо знать для оптимального использования C++Builder 5 при разработке приложений. Она включает введение в C++Builder и интегрированную среду разработки (Integrated Development Environment) (главы 1 и 2); советы по программированию на C++ и разработке программного обеспечения с C++Builder (главы 3–5); обсуждение вопросов компиляции, оптимизации и отладки (главы 6 и 7); а также исчерпывающую информацию по использованию и созданию VCL-компонентов (главы 8–11).
- Часть II. “Обмен информацией, базы данных и программирование в Web”. Эта часть, включающая глава с 12-ой по 14-ю, охватывает многие аспекты программирования обмена данными, работы с базой данных и в Web-среде. Она включает описание протоколов последовательного обмена данными и работы в Internet (глава 12); программирование на основе WebBroker, InternetExpress и XML (глава 13); программирование базы данных, в частности ADO Express, InterBase Express, с помощью нового компонента Data Module Designer, а также обсуждение параметров архитектуры баз данных (глава 14).
- Часть III. “Интерфейсы и распределенные вычисления”. Эта часть (главы 15–22) содержит подробную информацию по всем аспектам программирования интерфейсов и распределенных вычислений. Она включает создание и использование DLL-модулей, пакетов C++Builder и подключаемых модулей (глава 15); программирование COM, DCOM и COM+ (главы 16–18); MIDAS 3 (глава 19); CORBA (глава 20); интеграция с Microsoft Office, в частности с Word и Excel (глава 21); программирование ActiveX (глава 22).
- Часть IV. “Более сложные методы работы в C++Builder”. Эта часть (главы 23–26) охватывает дополнительные вопросы, которые обычно отсутствуют в книгах по C++Builder: усовершенствованные методы печати и представления данных (глава 23); исчерпывающая информация об использовании Win32 API (глава 24); обработка изображений (GDI, форматы GIF, JPEG и т.д.) и звука (WAV, MP3 и т.д.) с помощью C++Builder (глава 25); а также обсуждение более сложных вопросов программирования на основе DirectX и OpenGL (глава 26).
- Часть V. “Инсталляция и распространение приложения C++Builder”. Эта часть (главы 27–29) содержит дополнительную информацию, которая обычно не входит в книги по C++Builder. Она включает методы и рекомендации по созданию стандартных файлов справки Windows и файлов справки в формате HTML (глава 27); вопросы распространения программного обеспечения с особым вниманием к аспектам, связанным с условно-бесплатным программным обеспечением (глава 28); методы инсталляции и обновления программного обеспечения, включая контроль версий с помощью TeamSource (глава 29).
- Часть VI. “База знаний”. Эта часть включает главы 30 и 31, которые содержат набор подсказок, советов и рекомендаций по работе с C++Builder (глава 30); а также пример полноценного рабочего приложения (глава 31).
- Часть VII. “Приложение”. Приложение содержит исчерпывающий список ресурсов с описанием C++Builder, включая Web-сайты (особенно Web-сайты сообщества разработчиков, использующих программные продукты фирмы Borland, а также CodeCentral), телеконференции, списки рассылки, форумы, книги, журналы и пользовательские группы.

Используемые обозначения

В этом разделе описываются обозначения, используемые в книге для выделения особых сведений: примечаний, советов и предостережений.



Примечания содержат комментарии и отступления от основной темы, а также более полное описание отдельных тем.

Советы содержат ссылки и рекомендации по более эффективному использованию C++Builder 5.

Предостережения указывают читателю на возможные неприятности при программировании и сообщают, как избежать их возникновения.

Кроме того, в книге можно найти разные типографские соглашения, используемые для выделения следующих фрагментов текста.

- Команды, имена переменных, каталогов и файлов представлены с помощью особого `моноширинного шрифта`.
- Текст-заменитель в описаниях синтаксиса представлен с помощью особого *`моноширинного шрифта с наклонным начертанием`*. Такое обозначение указывает на то, что текст-заменитель должен быть заменен фактическим именем файла, параметром или другим элементом, который он представляет.

Основы C++ Builder 5

ЧАСТЬ

I

Крис Винтерс (Chris Winters)
Халид Алманай (Khalid Almannai)
Джарод Холингвэрт (Jarrod Hollingworth)
Викаш Шах (Vikash Shah)

Глава

1

Введение в C+ +Builder

В этой главе рассмотрены основы Borland C++Builder, одной из ведущих сред разработки для создания настольных, клиент/серверных, распределенных приложений и приложений для работы в Internet. Она сочетает простоту среды быстрой разработки приложений или RAD-среды (Rapid Application Development — RAD) с мощью и производительностью языка ANSI C++.

C++Builder является любимым инструментом программистов-любителей, команд разработчиков в небольших и средних фирмах, а также больших коллективов разработчиков в крупных корпорациях.

На заметку

Более подробную информацию о преимуществах использования C++Builder можно найти, обращаясь к ссылкам "Features & Benefits" и "New C++Builder Users" на Web-сайте C++Builder по адресу `http://www.borland.com/bcppbuilder/`.

Опытные пользователи C++Builder 5 могут пропустить эту главу, в которой рассматриваются основы C++Builder, вопросы миграции к C++Builder из других сред разработки, анализируются преимущества и недостатки C++Builder по сравнению с другими средами разработки, а также представляются новые компоненты C++Builder версии 5.

В несколько другой манере в ней представлена среда разработки, создаваемая фирмой Borland для операционной системы Linux, под условным названием Kylix, которая представляет собой один из наиболее ожидаемых программных продуктов для разработчиков за последние годы. Kylix по сути является аналогом C++Builder и Delphi для Linux и позволяет создавать приложения для Linux так же легко, как и приложения для Windows с помощью C++Builder и Delphi. Приемы и навыки работы C++Builder, а также большая часть созданного кода могут быть использованы в среде Kylix.

Основы C++Builder

В этом разделе описываются основы C++Builder и представлен пример простого приложения. Ознакомившись с этими основными сведениями, вы убедитесь в том, что пакет C++Builder достаточно прост в использовании.

Начало начал: Hello, World!

Итак, вы только что открыли коробку с C++Builder, отложили в сторону книги с руководствами пользователя, закончили инсталляцию и готовы приступить к работе. Не так ли? Ну что ж мы готовы помочь вам в изучении основ C++Builder, создании проектов, использовании VCL-библиотек (Visual Component Library), `Object Inspector` и многом другом. Если вы новичок и хотите побыстрее всему научиться, то эта глава предназначена именно для вас. А если вы опытный разработчик, то вам лучше пропустить ее и перейти к следующим главам.

Сначала разработчикам приходилось программировать в режиме командной строки с помощью таких замечательных программных продуктов, как Turbo C++. Однако с тех пор как Windows стала доминирующей операционной системой, компиляторы были оснащены средствами программирования для Windows. После представления компилятора Borland C++ Compiler 3.1 весь мир постепенно стал переходить к разработке приложений для Windows. Однако сначала эта работа была сложной, и для ее освоения требовалось приложить немало усилий. Порой требовалось написать от трех до пяти страниц кода только для того, чтобы отобразить окно на экране компьютера, поскольку в него нужно было включить код перехвата сообщений для манипуляций с окном, его перерисовки и выполнения других действий в среде Windows.

После введения библиотеки OWL (ObjectWindows Library) процесс программирования существенно упростился, так как OWL избавила программистов от многих рутинных опера-

ций. Компилятор Borland C++ Compiler 3.1 содержал библиотеку OWL, а потому разработка приложений для Windows немного упростилась. Спустя некоторое время фирма Borland выпустила компилятор Borland C++ Compiler 5.0 с усовершенствованной библиотекой OWL и интегрированной средой разработки (IDE).

Вскоре после успеха Borland C++ 5.0 был выпущен программный продукт Delphi, предназначенный для визуального программирования на языке Pascal. Затем фирма Borland выпустила C++Builder, который был создан на основе Delphi. Так как в C++Builder использована такая же технология, то разработка приложений для Windows существенно упростилась. Среда разработки Delphi основана на усовершенствованной RAD-технологии, унаследованной в C++Builder, что позволило ему стать одним из лучших современных компиляторов. По сравнению с Visual C++, разработка приложений с его помощью выполняется быстрее, причем приложения проще переносятся на другие платформы.

Для конечного пользователя, который собирается стать разработчиком, C++Builder стал одним из наиболее популярных компиляторов в мире. Им довольно легко пользоваться, так как можно быстро научиться основным приемам работы с ним. Разработчики Visual Basic смогут убедиться в том, что манипуляции с компонентами Visual Basic аналогичны манипуляциям с компонентами в C++Builder. Интегрированная среда разработки C++Builder до некоторой степени подобна интегрированной среде разработки Visual Basic, и можно довольно быстро научиться основным приемам работы с ней.

Для профессиональной работы с этим пакетом необходимо глубоко изучить язык C++, для чего может понадобиться несколько лет. Однако Borland C++Builder поможет вам в этом. Borland/Inprise существенно упростила библиотеку VCL-библиотек и ее функции классов, что избавило разработчика от выполнения многих рутинных и трудоемких операций. Если вы решили воспользоваться преимуществами быстрой разработки приложений, то использование C++Builder является наилучшим способом реализации этого намерения.

Итак, приступим. Прежде всего откройте руководство пользователя и убедитесь в том, что пакет C++Builder правильно инсталлирован и только после этого откройте интегрированную среду разработки.

На заметку

Авторы не будут подробно описывать здесь каждую часть C++Builder, поскольку этому посвящена вся книга в целом. Кроме того, подробную информацию по инсталляции и управлению C++Builder читатель может всегда найти в руководстве пользователя или интерактивной справке. Здесь лишь будет приведена основная информация, которая необходима для создания простейших приложений.

После запуска интегрированной среды разработки C++Builder откроются три основных окна среды программирования, которая называется *интегрированной средой разработки* или *IDE-средой* (*Integrated Development Environment*). Показанная на рис. 1.1 IDE-среда обладает всеми необходимыми инструментами программирования и является рабочей областью создаваемого разработчиком проекта. Здесь создаются элементы управления и компоненты, вводится код, конфигурируется C++Builder, а также могут задаваться свойства проекта.

Библиотека VCL, формы и компоненты

Библиотека визуальных компонентов — VCL-библиотека (Visual Component Library) — является репозитарием компонентов, используемых для создания приложений с помощью C++Builder. Компонентом называется объект, используемый для создания программы, — флажок, комбинированный список или рисунок. Эти компоненты выбираются с помощью щелчка левой кнопкой мыши и перемещаются в рабочую область. Компоненты VCL-библиотеки представляют собой код, который скомпилирован для выполнения определенных операций, что избавляет разработчика от необходимости всякий раз создавать его заново.

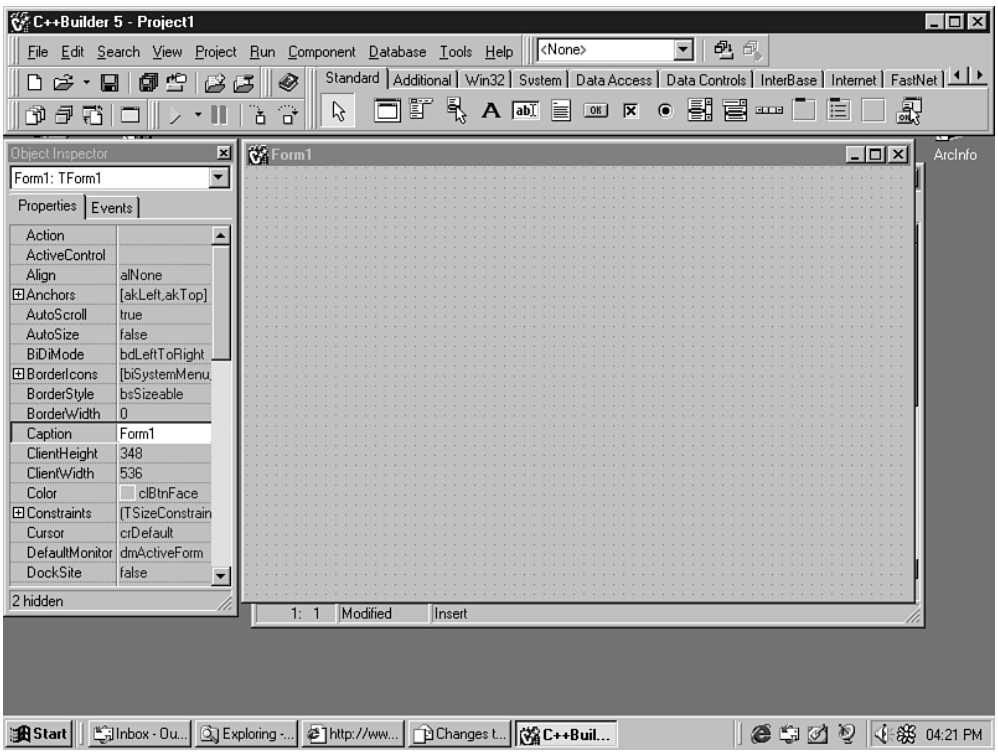


Рис. 1.1. Среда разработки C++Builder

Кроме того, разработчик может добавлять и создавать собственные компоненты, пользуясь способами, которые описаны в следующих главах. Таким образом, компоненты VCL-библиотеки избавляют разработчика от выполнения трудоемких и рутинных операций. Они располагаются в панели компонентов Component Palette и более подробно описаны ниже в этой главе.

Все компоненты обладают свойствами, которыми можно управлять с помощью кода или пакета C++Builder. Свойства компонента определяют способ его работы, внешний вид, набор функциональных возможностей и т.д. Их можно модифицировать с помощью вкладки свойств Properties в окне Object Inspector, которая располагается в левой части экрана под основной линейкой команд меню (см. рис. 1.1). Параметры свойств можно изменять с помощью кода, но без достаточного опыта работы с C++Builder и VCL-библиотекой это делать не рекомендуется.

Окно Object Inspector также содержит вкладку событий Events, в которой можно создавать события для определения способа взаимодействия пользователя с этим компонентом. Кроме того, эти события могут выполнять другие определенные разработчиком действия.

Форма

Форма — это окно, которое в большинстве случаев является пользовательским интерфейсом создаваемого приложения. Пустая форма создается автоматически при создании нового приложения с помощью C++Builder. Для создания пользовательского интерфейса в форму нужно просто добавить соответствующие визуальные компоненты, а затем указать их расположение и размер. В форму можно также добавить такие невизуальные компоненты, как тай-

меры. Во время создания приложения они будут иметь вид пиктограммы компонента, а во время запуска будут скрыты.

При запуске приложения пользователем форма по умолчанию отображается в центре экрана. Исходное расположение формы и другие ее параметры можно изменить, задавая соответствующие свойства формы с помощью окна Object Inspector.

Панель быстрого доступа к командам меню

В C++Builder можно использовать специальную кнопочную панель, которая обеспечивает быстрый доступ к таким наиболее часто используемым командам меню, как Run (Старт), Step Through (Пошаговая отладка), View Unit (Просмотр модуля), View Form (Просмотр формы) и Add to Project (Добавить в проект). Эту панель разработчик может настроить по своему желанию, добавляя в нее наиболее нужные команды и удаляя те, которые используются не очень часто.

Настройка панелей инструментов

Одним из основных достоинств среды разработки является возможность ее настройки. Кнопку быстрого доступа к команде меню можно добавить или удалить из любой панели инструментов, за исключением панели компонентов Component Palette, которая содержит компоненты, а не команды. В C++Builder предусмотрены следующие панели инструментов.

- Standard (Стандартная панель).
- View (Панель просмотра).
- Debug (Панель отладки).
- Custom (Панель настройки).
- Component Palette (Панель компонентов).

Для настройки панели инструментов C++Builder выполните следующее.

1. Щелкните правой кнопкой мыши в произвольном месте панели инструментов.
2. В контекстном меню выберите команду Customize (Настройка).
3. После этого на экране появится диалоговое окно Customize (Настройка панелей инструментов), которое показано на рис. 1.2.
4. Выберите вкладку Commands (Команды).
5. Теперь нужную команду можно зацепить и перетащить в панель инструментов.
6. Для удаления из панели кнопки быстрого вызова команды меню ее достаточно точно так же перетащить за пределы панели инструментов.

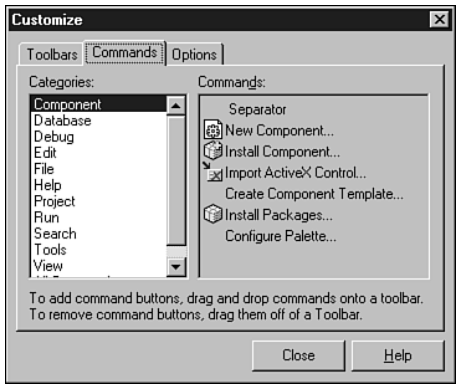


Рис. 1.2. Диалоговое окно настройки панелей инструментов Customize

Панель Component Palette

Панель (или палитра) компонентов Component Palette располагается сразу под основной линейкой команд меню и содержит все типы компонентов библиотеки VCL. Эти компоненты сгруппированы на вкладках по категориям, названия которых указаны над ними. Для выбора компонента щелкните на нем левой кнопкой мыши, а затем — в том месте формы, где его предполагается разместить. Как уже говорилось выше, свойства компонентов можно модифицировать с помощью окна Object Inspector или непосредственно с помощью кода.

События и обработчики событий

В этом первом уроке, посвященном C++Builder, попробуем разместить в форме кнопку, создадим для нее событие, а потом запустим созданное приложение.

Кнопки используются практически во всех приложениях Windows, поскольку этот простой объект дает возможность пользователю запустить некоторое событие.

Компонент Button (Кнопка) располагается во вкладке Standard панели Component Palette. Его пиктограмма имеет вид стандартной кнопки с надписью OK. Для размещения компонента Button в форме щелкните левой кнопкой мыши на этой пиктограмме, а затем — в центре формы. На рис. 1.3 показан пример формы с такой кнопкой.

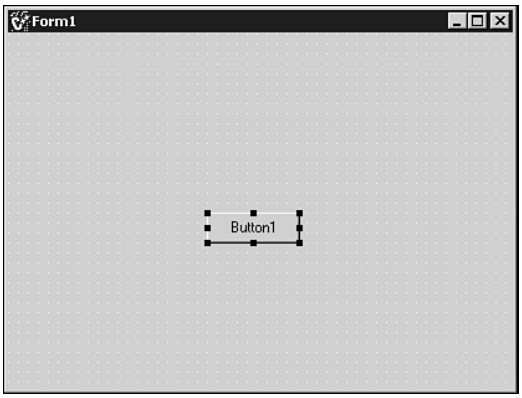


Рис. 1.3. Пример компонента Button, добавленного в форму

Вот и все! Кнопка уже добавлена в форму, и C++Builder автоматически создал экземпляр этой кнопки в коде приложения. Пока еще от этой кнопки никакой пользы нет, поскольку она ничего не выполняет. После компиляции и запуска приложения при щелчке на ней ничего не произойдет. Хотя обычно щелчок на кнопке приводит к выполнению некоторых действий, например, сохраняется информация, которую ввел пользователь, отображается сообщение для пользователя и т.д.

Во время выполнения приложения при щелчке на кнопке генерируется некоторое событие. Для того чтобы приложение смогло реагировать на такое событие, необходимо создать обработчик этого событий. Обработчиком события называется функция, которая вызывается автоматически после возникновения такого события. Во время создания приложения достаточно дважды щелкнуть на компоненте Button, чтобы C++Builder создал каркас обработчика `OnClick` для события, которое возникает при щелчке на этой кнопке во время работы приложения. То же самое можно сделать, выбирая эту кнопку, а затем щелкая дважды в текстовом поле OnClick во вкладке Events окна Object Inspector. После создания каркаса обработчика события в него можно добавить код для тех действий, которые должны быть выполнены после щелчка на этой кнопке.

Каркас обработчика события `OnClick` имеет следующий вид:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
}
```

Щелкнув правой кнопкой мыши в окне редактора кода Code Editor и выбрав из контекстного меню команду Open Source/Header File (Открыть исходный/заголовочный файл), можно увидеть следующий код:

```
//---------------------------------------------------------------
#ifndef Unit1
#define Unit1
//---------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
//---------------------------------------------------------------
class TForm1 : public TForm
{
__published:    // компоненты среды разработки
        TButton *Button1;
        void __fastcall Button1Click(TObject *Sender);
private:    // объявления пользователя
public:         // объявления пользователя
        __fastcall TForm1(TComponent* Owner);
} ;
//---------------------------------------------------------------
extern PACKAGE TForm1 *Form1;
//---------------------------------------------------------------
#endif
```

Этот код генерируется в C++Builder автоматически и показан здесь лишь для демонстрации того, что C++Builder может выполнить по умолчанию.

Добавим теперь в него код для отображения после щелчка на этой кнопке сообщения о том, что программа прекращает свою работу, с последующим закрытием созданного приложения. В окне редактора кода щелкните на вкладке Unit1.cpp, чтобы возвратиться к каркасу обработчика события для компонента Button. Введите следующий код внутри только что созданного каркаса структуры события:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    ShowMessage("Hello, world! Это тест! Нажмите OK");
    Close();
}
```

Этот код, вероятно, понятен даже новичкам. После запуска программы и щелчка на этой кнопке будет запущено событие. Созданный нами обработчик события `OnClick` отобразит на экране диалоговое окно с данным сообщением. После закрытия диалогового окна программа завершит работу благодаря вызванному методу `Close()`.

Запустим и посмотрим!

В панели ускоренного доступа щелкните на зеленой стрелке, которая выглядит, как кнопка Play на панели бытового магнитофона. Или выберите команду меню Run⇨Run. (Еще быстрее это можно сделать, нажав клавишу <F9>.) После щелчка на кнопке Run C++Builder

начнет компиляцию, а затем выполнение программы. После этого запущенное приложение будет ожидать щелчка на кнопке. После щелчка на кнопке OK на экране появится диалоговое окно с введенным нами сообщением, а затем программа завершит свою работу.

После просмотра результатов работы программы закроем текущий проект. Для этого выберите команду меню File⇨New Application. При этом в ответ на предложение со стороны C++Builder о сохранении проекта выберите No. Попробуем создать реальную программу, которая смогла бы выполнить что-нибудь стоящее.

Ваша первая рабочая программа

Выберите команду меню File⇨New Application. При этом C++Builder создаст новый проект и сгенерирует код для создания пустой формы. Теперь мы готовы приступить к созданию приложения с кодом минимального размера.

В панели Component Palette выберите вкладку Additional, а затем — компонент Image (Изображение). На его пиктограмме картинка: небо, холм и водоем у подножия холма. При размещении указателя мыши на этой пиктограмме контекстная подсказка сообщит название этого компонента.

После выбора компонента Image с помощью щелчка на нем поместите указатель мыши в форме и щелкните на ней один раз для размещения в ней этого компонента. При этом в форме появится квадратный контур компонента, который позволяет отображать графические объекты.

Во вкладке Properties окна Object Inspector выделите атрибут Stretch и задайте для него значение True.

В панели Component Palette перейдите во вкладку Dialogs и выберите компонент OpenDialog. Если потребуется, прокрутите панель Component Palette с помощью левой или правой стрелок. Компонент OpenDialog имеет вид открытой папки желтого цвета. Выберите ее, щелкнув левой кнопкой мыши, и поместите указатель в любом месте правого верхнего угла формы. Этот компонент теперь является частью вашей формы и используется для отображения диалогового окна, в котором можно выбирать и открывать файлы.

Теперь нужно установить значения атрибутов этого компонента. В окне Object Inspector найдите атрибут Filter во вкладке Properties и в его текстовом окне введите следующий текст:

```
BMP files|*.bmp
```

Во вкладке Standard панели Component Palette выберите два компонента Button. Компоненты можно размещать не по одному, а сразу несколько, удерживая клавишу <Shift> и щелкая нужное количество раз на пиктограмме компонента Button в панели Component Palette.

В результате этого после каждого щелчка мышью в форме появится новый компонент Button.

Теперь выберите белую стрелку селектора объектов Object Selector с левой стороны панели Component Palette, которая позволяет перемещать и выбирать компоненты или объекты в форме. Поскольку мы уже познакомились с кнопками, то попробуем поработать с другими типами компонентов.

Щелкните на компоненте `Button1` в форме для выбора этой кнопки. Попробуем теперь изменить атрибуты этого отдельно взятого компонента. В окне Object Inspector выберите атрибут Caption и замените словами `Get Picture` (Получить рисунок) имеющийся там текст. Атрибут Caption (Надпись) используется для отображения на кнопке информации для пользователя. Для создания новой надписи нужно удалить имеющийся там текст и ввести новый.

Перейдите во вкладку Win32 в панели Component Palette и выберите компонент StatusBar (Строка состояния), который имеет вид серой полоски с треугольником в левом нижнем углу. Строка состояния обычно располагается в нижней части приложения Windows и отображает информацию о его состоянии.

Поместите этот компонент в форму и вы увидите, что он автоматически разместится в нижней части формы.

Теперь дважды щелкните на компоненте Button, который уже находится в форме (панель Object Inspector отобразит Button1), для создания события. Используйте приведенный ниже код.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
   if(OpenDialog1->Execute())
   Image1->Picture->LoadFromFile(OpenDialog1->FileName);
    StatusBar1->SimpleText = OpenDialog1->FileName;
}
```

Остановимся ненадолго, чтобы объяснить, что происходит. Это окно Source Code Edit предназначено для редактирования кода приложения, и его можно открыть с помощью окна менеджера проектов Project Manager или с помощью окна Object Inspector. Именно в окне редактора кода приложения Source Code Edit вводится и редактируется код приложения.

Рассмотрим теперь более подробно саму форму. Для доступа к ней следует использовать кнопку Toggle Form/Unit панели ускоренного доступа к командам меню. Она содержит рисунок формы и листа бумаги со стрелками, указующими на форму с двух сторон. При размещении указателя мыши над этим компонентом будет отображена его контекстная подсказка Toggle Form/Unit (F12). Другой способ переключения между формой и кодом — использование клавиши <F12>; нажимая ее можно перемещать фокус ввода из кода в форму и обратно.

Щелкните на кнопке Button2 (в окне Object Inspector будет отображена кнопка Button2), которая уже размещена в форме, и укажите для надписи на ней текст **Close** (Закрыть). Дважды щелкните на этой кнопке и введите следующий код:

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{
    Close();
}
```

Теперь поместите снова фокус ввода в форму, щелкнув на ней. Изменения как будто не очень значительны? Вскоре вы увидите, как много действий может автоматически выполнить C++Builder на основе всего лишь нескольких введенных вами строк кода!

До запуска приложения расположите аккуратно кнопки и другие компоненты, чтобы придать форме привлекательный вид. Для выбора компонентов используйте белую стрелку с левой стороны панели Component Palette. Так как компонент StatusBar нельзя переместить, то его не следует упорядочивать вместе с кнопками.

Щелкните на зеленой стрелке Run или выберите команду меню Run⇨Run. При этом C++Builder начнет компиляцию программы при условии отсутствия каких-либо опечаток. В итоговом виде программа должна иметь две кнопки с надписями Get Picture и Close. Щелкните на кнопке Get Picture.

После этого на экране появится диалоговое окно Open с предложением выбрать файл с расширением .BMP. Откройте каталог операционной системы Windows на диске C: и выберите файл SETUP.BMP или какой-либо другой файл с расширением .BMP.

После выбора файла щелкните на кнопке OK и вы увидите в форме изображение Windows Setup. При этом в нижней части формы в строке состояния будет отображено название файла. Итак, вы создали ваше первое реально работающее приложение с минимально возможным кодом! Рассмотрим подробнее использованные нами компоненты.

Сначала в форме был размещен компонент Image, который позволяет отображать на экране файлы формата BMP в вашей программе. При этом среда разработки C++Builder сама проделала всю рутинную работу за вас.

Обычно для отображения графического файла в приложении для Windows требуется создать несколько страниц кода только для загрузки палитры цветов, которые используются файлом формата BMP. Кроме того, потребуется выполнить обработку исключительных ситуаций при анализе структуры BMP-файла, убедиться в том, что открываемый файл действительно имеет BMP-формат, задать разрешение и многое другое. И все это нужно только для того, чтобы отобразить сам рисунок. Как видите, C++Builder уже продела всю эту работу за вас. Все, что от вас требуется, — поместить компонент в форму и задать его свойства. Например, установить свойство Stretch, которое используется для подгонки рисунка под размеры компонента в форме. Для размещения больших рисунков, возможно, потребуется с помощью мыши также изменить размеры самого компонента.

После размещения в форме компонента `Image` в нее также был помещен компонент `OpenDialog`, который позволяет пользователю найти, выбрать и открыть нужный файл. В данном случае компонент `OpenDialog` использовался для выбора и открытия файла с изображением. При этом компонент `Image` определяет его принадлежность к разряду графических файлов, считывает его и отображает содержимое. Для свойства `Filter` (Фильтр) этого компонента было задано такое значение, которое указывало на поиск только файлов с расширением `.BMP`. При этом в диалоговом окне открытия файлов будет отображено название открываемого типа файла. Символ | используется как разделитель параметров, а `*.BMP` означает, что в диалоговом окне будут показаны только файлы с расширением `.BMP`.

Для просмотра или изменения параметров фильтра файлов во время создания приложения перейдите в форму `Form1` и выберите компонент `OpenDialog`, щелкнув на нем. Затем перейдите в окно Object Inspector. В разделе Filter дважды щелкните в текстовом окне — на экране появится окно редактора фильтра Filter Editor с отображением сведений, о которых было сказано выше.

Затем вы размещаете в форме два компонента `Button` и указываете для них надписи, которые идентифицируют эти кнопки. После этого для компонентов `Button` создаются события с помощью окна Object Inspector. В нем имеется вкладка Events, в которой перечислены все события компонента. Для установки некоторого события нужно дважды щелкнуть в текстовом окне этого события.

Для самостоятельного создания такого приложения с начала и до конца потребуется ввести более 20 страниц кода. Целью авторов было познакомить читателя в этой главе с основами программирования в C++Builder и продемонстрировать высокую скорость такой работы.

Попробуем теперь с помощью еще одного примера использования RAD-технологии в C++Builder создать программу с другими компонентами.

На заметку

Закройте текущий проект. Его не обязательно сохранять. Для этого выберите команду меню File⇨Close All и щелкните на кнопке No.

Затем для создания нового проекта выберите команду меню File⇨New Application. Сохраним этот проект, выбирая команду меню File⇨Save Project As.

Укажите новое имя для файла с кодом формы, заменяя при сохранении формы предлагаемое по умолчанию имя `Unit1.cpp` на `Mainfrm.cpp`.

После сохранения кода формы вам будет предложено сохранить код проекта. Замените предлагаемое по умолчанию имя `Project1.pbr` на `Proj.bpr`.

Поместите в форму два списка `ListBox`, которые находятся во вкладке Standard панели компонентов Component Palette. Расположите их на одной линии в любом месте формы. При этом C++Builder присвоит им имена `ListBox1` и `ListBox2`.

Поместите текстовое окно `EditBox` и две кнопки `Buttons` под этими списками. Эти компоненты также находятся во вкладке Standard панели компонентов Component Palette. Выровняйте их по своему усмотрению. В результате C++Builder создаст кнопки с именами `Button1` и `Button2`, а также текстовое окно с именем `Edit1`.

Выберите кнопку Button1, а затем в окне Object Inspector — свойство Caption и укажите для него название **ADD** (Создать).

Выберите кнопку Button2, а затем в окне Object Inspector — свойство Caption и укажите для него название **REMOVE** (Удалить).

Выберите текстовое окно Edit1, а затем в окне Object Inspector — свойство Text и удалите текстовую строку в нем.

Поместите элемент Label под текстовым окном Edit1, в окне Object Inspector выберите свойство Caption и укажите для него название **Friends' Names** (Имена друзей).

Для выбора самой формы щелкните где-нибудь на ней, но не на компонентах. То же самое можно сделать с помощью окна Object Inspector, выбирая в списке Form1. Выберите в окне Object Inspector вкладку Events и найдите в ней событие OnCreate. Дважды щелкните на нем, и C++Builder создаст показанный ниже код обработчика этого события.

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
}
```

Теперь вовнутрь этой структуры нужно поместить код, который будет выполнен после создания формы во время запуска приложения. Это значит, что при запуске программы Windows создаст форму и выполнит код этого события. Введите для этого события следующий код:

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    ListBox1->Items->Add("David Sexton");
    ListBox1->Items->Add("Randy Kelly");
    ListBox1->Items->Add("John Kirksey");
    ListBox1->Items->Add("Bob Martling");
}
```

Вернитесь в форму и дважды щелкните на кнопке Button1 с надписью Add для создания связанного с ней события. В код обработчика этого события введите следующие строки:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    String GetListItem =
       ListBox1->Items->Strings[ListBox1->ItemIndex];
    ListBox2->Items->Add(GetListItem);
}
```

Вернитесь в форму и дважды щелкните на кнопке Button2 с надписью Remove для создания связанного с ней события. В код обработчика этого события введите следующие строки:

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{
    ListBox2->Items->Delete(ListBox2->ItemIndex);
}
```

Вернитесь в форму и дважды щелкните на текстовом окне Edit1. Найдите событие OnKeyPress во вкладке Events окна Object Inspector. Щелкните дважды в текстовом поле этого события для создания обработчика этого события и введите в нем следующий код:

```
void __fastcall TForm1::Edit1KeyPress(TObject *Sender,
                                      char &Key)
{
    if (Key==13)
    {
```

```
        ListBox2->Items->Add(Edit1->Text);
        ListBox1->Items->Add(Edit1->Text);
    }
}
```

Теперь сохраним созданный проект. Для этого выберите команду меню File⇨Save All. После сохранения проекта попробуем создать приложение, нажимая комбинацию клавиш <Ctrl+F9> или выбирая команду меню Project⇨Make Proj.exe. В результате будет получен выполняемый файл этого приложения, конечно, при условии отсутствия ошибок и опечаток.

Запустим приложение, щелкнув на зеленой стрелке или выбрав команду меню Run⇨Run, и посмотрим, как оно будет работать. Рассмотрим вкратце принцип работы этого кода.

Приложение будет иметь вид обычного окна с двумя кнопками и двумя списками имен в них.

Текстовое окно (т.е. компонент `Edit1` в этом примере) готово для ввода имен ваших друзей.

После нажатия клавиши <Enter> каждое введенное имя будет добавлено не только в первый, но и во второй список.

Выберите одно из имен и щелкните на кнопке `Add`, тогда указанное имя появится только в правом списке. Если щелкнуть на кнопке Remove, то указанное имя будет удалено только из правого списка. Попробуйте выбрать некоторые имена и добавить в правый список, а затем удалить их.

Как известно, созданное нами событие `FormCreate` выполняется сразу же после создания формы. В нем содержался код для добавления строк в список `ListBox1`, и после создания формы эти строки действительно будут добавлены в список.

Другое событие связано с кнопкой `Button1`. Оно создает строку `GetListItem` типа `String`, которая содержит указанный пользователем элемент. Как событие определяет, что был выбран именно этот элемент? Очень просто: оно считывает индекс элемента. Если выбор еще не сделан, то индекс равен `null`. Следующая строка кода обработчика события добавляет строку на основе индекса из списка `ListBox1`.

Код обработчика события кнопки `Button2` еще меньше, чем код обработчика события первой кнопки. Он получает индекс элемента списка `ListBox2` и удаляет его.

В обработчике третьего события используется событие нажатия клавиши `OnKeyPress` для списка `Edit1`. Это событие возникает, когда пользователь вводит данные и нажимает клавишу. Это приводит к выполнению кода обработчика события. Это событие отслеживает нажатие клавиши <Enter>, код которой равен 13. Для этой цели можно также использовать значение `VK_ENTER`, которое в C++Builder обозначает клавишу <Enter>. Команда `if` проверяет истинность равенства кода 13 и переданного значения параметра `Key`. Если это так, то выполняется код в теле команды `if` и строка текстового окна добавляется в оба списка.

Итак, мы создали три события, введя код относительно небольшого объема. Кроме того, некоторые компоненты были размещены в форме без какого-либо кода вообще. В итоге мы получили рабочую программу с минимальным объемом кода.

Как видите, это приложение было создано всего за несколько минут. После знакомства с панелью Component Palette, окном Object Inspector и основными операциями интегрированной среды разработки можно приступать к изучению более сложных элементов C++Builder. Сравнивая время, необходимое для разработки с помощью других инструментов, например Visual C++ или Microsoft Foundation Classes (MFC), можно убедиться, что C++Builder намного превосходит их.

Читателю будет полезно также познакомиться с другими командами меню C++Builder. Контекстная справка поможет разобраться не только с назначением команд меню, но и с элементами окна Object Inspector, а также с некоторыми другими возможностями интегрированной среды разработки.

Что делать

В этом разделе собраны ответы на наиболее часто возникающие вопросы о работе с C++Builder.

- Как получить доступ к коду проекта и каждой формы?

  Это можно сделать с помощью окна Project Manager и команды меню Project. Команда меню Project⇨View Source позволяет отобразить основной код приложения. А для просмотра кода форм, включаемых файлов или файлов ресурсов, следует использовать окно Project Manager. Для открытия окна Project Manager выберите команду меню View⇨Project Manager или нажмите комбинацию клавиш <Ctrl+Alt+F11>.

- Как изменить свойства компонента?

  Это можно сделать с помощью окна Object Inspector. Для его открытия нажмите клавишу <F11> или выберите команду меню View⇨Object Inspector. Затем в окне Object Inspector выберите нужный компонент, и его свойства будут отображены на экране. Теперь можно приступать к указанию событий и редактированию свойств.

- Мне не удается точно расположить компоненты. Как это сделать вручную?

  Для перемещения компонента в указанное вами место удерживайте клавишу <Ctrl> и переместите этот компонент в выбранное место с помощью стрелок курсора. Это позволит вам расположить компоненты C++Builder с точностью до одного пикселя.

- После компиляции и запуска моего приложения оно как будто зависло. При этом на экране появилось множество странных окон, с которыми я не знаю что делать. Как от этого избавиться?

  Попробуйте перезапустить программу с помощью C++Builder. Для этого нажмите комбинацию клавиш <Ctrl+F2> или выберите команду меню Run⇨Program Reset. При этом выполнение вашего приложения будет полностью прекращено, все связанные с ним окна будут закрыты, и на экране отобразится код вашего приложения.

- Я хотел бы создать пиктограмму для своей первой программы и включить ее в программу. Как это сделать?

  Для этого рекомендуется использовать очень надежный и удобный графический редактор Image Editor. Откройте его, выбрав команду меню Tools⇨Image Editor. Создайте в нем изображение новой пиктограммы и сохраните его. Затем выберите команду меню Project⇨Option для открытия диалогового окна Project Options. Выберите вкладку Application и щелкните на кнопке Load Icon. Найдите вашу пиктограмму и щелкните на кнопке OK. После этого вам потребуется заново скомпоновать ваш проект, так как одна только компиляция или сборка проекта не позволит достичь цели. Заново скомпоновать весь проект можно, выбрав команду меню Project⇨Build All Projects. После этого ваше приложение будет содержать новую пиктограмму.

- Всякий раз при компиляции приложения моя форма получает название `Form1`. Как переименовать ее?

  Как известно, окно Object Inspector используется для указания свойств компонентов, но его можно также использовать для указания свойств форм. Для переименования формы следует отредактировать нужным образом значение атрибута `Caption`. Попробуйте поэкспериментировать с другими атрибутами окна Object Inspector.

- Существует ли более простой способ выбора команд меню?

  Да, для этого предусмотрена панель ускоренного доступа к командам меню, которая (по умолчанию) располагается над окном Object Inspector. Например, для создания

нового объекта приложения следует щелкнуть на кнопке с изображением белого листа бумаги. Для того чтобы узнать назначение кнопки, нужно подержать несколько секунд указатель курсора над ней и прочесть контекстную подсказку, которая появится над кнопкой.

- После размещения компонентов я случайно переместил их. Существует ли какой либо способ закрепления компонентов на их местах?

  Да, заблокировать перемещение компонентов можно с помощью команды меню Edit⇨Lock Controls.

Что нового в C++Builder 5

Как и в прошлых версиях Borland C++Builder и Borland Delphi, в C++Builder 5 имеется несколько новых элементов, которые впервые появились в Delphi 5. Новые возможности открываются в области Web-программирования, разработки приложений для распределенных вычислений, коллективной разработки, локализации приложений, отладки, разработки приложений для работы с базами данных, а также для повышения производительности работы программистов. Новые компоненты и усовершенствования подробно описываются далее в этой книге.

C++Builder 5 выпускается в трех версиях: стандартной — Standard (Std), профессиональной — Professional (Pro), а также корпоративной — Enterprise (Ent). Хотя стандартная версия имеет наименьшее число компонентов, она все же является мощным инструментом создания приложений для Windows и включает около 85 компонентов RAD-программирования, удостоенный многих наград компилятор, усовершенствованный отладчик и многое другое. Профессиональная версия содержит более 150 компонентов, включая новый инструмент защиты кода ***CodeGuard***™, средства многопроцессовой отладки и стандартные инструменты работы с базами данных. Корпоративная версия включает более 200 компонентов, включая ***Internet Express***, инструменты создания CORBA-компонентов, поддержку Microsoft SQL Server 7 и Oracle 8i, средства разработки ***MIDAS***, полный набор инструментов локализации, менеджер контроля версий ***TeamSource*** и многое другое.

В C++Builder 5 отсутствует лишь инструмент контроля версий PVCS Version Control фирмы Merant (прежде Intersolv).

На заметку

Полное описание всех новых компонентов в каждой версии C++Builder 5 можно найти в разделе “Feature List” на Web-сайте C++Builder по адресу `http://www.borland.com/bcppbuilder/`.

Эту информацию также можно получить в разделе “What’s New” в оперативной справке C++Builder.

Компоненты, перечисленные в следующих разделах, имеются в профессиональной и корпоративной версиях C++Builder 5 и отсутствуют в стандартной версии, за исключением особых замечаний. Далее при рассмотрении новых компонентов не будут указываться различия между версиями C++Builder, поэтому заинтересованному читателю предлагается обращаться к справочным материалам.

Web-программирование

Специалистам известно, что одним из достоинств C++Builder являются его инструменты разработки приложений для Web и Internet. В C++Builder 5 специалисты фирмы Borland также включили программу-мастер создания ASP-страниц ***Active Server Page (ASP) Application Wizard***, новый компонент ***Internet Express*** (Ent), новый компонент-броузер, а также усовершенствования брокера ***WebBroker***.

Новый компонент ***Internet Express*** (Ent) позволяет создавать "тонкие" клиенты для Web-среды для представления данных, полученных от MIDAS-серверов и внешних баз данных на основе языка XML и HTML 4. Тонкие клиенты обычно меньше по размеру, чем обычные (или "толстые") клиенты. Они не могут получить непосредственный доступ к базе данных, а потому на клиентском компьютере не требуется инсталлировать механизм доступа к базе данных Borland Database Engine (BDE).

WebBroker теперь включен также в профессиональную версию C++Builder 5 (ранее он присутствовал только в корпоративной версии) и способен выполнять предварительный просмотр на основе стандарта HTML 4. Еще одним новшеством для Web-программирования в C++Builder 5 является компонент Web-броузера, который позволяет интегрировать просмотр HTML-страниц в ваших приложениях. Этот, на первый взгляд, простой компонент может быть использован самыми разными способами.

- Для создания собственного Web-броузера, который, например, позволит пользователям просматривать Web-страницы только внутри корпоративной сети intranet.
- Для создания пользовательского интерфейса вашей программы в HTML-формате. Этот подход имеет несколько преимуществ. Во-первых, в создании пользовательского интерфейса могут принять участие не только профессиональные программисты, но и дизайнеры Web-страниц вашей компании. Кроме того, пользовательский интерфейс можно настроить динамически для пользователей определенного уровня за счет простой загрузки другого HTML-файла. Пользовательский интерфейс можно также особым образом настроить под конкретного клиента. При этом доставка новой версии пользовательского интерфейса выполняется так же просто, как и загрузка HTML-файла по сети, т.е. именно то, что броузеры умеют делать особенно хорошо.
- Для интеграции файлов справки на основе стандарта HTML, что позволяет создавать их так же просто, как Web-страницы.

В некоторых программных продуктах, например в Microsoft Encarta и Microsoft Office 2000, компонент Web-броузера используется именно так. Провайдеры America Online (AOL) и CompuServe используют компонент Web-броузера для создания их собственных приложений-броузеров.

Приложения для распределенных вычислений

В CORBA (Ent) теперь предусмотрена поддержка брокера объектных запросов ***VisiBroker ORB*** версии 4.00 и совместимых с CORBA 2.3 клиентов и серверов. Среди других новых компонентов CORBA следует отметить ***Portable Object Adapter*** и ***Objects By Value***, которые являются усовершенствованиями ***Visual TypeLibrary Editor***, ***Interface Repository***, ***CORBA Wizard*** и т.д.

В ***MIDAS*** (Ent) теперь предусмотрена поддержка пакетов данных XML, ***DataBroker*** без записи информации о состоянии, нового компонента Web Connection, пула серверных объектов и функций провайдера.

Коллективное создание приложений

Замечательным новым компонентом C++Builder 5 является инструмент коллективного создания приложений ***TeamSource***. Он входит в состав корпоративной версии и может быть приобретен дополнительно для использования его совместно с профессиональной версией.

TeamSource является внутренним вариантом менеджера версий, который позволяет параллельно работать над одним проектом сразу нескольким программистам. При этом каждый программист работает со своими собственными копиями проекта. В большинстве случаев

TeamSource автоматически управляет слиянием внесенных изменений, а визуальная утилита сравнения упрощает задачу разрешения конфликтов. ***TeamSource*** отслеживает версии и позволяет отмечать отдельные состояния проекта.

Реальный внешний контроль версий выполняется отдельным специализированным программным обеспечением. Для этой цели в корпоративной версии C++Builder 5 предусмотрено использование внешней версии пакета Borland ZLib, а также поддержка интерфейса для менеджера версий Merant PVCS (который можно приобрести отдельно). Для поддержки других внешних вариантов менеджера версий при работе с ***TeamSource*** можно создать отдельные подключаемые модули.

Локализация приложений

В корпоративную версию C++Builder 5 Enterprise впервые включены компоненты ***Translation Suite***, ***Translation Repository***, ***RC Translator*** и ***DFM Translator***. Их можно приобрести отдельно для использования совместно с профессиональной версией C++Builder Professional. Они позволяют интернационализировать или локализовать приложение для других языков или культур. В них также предусмотрена возможность создания приложения для работы сразу в нескольких разных языковых и культурных средах.

Программа-мастер ***Resource DLL Wizard*** обладает улучшенными средствами для перевода приложений на другие языки.

Отладка

Вероятно, одним из наиболее полезных новых компонентов для программистов C++Builder является ***CodeGuard***. ***CodeGuard*** — это инструмент для выявления ошибок на этапе выполнения программы, который позволяет обнаружить утечку памяти и других ресурсов и определить причину этого. Кроме того, будут полезны также другие новые инструменты отладки для просмотра работы FPU/MMX-функций процессора, средства установки отдельных и групповых контрольных точек (во всех версиях) для программируемого контроля и управления сразу несколькими контрольными точками.

Разработка приложений для работы с базами данных

Для работы с базами данных в C++Builder 5 предусмотрено сразу несколько новых компонентов, включая ***DataModule Designer***, ***InterBase Express*** и ***ADO Express***.

DataModule Designer позволяет просматривать и проектировать родительски-дочерние отношения среди компонентов доступа к данным в древовидной иерархии, а также зависимости типа сущность-связь (включая связи типа основные-вспомогательные сведения) в режиме просмотра диаграммы данных.

С помощью компонента ***InterBase Express*** можно создавать высокопроизводительные приложения для работы с базами данных, не прибегая к использованию Borland Database Engine (BDE). Для улучшенной отладки методов доступа к данным в нем предусмотрен новый SQL-монитор.

ADOExpress является новым набором компонентов для доступа к реляционным и нереляционным базам данных на основе технологий ActiveX Data Object (ADO) и OLEDB фирмы Microsoft. С помощью ADO можно осуществлять доступ к стандартным базам данным, электронной почте, файловым системам, электронным таблицам и другой информации. Для приложений на основе ADOExpress не требуется использовать BDE.

Помимо этих новых компонентов, в C++Builder 5 теперь предусмотрена поддержка InterBase 5.6, MS SQL Server 7 и Oracle 8i.

Повышение производительности работы разработчиков

В новой версии предусмотрено несколько новых компонентов для повышения производительности работы разработчиков. Многопоточная фоновая компиляция (во всех версиях) не мешает программисту продолжать свою работу во время работы компилятора. Возможность настройки рабочей среды Custom Desktop Settings (во всех версиях) и включения режима авто-отладки позволяет организовывать и сохранять разные типы интегрированных сред разработки. Т.е. для отладки приложений используется отдельный макет рабочей среды, который можно включать и выключать по мере необходимости с одновременной автоматической сменой настроек рабочей среды.

Интегрированный менеджер списка неотложных дел позволяет вставлять простые напоминания с комментариями и указаниями приоритетности ваших проектов. При этом они могут быть вставлены в код как напоминание о том, что к этому месту следует вернуться позже для завершения работы. Для работы с этим списком задач предусмотрен отдельный режим.

В новую версию включено несколько программ-мастеров, включая ***Windows 2000 Client Logo Application Wizard*** (во всех версиях), ***Console Wizard*** (во всех версиях), ***Control Panel Applet Wizard*** и даже простые программы-мастера ***C Application Wizard*** и ***C++ Application Wizard*** (во всех версиях).

Кроме того, предусмотрена улучшенная поддержка Microsoft Visual C++, включая поддержку проектов Visual C++ 6.0, Microsoft Foundation Class (MFC) 6.0 и поддержку Active Template Library (ATL) 3.0.

В новой версии C++Builder 5 есть также много других новых инструментов и усовершенствований, повышающих производительность работы программиста, включая подмену параметров проекта для отдельных файлов (во всех версиях), категории свойств (во всех версиях) и отображения свойств (во всех версиях) в окне Object Inspector; программируемое отображение клавиш в редакторе кода (во всех версиях); новые компоненты (во всех версиях), а также возможность импорта серверов автоматизации как компонентов.

Сопроводительный компакт-диск с инструментами сторонних разработчиков

Профессиональная и корпоративная версии C++Builder 5 поставляются с сопроводительным компакт-диском с более чем 40 инструментами сторонних разработчиков. Одни из них являются полными версиями, другие — условно бесплатными и пробными версиями. Ниже приводится краткая характеристика некоторых полных и бесплатных версий этих инструментов.

- Debug Server 1.1 фирмы Elitedevelopments — это менеджер результатов отладки COM/DCOM-технологии, который позволяет централизованно собирать, отображать отладочные сообщения и управлять ими.
- 3D Engine SDK 3.0 и WorldBuilder 3.1 фирмы Morfit позволяет вставлять 3D-графику в приложения и разрабатывать аркадные игры.
- GExperts — это программный продукт с открытым кодом, который содержит десятки усовершенствований среды разработки, предназначенные для ускорения процесса создания приложений. В нем предлагается несколько программ-экспертов (“Experts”) для перемещения, поиска и преобразования кода. GExperts также содержит клавиши для ускорения выполнения наиболее часто встречающихся задач программирования, для быстрого доступа к нужной информации и настройки среды разработки.

Модернизация и совместимость

C++Builder 5 был официально выпущен 22 марта 2000 года. Во время создания этой книги было известно о существовании некоторых проблем совместимости C++Builder 5 с существующими проектами и инструментами сторонних разработчиков, а также некоторых других проблем. Дело в том, что пакет обновлений (patch) в то время еще не был выпущен в свет. Наиболее свежий список всех известных ошибок C++Builder 5 можно найти в разделе `Updates and Patches⇨Investigate Bugs` на Web-сайте Borland C++Builder по адресу `http://www.borland.com/bcppbuilder/`.

Для получения самой свежей информации об этом можно подписаться на рассылку технических новостей о C++Builder на Web-сайте фирмы Borland. Кроме того, в файле `README.TXT` в корневом каталоге инсталляционного компакт-диска каждой версии C++Builder можно найти самые последние сведения о ней, известные на момент выпуска данной версии.

Модернизация предыдущей версии C++Builder

C++Builder версии 5 может сосуществовать на одном компьютере с более ранними версиями C++Builder, но такие совместно используемые приложения, как среда разработки, будут обновлены до самой свежей версии. Кроме того, C++Builder 5 может сосуществовать с Delphi с соблюдением тех же общих правил.

Если вам необходимо использовать более раннюю версию C++Builder, то в таком случае ее необходимо деинсталлировать перед инсталляцией C++Builder 5. Дополнительную информацию об инсталляции и модернизации можно прочесть в файлах `INSTALL.TXT` и `README.TXT` в корневом каталоге инсталляционного компакт-диска.

Использование существующих проектов в C++Builder 5

При загрузке проекта, созданного с помощью предыдущей версии C++Builder, этот проект автоматически конвертируется в формат C++Builder 5. Файл проекта модернизируется и преобразуется к формату XML, а перечисленные в ней VCL-библиотеки прежних версий модернизируются до библиотек C++Builder 5. Кроме того, в код основного проекта также вносятся небольшие изменения в предложения `USE*`.

Возможно, вам уже приходилось создавать код специально для какой-либо версии или использовать программные продукты сторонних разработчиков, которые создают такой код. Для изменения свойств и методов в VCL-библиотеке в соответствии с требованиями разных версий C++Builder необходимо использовать предложения `#ifdef VERXXX` для корректной компиляции разных версий. Если какой-либо чувствительный к номеру версии код для организации прямой совместимости проверяется явно на принадлежность к C++Builder версии 4 или более ранним версиям без обработки с помощью конструкции `catch...all`, то эти предложения придется обновить.

Например, если выполняется проверка версии для кода, специфичного для C++Builder версии 4, с помощью предложения `#ifdef VER125`, то вам придется добавить либо предложения `#ifdef VER130` и `#endif`, либо `#else` для обработки с помощью конструкции `catch...all` для указания кода специфичного для C++Builder версии 5.

Если у вас нет исходного кода, что обычно бывает с компонентами или пакетами сторонних разработчиков, то необходимо получить от поставщика его обновленную версию.

Создание проектов, совместимых с предыдущими версиями C++Builder

Если необходимо создать приложение, которое будет использоваться с более ранними версиями C++Builder, то вы должны иметь в виду следующее. Нельзя использовать в формах и коде проекта такие новые компоненты, как свойства, методы или события. Кроме того, существует несколько новых параметров проекта, компилятора и компоновщика, которые не следует использовать. Файл справки содержит более подробную информацию о новых компонентах C++Builder 5.

В C++Builder 5 файлы форм хранятся в текстовом формате, а не в двоичном, как в предыдущих версиях. Для сохранения формы в двоичном формате следует щелкнуть правой кнопкой мыши на форме и снять флажок параметра Text DFM. Если вы хотите, чтобы все создаваемые впоследствии формы сохранялись в двоичном формате, то следует снять флажок параметра New forms as text (Сохранять новые формы в текстовом формате) во вкладке Preferences (Предпочтения) диалогового окна Environment Options (Параметры среды), которое появляется при выборе команды меню Tools⇨Environment Options (Инструменты⇨Параметры среды).

Файлы проектов сохраняются в формате XML. В предыдущих версиях C++Builder для сохранения файла проекта использовался формат сборочного файла (или make-файла). Такой файл можно экспортировать с помощью команды меню Project⇨Export Makefile. При этом потребуется переименовать make-файл с расширением `.bpr` и внести несколько изменений в значениях параметров, перечисленных в этом файле, включая номер версии C++Builder, заданный параметром `VERSION`, номера версий VCL-библиотеки в параметрах `LIBRARIES`, `SPARELIBS`, `PACKAGES` и `CFLAG1`. Преобразование файла проекта к предыдущим версиям выполняется несколько сложнее, поскольку для этого могут потребоваться также другие изменения.

Новым компонентом версии C++Builder 5 является список неотложных задач. Однако наличие в коде упоминаний о локальных и глобальных неотложных задачах никак не повлияет на его работу в предыдущих версиях C++Builder.

Если при открытии проекта C++Builder 4 появляется сообщение `Error reading symbol file` (Ошибка при считывании символов из файла), то это приложение следует скомпоновать заново. Дело в том, что символьные файлы C++Builder 4 несовместимы с C++Builder 5.

Другие вопросы модернизации проекта

В C++Builder 5 появилось несколько новых параметров для существующих методов, данных, классов и компонентов VCL-библиотеки. Вот основные изменения, внесенные в новую версию.

- `Tpoint`. `Tpoint` больше не является структурой. В синтаксисе инициализации фигурные скобки ({ и }) заменены на формальное присвоение типа `Tpoint` двум координатам. Более подробную информацию об этом можно найти в разделе “Tpoint, compatibility issues” оперативной справки.
- `TPropertyEditor`. Список параметров для конструктора `TPropertyEditor` теперь имеет вид `TPropertyEditor(const di_IFormDesigner Adesigner, int AropCount)`.
- `TComponentList`. Для хранения и поддержания списка компонентов применяется новый класс `TComponentList`, определение которого содержится во включаемом файле `cntnrs.hpp`. Более подробную информацию об этом можно найти в оперативной справке по `TComponentList`.
- `AnsiString sprintf`. Метод `sprintf()` класса `AnsiString` теперь записывает новую строку вместо текущей, а не добавляет к ней новую строку. Прежние функциональные обязанности метода `sprintf()` теперь выполняет новый метод `cat_sprintf()`.

- `TCppWebBrowser` и `THTML`. Компонент `TCppWebBrowser` в Web-странице заменяет компонент `THTML` фирмы Netmasters. Более подробную информацию об этом можно найти в оперативной справке.
- `MIDAS`. Некоторые изменения были внесены в `MIDAS`; подробную информацию об этом можно найти в оперативной справке.

Помимо перечисленного выше, существует много других аспектов модернизации и совместимости с C++Builder 5. Более подробную информацию об этом можно найти в оперативной справке во вкладке Contents в разделе "Upgrading to Borland C++Builder 5".

Преобразование кода Delphi в код C++Builder

Этот раздел посвящается не обучению языку C++, а, скорее, программированию в C++Builder для тех, кто обладает навыками программирования в Delphi. Прежде всего следует отметить, что при программировании в C++Builder учитывается регистр символов. Это может вызвать затруднения, если вы привыкли программировать в Delphi, но к этому можно быстро привыкнуть.

В отличие от программирования в среде Delphi, при программировании на языке C++ необходимо учитывать регистр символов.

Здесь будет предпринята попытка сравнить Delphi и C++Builder в основных областях программирования. В результате этого обсуждения читатель сможет легко и просто преобразовывать код Delphi в код C++Builder.

Комментарии

Прежде всего необходимо усвоить, как в C++Builder вводятся комментарии. В табл. 1.1 сравниваются способы создания комментариев в Delphi и C++Builder.

Таблица 1.1. Способы ввода комментариев в Delphi и C++Builder

Delphi	C++Builder
{ Комментарий в Delphi }	/* Комментарий в C++Builder */
(* Еще один комментарий в Delphi *)	
// Строка комментария	// Строка комментария

Переменные

Так же, как и в Delphi, в C++Builder прежде чем использовать какой-либо тип переменной, его необходимо объявить. В табл. 1.2 типы переменных в Delphi сравниваются с типами переменных C++Builder с указанием их длины.

В этой таблице перечислено большинство переменных, а более подробную информацию можно найти в справке C++Builder.

Таблица 1.2. Сравнение типов переменных в Delphi и C++Builder

Тип переменной	Размер (в байтах)	Delphi	C++Builder
Целая со знаком	1	ShortInt	char
	2	SmallInt	short, short int
	4	Integer, LongInt	int, long
	8	Int64	__int64
Целая без знака	1	Byte	BYTE, unsigned short
	2	Word	unsigned short
	4	Cardinal, LongWord	unsigned long
Действительная с плавающей запятой	4	Single	float
	8	Double	double
	10	Extended	long double
Универсальная	16	Variant, OleVariant, TVarData	Variant, OleVariant, VARIANT
Символьная	1	Char	char
	2	WideChar	WCHAR
Динамическая строковая	-	AnsiString	AnsiString
Строковая с завершающим нулем	-	PChar	char *
Строковая широкая с завершающим нулем	-	PWideChar	LPCWSTR
Динамическая строковая 2-байтовая	-	WideString	WideString
Указатель	8	Pointer	Void *
Логическая	1	Boolean	bool

Константы

Существует два способа объявления констант в C++Builder. Прежний способ был основан на следующем варианте использования директивы препроцессора `#define`:

```
#define myconstant 100
```

Новый и более безопасный способ определения констант основан на применении ключевого слова `const`:

```
const int myconstant = 100;
```

Здесь `myconstant` определена не только как константа, но и как величина целого типа.

Операторы

В этом разделе рассматриваются операторы C++Builder, а также способы преобразования операторов Delphi в операторы C++Builder. В табл. 1.3 перечислены типы операторов, которые используются в Delphi, и их аналоги в C++Builder.

Таблица 1.3. Сравнение операторов присвоения в Delphi и C++Builder

Оператор	Тип оператора	Delphi	C++Builder
Присвоение	Присвоить	:=	=
	Сложить, а потом присвоить	Нет	+=
	Отнять, а потом присвоить	Нет	-=
	Умножить, а потом присвоить	Нет	*=
	Разделить, а потом присвоить	Нет	/=
	Разделить по модулю, а потом присвоить	Нет	%=
	Побитовая операция And, а потом присвоение	Нет	&=
	Побитовая операция Or, а потом присвоение	Нет	|=
	Побитовая операция Xor, а потом присвоение	Нет	^=
	Побитовая операция Shl, а потом присвоение	Нет	<<=
	Побитовая операция Shr, а потом присвоение	Нет	>>=
Сравнение	Равно	=	==
	Не равно	<>	!=
	Больше	>	>
	Больше или равно	>=	>=
	Меньше	<	<
	Меньше или равно	<=	<=
Арифметические операции	Сложить	+	+
	Отнять	-	-
	Умножить	*	*
	Разделить действительные числа	/	/
	Разделить целые числа	div	/
	Разделить по модулю	mod	%
Логические операции	And	and	&&
	Or	or	||
	Not	not	!
Побитовые операции	and	And	&
	or	Or	|
	xor	Xor	^

Окончание табл. 1.3

Оператор	Тип оператора	Delphi	C++Builder
	not	Not	~
	Сдвиг влево	Shl	<<
	Сдвиг вправо	Shr	>>
Указатели	Получение типа объекта по указателю	^Type	Type*
	Разыменование (получение значения объекта по указателю)	pointer^	*Pointer
	Адрес переменной	@Variable	&Variable
	Ссылка	var	Type&
Класс	Объявление класса	class	class
	Объявление структуры	record	struct
	Указание области видимости	.	::
	Прямой доступ	.	.
	Косвенный доступ	Нет	->
Другие	Инкремент	Inc()	++
	Декремент	Dec()	--
	Строковая кавычка	'	"

Операторы присвоения

В C++ различные операторы можно комбинировать с оператором присвоения (+=). Например, выражение

```
x = x + 2;
```

можно переписать таким образом:

```
x += 2;
```

Операторы инкремента и декремента

В C++ операторы инкремента и декремента можно использовать в префиксной и постфиксной записи. Вот пример такого применения этих операторов:

```
x = ++y; // префиксная запись
x = y++; // постфиксная запись
```

Первое выражение указывает компилятору сначала увеличить на единицу значение переменной y, а затем присвоить переменной x новое значение y. Второе выражение имеет иной смысл. Сначала переменной x присваивается значение переменной y, и только значение переменной y увеличивается на единицу. Рассмотрим эти правила на следующем примере:

```
int x=5, y=5;
x = ++y;
// здесь x=y=6
//   затем
x = y++;
// теперь x=6 и y=7
```

Условные операторы

В C++ только условный оператор (`?:`) содержит три выражения и возвращает одно значение. Вот его синтаксис:

```
(выражение1) ? (выражение2) : (выражение3)
```

Этот оператор возвращает значение выражения `выражение2`, если выражение `выражение1` возвращает значение `true`. В противном случае оно возвращает значение выражения `выражение3`. Например, выражение

```
return ((a >b)? c : d);
```

можно переписать, используя конструкцию `if-else`, в следующем виде:

```
if (a>b) return (c)
else return  (d);
```

Оператор (*)

Оператор (*) используется для объявления указателей. Этот же символ применяется для разыменования указателя, и в таком случае он называется *оператором разыменования* (*dereference operator*). Разыменование представляет собой способ получения значения объекта, к которому отсылает данный указатель. Компилятор способен уловить это различие. Рассмотрим следующий пример:

```
int x, y = 8;
int* ptr = &y;  // объявление указателя ptr и его инициализация адресом y
x = *ptr;       // разыменование указателя ptr.
```

А вот как то же самое выглядит в Delphi:

```
var
x,y : Integer;
ptr : ^Integer;
begin
y:=8;
ptr := @y;
x := ptr^;
end;
```

Операторы (*) и (&) в C++ эквивалентны операторам (^) и (@) в Delphi. Оператор (&) называется оператором взятия адреса и используется для извлечения адреса переменной в оперативной памяти.

Оператор (&) также используется для объявления ссылок. Ссылка — это всего лишь особый вид указателя, который можно рассматривать как обычный объект. Ссылки очень удобны для передачи параметров в функциях. В Delphi параметры, передаваемые по ссылке, также называются переменными параметрами. В этом случае оператор (&) в C++ эквивалентен ключевому слову `var` в Delphi.

Операторы `new` и `delete`

Переменную можно объявить таким образом:

```
char buffer[255];
```

Для переменной `buffer` выделяется место в стеке, а сама она называется локальной переменной. Использование локальных переменных связано с двумя проблемами. Во-первых, они удаляются после возвращения функцией значения. Во-вторых, размер памяти, которая выделяется для стека, ограничен.

Эти проблемы можно решить, выделяя память в куче:

```
char* buffer;
buffer = new char;
```

То же самое можно записать в одной строке:

```
char* buffer = new char;
```

Теперь эту переменную можно использовать в любом месте программы, задавая для нее любой размер. Однако при этом следует помнить, что выделенная для нее память должна быть впоследствии освобождена с помощью оператора `delete`, как показано ниже.

```
delete buffer;
```

Операторы для работы с классами

Существует два способа доступа к членам и функциям класса на основе прямого и косвенного доступа. Более подробные сведения об этом можно найти в разделе о классах ниже в этой главе.

Управление потоком выполнения программы

Так же, как и в Delphi, в C++Builder имеется несколько структур для организации условного ветвления и введения циклов:

- выражения типа `if-else`;
- выражения типа `switch`;
- циклы `for`;
- циклы `while` и `do-while`.

В C++Builder также предусмотрены команды `break` и `continue`, которые аналогичны одноименным командам в Delphi и предназначены для прерывания или продолжения потока выполнения. В табл. 1.4 показаны примеры организации условного ветвления и введения циклов в Delphi и C++Builder.

Таблица1.4. Сравнение операторов ветвления и введения циклов в Delphi и C++Builder

Выражение	Delphi	C++Builder
if-else	if (i = значение) then begin выражение1; ...; end;	if (i == значение) { выражение1; ...;}

	if (i = значение1) then
	begin выражение1; ...; end
	else if (i = значение2)
	then begin выражение1; ...; end
	else
	begin выражение1; ...; end;
		if (i == значение1)
		{ выражение1; ...;}
		else if (i == значение2)
		{ выражение1; ...;}
		else
		{ выражение1; ...;}

switch	case <выражение_селектор> of
	значение1 :
	выражение;
	значение2 :
	выражение;
	
	else выражение;
	end; // конец case
		switch (<выражение>){
		case значение1:
		выражение; break;
		case значение2:
		выражение; break;
		
		default : выражение;
		} // конец switch

Окончание табл. 1.4

Выражение	Delphi	C++Builder

for	for i := значение1 to значение2 do
	begin выражение1; ...; end;
	// инкремент только на 1
	for i:= значение1 downto значение2
	do
	begin выражение1; ...; end;
	// декремент только на 1
		for(i=значение1;i<=значение2;
		i+=inc)
		{ выражение1; ...;}
		// инкремент на любое целое
		// значение inc
		for(i=значение1;i>=значение2;
		i-=dec)
		{ выражение1; ...;}
		// декремент на любое
		// целое значение dec

While	while i = значение do begin выражение1; ...; end;	while (i == значение) { выражение1; ...; }
do-while	repeat выражение1; until (i = значение)	do { выражение1; } while (i != значение);

Ниже приведены рекомендации по использованию этих команд и операторов.

- Ключевое слово `break` следует поместить в конце выражения каждого оператора `case` для прекращения его выполнения. В противном случае будет продолжено выполнение выражений, которые содержатся после следующих операторов `case`, до тех пор, пока не будет достигнут оператор `break` или конец оператора `switch`. Такой способ работы выглядит довольно странно, но бывают случаи, когда он удобен.
- Выражение-селектор в операторе `switch` не может содержать непорядковые типы, например строки.
- Цикл `for(;;)` эквивалентен циклам `while(true)` или `while(1) loop`. Все они являются бесконечными циклами, а потому для выхода из цикла следует использовать ключевое слово `break`.

Функции и процедуры

Так же, как и в Delphi, в C++Builder до использования функции она должна быть объявлена (т.е. создан ее прототип). В отличие от Delphi, в C++Builder нет специальных ключевых слов, как `function` и `procedure`, которые используются для объявления функций. В табл. 1.5 приведены примеры объявления функций в Delphi и C++Builder.

Таблица1.5. Примеры объявления функций в Delphi и C++Builder

Delphi	C++Builder
`procedure Add;`	`void Add();`
`procedure Add(x, y: Integer);`	`void Add(int x, y);`
`function Add: Integer;`	`int Add();`
`function Add (x, y: Integer): Integer;`	`int Add (int x, y);`

Любая программа на языке C или языке C++ должна содержать функцию `main()`, которая является точкой входа программы. Для приложений с графическим интерфейсом пользователя такая функция называется `WinMain()`. Функция `main()` работает так же, как и любая другая функция: она принимает параметры и возвращает значения. На рис. 1.4 показан пример использования функции `main()`для создания нового консольного приложения. Обратите внимание на то, что новое консольное приложение можно создать с помощью специальной программы-мастера ***Console Wizard***, выбирая команду меню File⇨New⇨Console Wizard.

Функция `main()` принимает два параметра — `argc` и `argv` — и возвращает целочисленное значение. В отличие от других функций, ее нельзя вызвать непосредственно — она вызывается автоматически после запуска программы.

Для открытия блока функции в языке C++ используется открытая фигурная скобка ({), а для закрытия — закрытая (}). В Delphi они эквивалентны ключевым словам `begin` и `end`, соответственно.

На рис. 1.4 показано, что в C++Builder для возвращения значения в функции используется ключевое слово `return`. Оно также вызывает прекращение работы вызванной функции (в данном случае `main()`). При его использовании следует проявить осторожность. В отличие от ключевого слова `Result`, которое используется в Delphi, ключевое слово `return` прекращает выполнение функции немедленно, поэтому любые выражения после него будут игнорироваться.

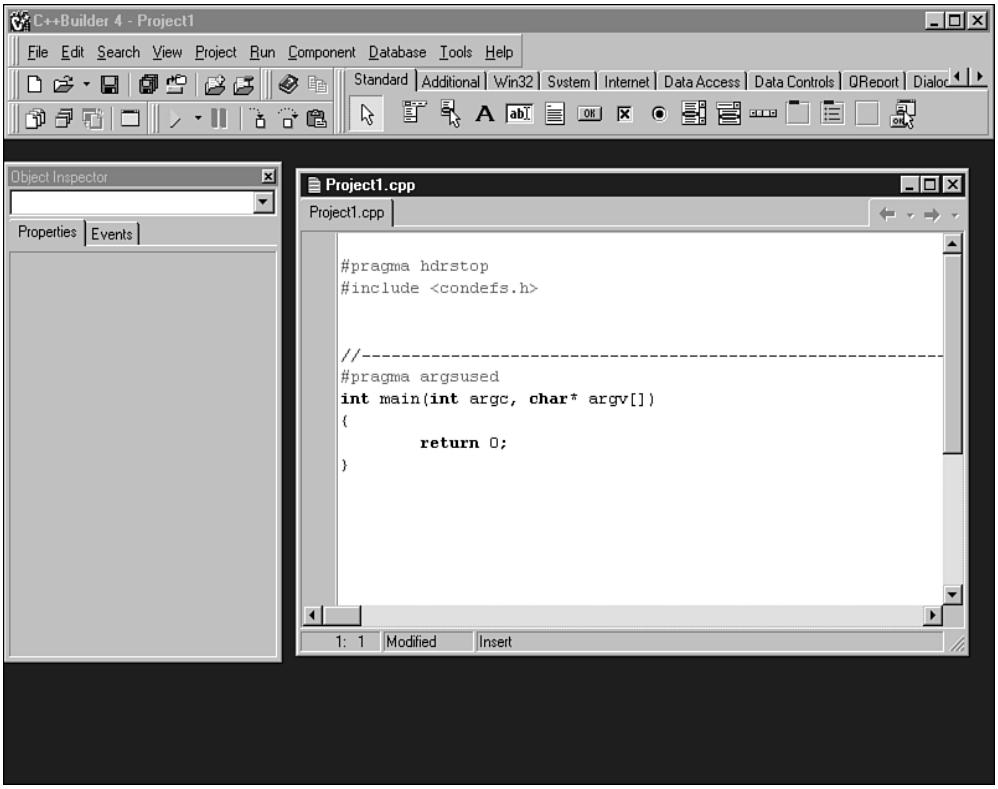


Рис. 1.4. Окно редактора сода C++Builder с текстом нового консольного приложения.

Классы

В Delphi и C++Builder используется одинаковый способ управления доступом к функциям. Они скрывают эти функции в структурах под названием *классы*. Включение нескольких функций в один класс для выполнения какой-либо задачи называется *инкапсуляцией*.

Структура `struct` унаследована из языков C, C++ и практически идентична классу. Единственное отличие между структурами и классами заключается в том, что члены-данные и члены-функции в структуре по умолчанию являются открытыми (`public`), а в классе — закрытыми (`private`).

Так же, как и в Delphi, класс в C++Builder имеет следующие компоненты:

- конструктор и деструктор,
- инструменты управления доступом к членам-данным и членам-функциям,
- указатель `this`, который эквивалентен указателю `self` в Delphi.

Одним из больших достоинств классов в C++ является то, что они могут применяться для организации *множественного наследования*. Множественное наследование возникает в ситуации, когда некий класс является производным от двух или более базовых классов. В Delphi (а фактически, в языке Pascal) класс может быть производным только от одного базового класса.

Рассмотрим синтаксис класса для разных типов наследования.

- Нет наследования.

```
class MyClass {   // нет наследования
// По умолчанию все члены являются закрытыми,
// а потому здесь помещаются объявления закрытых членов
private:
// Объявление открытых членов
protected:
// Объявление защищенных членов
public:
// Объявление закрытых членов
} ;  // Описание класса MyClass завершается точкой с запятой
```

- Единичное наследование.

```
class MyClass: BaseClass1
{ ...} ;
```

- Множественное наследование.

```
class MyClass: BaseClass1, BaseClass2,  BaseClass3
{ ...} ;
```

Конструкторы и деструкторы

Каждый класс имеет конструктор и деструктор. Если программист не создал какой-то особый конструктор или деструктор, то компилятор создаст их по умолчанию.

Основное назначение конструктора заключается в выделении пространства для класса и инициализации членов-данных класса. Задача деструктора состоит в освобождении этого выделенного пространства.

Класс может иметь несколько конструкторов, но деструктор у него может быть только один. Конструкторы и деструктор класса являются обычными членами-функциями этого класса, за исключением перечисленных ниже особенностей.

- Они не возвращают никаких значений (даже значения типа `void`).
- Конструктор и деструктор имеют то же имя, что и класс, но перед именем деструктора указывается символ ~.

```
class A {
Public:
A();  // конструктор
~A(); // деструктор
}
```

- Конструктор может принимать параметры, а деструктор — нет.
- Их нельзя вызывать как обычные функции.

Доступ к членам-данным и членам-функциям

Рассмотрим приведенный ниже пример простого класса.

```
class Rabbit {
Private
int speed=20;
} ;
```

Позже его можно объявить в программе с помощью одного из следующих двух методов:

```
Rabbit Rabbit1;
Rabbit* Rabbit2 = new Rabbit;
```

Переменная `Rabbit1` создается в стеке, а переменная `Rabbit2` — в куче. Быстрый доступ к членам-данным обоих переменных можно получить с помощью показанного ниже оператора прямого доступа:

```
Rabbit1.speed=30;
(*Rabbit2).speed=30;
```

Для упрощения доступа к членам-данным такого класса, как `Rabbit2`, в языке C++ предусмотрен оператор (`->`). Предыдущее выражение с помощью этого оператора можно переписать следующим образом:

```
Rabbit2->speed=30;
```

Указатель `this`

Так же, как указатель `self` в Delphi, в C++Builder указатель `this` является скрытым членом всех классов. Ниже приводится пример использования указателя `this`.

```
__fastcall TForm1::TForm1(TComponent* Owner) : TForm(Owner)
{
TLabel *Label1 = new TLabel(this);
Label1->Parent = this;
Label1->Caption = this->Width;  // это эквивалентно Form1->Width;
}
```

Директивы препроцессора

Директивами препроцессора называются особые инструкции для компилятора. Они указываются в строках, которые начинаются с символа #. Директивы препроцессора обычно располагаются в начале модуля. Некоторые директивы могут размещаться в любом месте модуля, другие — только в определенных местах. Например, директива `#pragma argsused`, должна располагаться до определения функции, для которой она применяется. Рассмотрим подробнее директивы препроцессора `#define` и `#include`.

Директива `#define`

Как было сказано выше, команда препроцессора `#define` может использоваться для объявления констант.

Команда `#define` также используется для создания макросов. Макросы похожи на функции и могут принимать параметры. Ниже показан пример двух макросов.

```
#define MAX(A,B) ((A)>(B)?(A): (B)
#define MIN(A,B) ((A)>(B)?(B): (A)
```

В программе их можно использовать следующим образом.

```
int x, y, max ;
x = 5;
y = 6;
max = MAX(x,y); // Напомним, что в C++Builder следует учитывать регистр букв.
```

Директива `#include`

До обсуждения директивы `#include` необходимо рассмотреть понятие заголовочного файла. В языке C++ заголовочный файл применяется как интерфейс для внешних файлов с исходным кодом.

В большинстве случаев все объявления констант, переменных и функций следует размещать в заголовочных файлах и только описания следует помещать во внешних файлах исходного кода с расширением `.CPP`.

Во время компиляции директивы `#include` будут заменены тем заголовочным файлом, который указан этой директивой. Синтаксис этой директивы может иметь одну из таких форм:

```
#include <заголовочный_файл.hpp>
#include "заголовочный_файл.hpp"
```

Первое выражение сообщает компилятору о том, что нужно выполнить поиск заголовочного файла среди включаемых файлов, указанных во вкладке Directories/Conditionals диалогового окна Project Options, которое можно вызвать с помощью команды меню Project⇨Options. Второе выражение сообщает компилятору о том, что поиск заголовочного файла сначала нужно выполнить в текущем каталоге, а затем продолжить поиск среди указанных включаемых файлов.

Типы файлов

В табл. 1.6 приведена сравнительная характеристика типов файлов, которые используются в Delphi и в C++Builder.

Таблица 1.6. Сравнение типов файлов, используемых в Delphi и C++Builder

Delphi	C++Builder	Описание
DPR	BPR	Файл проекта Builder
PAS	CPP	Каждый модуль имеет один файл с кодом плюс один файл проекта
DFM	DFM	Двоичный файл, в котором описывается форма и все ее компоненты
PAS	H or HPP	Заголовочный файл
RES	RES	Компилированный двоичный файл ресурсов
DCU	OBJ	Скомпилированный двоичный объектный файл
BPG	BPG	Комбинация нескольких проектов в одном групповом файле
DPK	BPK	Файл с перечислением модулей, используемых в данном пакете
Нет	BPI	Файл импорта библиотеки, созданный для каждого файла пакета
BPL	BPL	Файл библиотеки с особыми компонентами Builder (подобный DLL-модулям в Windows), подключаемый при выполнении программы
Нет	LIB	Статический файл библиотеки
~*	~*	Резервный файл
Нет	MAK	Текстовый файл с информацией о том, какие файлы Builder должен скомпилировать и скомпоновать для этого проекта
MAP	MAP	Текстовый файл с информацией о том, какие отладочные задачи низкого уровня следует выполнить
Нет	TDS	Этот файл содержит отладочную информацию (Turbo Debugger Symbol — TDS)

Для пояснения приведенных выше сведений их следует прокомментировать.

- Первые пять типов файлов создаются после первого сохранения проекта.
- Файлы формата CPP и HPP эквивалентны разделам реализации и интерфейса файла формата PAS, соответственно.
- Файлы формата OBJ создаются сразу после выбора команды меню Project⇨Compile Unit, а также при компиляции пакета.
- Файл формата BPK создается при сохранении пакета.
- Файлы формата BPI, BPL и LIB создаются при компиляции или инсталляции пакета.
- В C++Builder 4 и C++Builder 5 файл формата MAK заменен файлом формата BPR. При попытке открытия файла формата MAK в C++Builder версии 4 или выше, интегрированная среда разработки автоматически преобразует его в файл формата BPR.
- Файл формата TDS содержит отладочную информацию. Его размер увеличится при включении в него отладочной информации об этапе компиляции. Чтобы включить в него отладочную информацию о компиляции, следует установить флажок параметра Debug Information во вкладке Compiler диалогового окна Project Options, которое можно вызвать с помощью команды меню Project⇨Options.

Преимущества и недостатки C++Builder версии 5

В этом разделе мы рассмотрим некоторые особенности C++Builder, которые могут существенно влиять на процесс создания программного обеспечения. Было бы справедливо предположить, что большинство читателей этой книги уже выбрали C++Builder в качестве предпочтительной платформы для создания программного обеспечения. В таком случае мы попытаемся вкратце описать некоторые компоненты C++Builder, которые являются его несомненными достоинствами, а также обратить внимание читателя на другие компоненты, работа с которыми может вызвать трудности.

Визуальная реальность: Rapid Application Development — действительно быстрое создание приложений

Интегрированная среда разработки или IDE-среда C++Builder с дополнительной библиотекой визуальных компонентов, или VCL-библиотекой (Visual Component Library), является единственным исчерпывающим, интегрированным решением для программистов на языке C++, которым необходима действительно визуальная среда для создания приложений с богатым и профессионально оформленным интерфейсом пользователя. Набор шаблонов проекта C++Builder позволяет создать и запустить новую программу в течение одной минуты. Более быстрого способа программирования на языке C++ для Windows просто нельзя найти. Репозитарий объектов (Object Repository) предоставляет неоценимую возможность повторно использовать созданный ранее код, благодаря чему повышается производительность труда разработчика и согласованность созданного программного обеспечения. Даже пользователь с небольшим опытом работы сможет создать простой многооконный текстовый редактор с командами меню, панелями инструментов (которые можно перемещать по экрану или закреплять на сторонах основного окна приложения) и другими основными компонентами в течение получаса. Причем, этот способ работы не только очень эффективен, но и достаточно удобен. Возможность просмотра внешнего вида приложения еще во время создания позволяет сократить затраты на тестирование и исключить возможные ошибки.

В дополнение к уже имевшимся в прежних версиях компонентам в C++Builder 5 добавлены новые компоненты, которые существенно упрощают процесс визуального создания приложений. Например, компонент `TFrame` позволяет работать с независимыми и повторно используемыми окнами в виде форм, которые можно помещать внутри форм или даже на панели Component Palette для последующего использования. Среди усовершенствований окна Object Inspector следует отметить улучшенные редакторы свойств `TColor` и `TCursor`, а также списки свойств и событий, которые разбиты по категориям для удаления с переднего плана наиболее редко используемых наборов компонентов. Это значительно упрощает поиск нужных компонентов.

В левой части рис. 1.5 в окне Object Inspector показан редактор свойства `TCursor` при работе с проектом, содержащим два фрэйма, которые используются внутри основной формы.

Хотя пакет Visual C++ фирмы Microsoft содержит в своем названии слово “визуальный” (“visual”), в нем нет подобных средств визуального программирования на основе WYSIWYG-методологии. Это, вероятно, наиболее значительное преимущество C++Builder по сравнению с Visual C++.

У пользователей Visual C++ есть два альтернативных варианта создания приложения для Windows. Один из них — использование имеющегося редактора ресурсов и библиотеки классов Microsoft Foundation Classes (MFC). Второй — создание основного кода в виде динамически подключаемых библиотек (dynamic link library — DLL) вместе с графическим интерфейсом пользователя (graphical user interface — GUI), созданным в другой среде, которая в большей степени подходит для визуального программирования, например, на основе собственного языка Visual Basic фирмы Microsoft.

В первом случае вам не удастся добиться достаточно высокой степени визуального программирования. Несмотря на то что редактор ресурсов в Visual C++ позволяет создавать диалоговые окна почти так же, как и с помощью конструктора форм в C++Builder, все же в C++Builder предусмотрено гораздо больше разнообразных способов работы с элементами управления при создании приложения. Более того, в C++Builder предусмотрена поддержка MFC, которая в основном предназначена для тех редких случаев, когда необходимо обеспечить совместимость с уже имеющимся кодом на основе MFC. (В Visual C++ не предусмотрена поддержка кода, созданного с помощью C++Builder.)

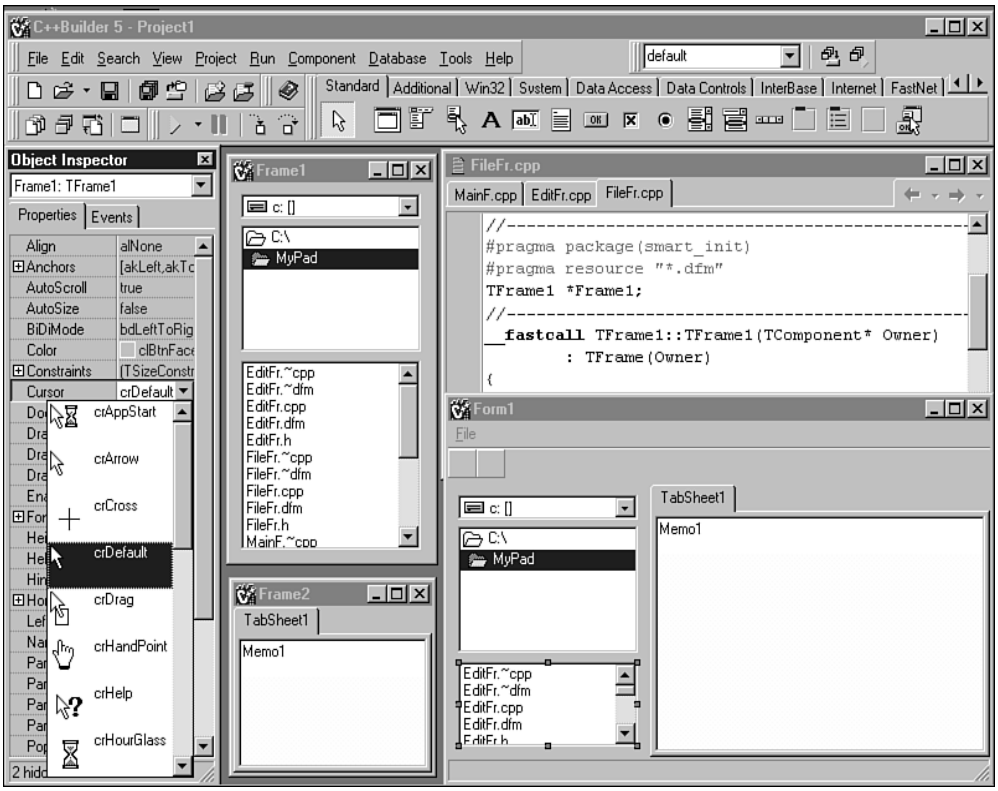


Рис. 1.5. Некоторые усовершенствования визуальной среды создания приложений в C++Builder 5

Более того, редактор ресурсов в Visual C++ содержит только стандартные элементы управления, тогда как в C++Builder можно использовать тысячи дополнительных VCL-компонентов. MFC является очень "тонкой" оболочкой вокруг интерфейса прикладного программирования Windows или API-интерфейса (Application Programming Interface — API). Хотя он в большей мере соответствует API-интерфейсу, чем VCL, но не обладает столь же высокой встроенной функциональностью, поэтому значительная часть работы возлагается на программиста.

Во втором случае имеются некоторые преимущества с точки зрения архитектуры приложения, т.к. "основной" код приложения отделен от пользовательского интерфейса, что позволяет добиться гибкости и расширяемости. Тем не менее способность C++Builder служить в качестве компилятора C++ и возможность визуальной разработки GUI-интерфейса еще раз подтверждают мнение о большей эффективности C++Builder. В хорошо спроектированном и созданном с помощью C++Builder приложении можно с тем же успехом выполнить отделение GUI-интерфейса от основного кода.

Поддержка стандарта языка C++

В течение многих последних лет фирма Borland на несколько шагов опережала конкурентов благодаря соответствию ее компилятора последнему стандарту ANSI/ISO. В этом смысле не является исключением и C++Builder версии 5, созданный на основе 32-разрядного компилятора C++ (BCC32) версии 5.5. Подробные сведения об этом языке формально изложены в последней спецификации стандарта ANSI/ISO. В некоторых отношениях он отличается от предыдущей стандартной версии, это особенно касается шаблонов и пространства присваиваемых имен в C++.

Для небольших и монолитных приложений, разработанных на основе некого прототипа и с помощью конструктора форм это может быть несущественно. Однако при создании крупных и масштабируемых приложений строгое соответствие стандартам компилятора BCC32 очень существенно для долговременного успеха всего проекта в целом.

Однако в результате политики фирмы Borland в отношении постепенных улучшений основного компилятора некоторые разработчики программного обеспечения могут обнаружить, что существующие программы, корректно компилируемые в прежних версиях C++Builder, не компилируются в версии 5. Эта проблема возникает только в тех случаях, когда в программе используется компонент, который претерпел значительное изменение в последующей версии, например шаблон.

Автору на собственном опыте довелось испытать трудности длительного преобразования такого приложения, сначала созданного с помощью C++Builder 3, для организации его совместимости с версией 4, а затем — с версией 5. Это отдельно взятое приложение было не совсем удачно запрограммировано, что следует иметь ввиду разработчику, и в результате затормозило всю работу над этим проектом. Как опытный программист, стремящийся постоянно повышать свою квалификацию, автор считает этот опыт достаточно полезным для дальнейшей деятельности. К сожалению, такие ситуации вряд ли могут привести в восторг руководство проекта и клиента. Мораль этой истории такова, что долговременные преимущества компилятора чрезвычайно желательны, но в каждом отдельном случае следует предвидеть возможные последствия его применения.

В C++Builder также расширены возможности компилятора по поддержке собственных (т.е. непереносимых) расширений языка C++ для поддержки VCL-библиотеки. Хотя во многих случаях для переносимости их можно избежать, компилятор имеет еще одну особенность. Например, ключевое слово `__closure` может применяться для работы с

указателем на член-функцию экземпляра, а не класса, потому что в рамках текущей спецификации языка C++ не существует способа достижения подобного эффекта. Более подробную информацию по этому поводу можно найти по ссылке “keyword extensions” (расширения ключевых слов) на странице о ключевом слове `__closure` в оперативной справке.

Выбор среды разработки

Несмотря на заявления создателей компиляторов, идеальной среды разработки программного обеспечения не существует. Всегда найдутся сторонники одного программного продукта, которым не нравятся все остальные. Так как программное обеспечение эволюционирует, то неизбежно будут возникать проблемы, связанные с появлением ошибок в используемом инструменте создания программного обеспечения. Такова суровая правда жизни разработчика программного обеспечения.

C++Builder 5 является инструментом для создания программного обеспечения, а не панацеей для решения всех проблем программирования. Конечно, это замечание относится не только к C++Builder. К тому же для одних областей создания программного обеспечения он идеально подходит и превосходит все остальные инструменты, а для других — совсем не подходит.

При этом нет никаких сомнений, что C++Builder 5 является надежной и интуитивно понятной средой создания программного обеспечения, которая может стать чрезвычайно мощным и эффективным инструментом. У пользователей, идущих по предложенному Microsoft пути, либо есть другие достаточно убедительные доводы для этого, либо одна единственная причина: просто Borland — это не Microsoft.

Во время создания этой книги предполагалось, что Borland создаст версию C++Builder для операционной системы Linux. Существование версий программного продукта для многих платформ, несомненно, вызывает интерес к нему и склоняет пользователей в пользу его выбора. Более подробно об этом рассказывается в разделе о Kylix этой главы.

Преимущества и недостатки C++Builder

После такого обсуждения можно сделать некоторые полезные выводы. Хотя приведенный ниже список не является исчерпывающим, все же можно выделить явные преимущества и недостатки C++Builder. При этом не принимаются во внимание средства создания приложений для работы с базами данных в Internet, наличие которых также является неоспоримым достоинством C++Builder, но они будут приведены ниже.

Итак, преимущества C++Builder 5 таковы.

- C++Builder — мощный инструмент создания форм.
- Он подходит для традиционного программирования на языке C++ и для быстрого создания интерфейса пользователя, потому что в нем интегрирован ANSI/ISO–совместимый компилятор с истинно визуальной средой разработки.
- Его исчерпывающе полная библиотека визуальных компонентов Visual Component Library может быть дополнена компонентами сторонних разработчиков (включая компоненты Delphi) и компонентами собственной разработки.
- Собственные расширения стандарта языка C++ поддерживают дополнительные программные конструкции.

- Благодаря шаблонам проекта и репозитарию объектов Object Repository можно быстро создавать новые проекты и повторно использовать созданный ранее код.
- Простая поддержка баз данных осуществляется с помощью специальных компонентов для работы с данными и Borland Database Engine.
- Он имеет мощные средства для создания Internet-приложений.
- Преобразование кода приложений Visual C++ в код C++Builder выполняется очень просто.
- В нем предусмотрен способ возможного переноса приложений в операционную среду Linux.

Ниже приводится перечень недостатков C++Builder 5.

- Простота использования конструктора форм может привести к созданию программы с беспорядочной структурой, если заблаговременно не позаботится о ее структуре.
- Не стоит модернизировать среду разработки, если для компиляции проекта в C++Builder 5 в код программы придется внести значительные изменения.
- Преобразование кода приложений C++Builder в код Visual C++ обычно выполняется очень сложно.
- Выбор программных продуктов фирмы Borland, а не фирмы Microsoft, может иметь определенные последствия вследствие доминирующей роли фирмы Microsoft на рынке.

Подготовка к работе с Kylix

Kylix — это внутреннее условное название среды интегрированной разработки программного обеспечения фирмы Borland для операционной системы Linux. (Официальное название во время создания этой книги еще было неизвестно.) Другими словами, это Delphi и C++Builder для Linux. С помощью Kylix можно создавать приложения для операционной системы Linux так же легко, как и приложения для операционной системы Windows с помощью C++Builder. При этом появится возможность переносить многие существующие приложения C++Builder для работы в операционной системе Linux.

У Kylix и C++Builder (и Delphi) имеются как сходства, так и различия (последние отражают различия между операционными системами Windows и Linux). Некоторые из них будут рассмотрены, чтобы представить вниманию читателя новые возможности C++Builder. При необходимости переноса приложений в операционную среду Linux следует учитывать эти сходства и различия между Kylix и C++Builder.

У бой-скаутов есть очень простой девиз “Будь готов!” Автор считает, что это отличный девиз, который следует иметь в виду в процессе создания программного обеспечения.

Обратите внимание на то, что следующая часть этого раздела основана на информации, полученной из открытых публикаций в печати и обсуждений Kylix, которые были доступны во время создания этой книги. Эта информация должна рассматриваться как предварительная, поскольку продукт еще не готов и его компоненты и возможности не представлены для всеобщего обсуждения.

Сходства между Kylix и C++Builder

Оболочка Kylix во многом похожа на C++Builder. За исключением общих отличий между пользовательским интерфейсом в операционной среде Windows и разнообразными пользовательскими интерфейсами в операционной среде Linux, IDE-среда в обоих продуктах практически идентична. Она включает Object Inspector, обозреватель классов, разные вкладки компонентов в панели Component Palette, а также редактор кода. При этом разработчик сможет использовать все свои знания, полученные при работе с C++Builder. На рис. 1.6 показан внешний вид проекта Delphi, загруженного в предварительную версию Kylix с использованием графического интерфейса пользователя K Desktop Environment (KDE) операционной системы Linux. Обратите внимание на сходство с интегрированной средой разработки C++Builder.

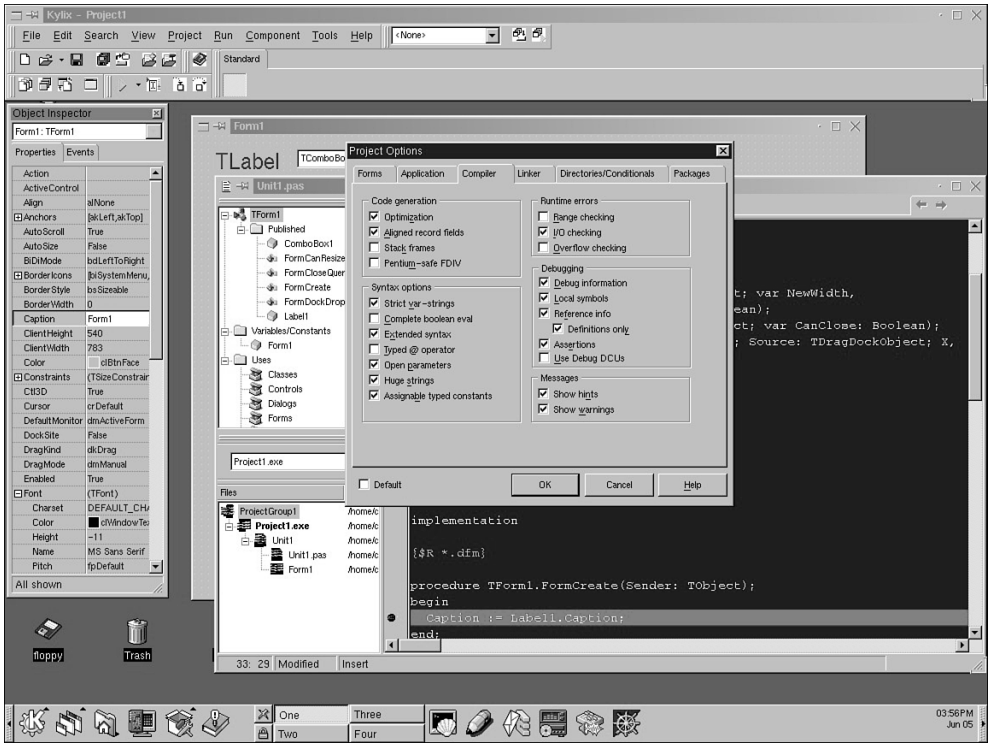


Рис. 1.6. Интегрированная сре-да разработки Kylix в опе-рационной системе Linux на основе графического интерфейса пользователя K Desktop Environment

В C++Builder используется много общих функциональных возможностей и компонентов интерфейса пользователя Delphi. По мнению автора, присоединение к ним среды Kylix приведет к тому, что многие компоненты могут быть в равной степени использованы во всех трех продуктах. Новые компоненты, появившиеся в каком-нибудь из этих программных продуктов, будут доступны в следующем выпуске других продуктов этого семейства.

В Kylix можно так же, как и в C++Builder, программировать на языке C++. При этом вам вряд ли удастся заметить какие-либо различия, поскольку C++ — это в любом случае C++.

Различия между Kylix и C++Builder

Поскольку операционная система Linux отличается от Windows, то некоторые компоненты C++Builder будут отсутствовать в Kylix, по крайней мере в первое время. Связанные с Windows технологии включают COM, DCOM, COM+, ActiveX и Win32 API-функции. Несколько лет назад Microsoft, Digital и другие фирмы приступили к созданию DCOM-технологии для UNIX-серверов. Эта работа еще не закончена и по сей день, хотя, возможно, это только вопрос времени. Предполагается, что на определенном этапе в них будет включена поддержка технологии SOAP (Simple Object Access Protocol).

По мнению автора, многие Win32 API-функции будут иметь эквивалентные функции в Linux, прозрачно используя системный API-уровень или особый интерфейс Linux. В последнем случае перенос приложений между двумя системами существенно усложнится.

VCL-библиотека для Linux получила название Borland Component Library for Cross Platform или CLX. Аббревиатура CLX произносится, как английское слово “clicks”. Различные компоненты CLX сгруппированы в такие логические модули, как визуальные компоненты, компоненты доступа к данным и т.д. GUI-компоненты будут встроены в оболочку Qt фирмы TrollTech. Qt является кросс-платформенной оболочкой для создания приложений на языке C++ с GUI-интерфейсом, которая была выбрана фирмой Borland в качестве наиболее подходящей для использования в операционной системе Linux. Оболочка Qt инсталлируется как вместе с KDE, так и отдельно, если используется Gnome или какой-либо другой графический интерфейс пользователя Linux. Библиотека CLX появится в версиях C++Builder и Delphi для Windows для упрощения переноса приложений между Linux и Windows. Однако для повышения производительности работы приложения для Windows при его создании следует использовать VCL-библиотеку.

Kylix будет иметь новый тонкий уровень доступа к базам данных, который, вероятно, будет называться DBExpress и использоваться как аналог механизма доступа к данным Borland Database Engine (BDE). Во время создания этой книги предполагалось, что он обеспечит поддержку ограниченного набора основных систем баз данных, для чего потребуется только один DLL-модуль, который нужно будет инсталлировать вместе с клиентом.

Перенос проектов C++Builder в Kylix

Перенос проектов C++Builder в Kylix будет зависеть от компонентов, которые будут реализованы в Kylix. В предыдущих разделах уже обсуждались вопросы, связанные с переносом приложений.

Перенос большинства приложений не вызовет никаких трудностей. Было бы замечательно, если бы при этом удалось избежать модификаций основного кода, но на самом деле для многих приложений потребуется незначительная подгонка некоторых параметров.

Так когда же он будет выпущен?

На основании существующих мнений и прогнозов можно предположить, что Delphi-версия среды разработки программного обеспечения Kylix будет выпущена в конце 2000 года. Поэтому к моменту выпуска этой книги она уже может выйти в свет, и читатель будет знать о ней гораздо больше, чем автор этой книги в момент ее создания!

А когда будет выпущена C++Builder-версия среды разработки программного обеспечения Kylix? В настоящее время предполагается, что датой выпуска будет начало 2001 года.

Резюме

В этой главе излагаются основы C++Builder для начинающих программистов и программистов, имеющих опыт работы с Delphi. После такого предварительного знакомства с C++Builder читатель может представить себе огромные возможности этого программного продукта для создания приложений для Windows. Здесь были представлены также новые компоненты и усовершенствования C++Builder версии 5, а также описаны особенности модернизации и совместимости при переносе ранее созданных проектов и инструментов сторонних разработчиков. Наконец, в ней была представлена среда разработки Kylix, предлагаемая фирмой Borland для создания приложений для платформы Linux.

Представленные в этой главе концепции более подробно будут описаны в следующих главах.

Глава

2

Проекты C++Builder и дополнительные сведения об IDE-среде

Джэйми Оллсоп
Йото Йотов
Джарод Холингвэрт
Халид Алманай
Дэн Баттерфилд

В главе 1 уже были представлены основные сведения об IDE-среде и проектах, которые являются составными частями C++Builder. Материал этой главы основан на сведениях, которые уже приводились в главе 1. Прежде всего следует внимательно рассмотреть проекты, включаемые в проект файлы, а также окно менеджера проектов Project Manager. После этого мы рассмотрим методы работы с окном репозитария объектов Object Repository для хранения и повторного использования кода и других созданных вами программных компонентов.

Мы также рассмотрим пакеты, особый тип динамически подключаемой библиотеки (dynamic link library — DLL), а также предоставляемые ими возможности для IDE-среды и приложений. Наконец, будут рассмотрены новые компоненты IDE-среды C++Builder 5.

Проекты C++Builder

C++Builder позволяет существенно упростить работу программистов. Представленная в главе 1 интегрированная среда разработки, или IDE-среда (Integrated Development Environment — IDE), позволяет создавать приложения на основе двух разных подходов — с помощью графического конструктора форм Form Designer и редактора кода.

Благодаря использованию ***ClassExplorer***, ***Code Insight*** и стандартных компонентов конструктора форм, C++Builder облегчает труд программиста, автоматически создавая огромное количество рутинного кода!

В C++Builder предусмотрено создание разных типов приложений.

- Windows-приложения или консольные приложения.
- Приложения с архитектурой клиент/сервер.
- Динамически подключаемые библиотеки (dynamic link libraries — DLL).
- Настраиваемые компоненты и пакеты.
- Объекты компонентной объектной модели (Component Object Model — COM) и элементы управления ActiveX.

Все эти приложения создаются с помощью проектов. Проектом называется набор файлов с исходным кодом на языке C++, файлов форм и других типов файлов, которые определяют данное приложение. Для сохранения структуры и различных параметров проекта, определяющих способ создания приложения, в C++Builder используется специальный файл проекта.

Файлы, входящие в состав проекта C++Builder

Проекты в C++Builder состоят из файлов разных типов. Некоторые из них создаются автоматически при создании нового проекта или добавлении новых компонентов в уже существующий проект. Одни файлы создаются самим программистом, другие — при компиляции приложения. Все эти файлы можно разбить на следующие условные группы:

- основные файлы проекта;
- файлы форм;
- файлы пакета;
- файлы параметров рабочего стола;
- резервные файлы.

Основные файлы проекта

При создании большинства проектов автоматически создаются три основных типа файлов, которые перечислены ниже.

- Файл проекта C++Builder *имя_проекта*.bpr.
- Основной файл с исходным кодом проекта *имя_проекта*.cpp.
- Основной файл ресурсов проекта *имя_проекта*.res.

Эти файлы создаются в оперативной памяти компьютера только для того, чтобы можно было начать работу. Они не будут сохранены на жестком диске до тех пор, пока программист не сохранит весь проект целиком.

По умолчанию проект получает имя Project1. Это имя можно изменить при первой попытке сохранения проекта, переименовав файл Project1.bpr в диалоговом окне Save Project As. При создании других файлов проекта следует избегать совпадения их имен с именами основных файлов проекта и других файлов модулей, так как это может привести к путанице и в некоторых случаях не позволит просматривать содержимое этих файлов с помощью редактора кода.

Файл проекта C++Builder

Файлом проекта называется текстовый файл с параметрами проекта и правилами его создания. В разных версиях C++Builder формат этого файла немного отличается. В версиях 1–4 C++Builder этот файл имел формат файла сборки (Makefile). В C++Builder версии 1 файл проекта имел расширение .mak, чтобы подчеркнуть наличие такого формата. В C++Builder версии 5 этот файл оформлен в формате XML (Extensible Markup Language). Более подробная информация о файле формата XML приводится далее в этой главе.

Основной файл с исходным кодом проекта

Этот файл содержит код входа (запуска) приложения, макросы типа USEUNIT и USEFORM, в которых определены встроенные в приложение файлы, и многое другое. В C++Builder предусмотрена автоматическая поддержка макросов в процессе создания приложения.

В стандартном приложении Windows основной файл с исходным кодом проекта содержит функцию WinMain() в качестве точки входа приложения. В других типах приложений эта функция может называться DllEntryPoint() или просто main(). В отличие от большинства других автоматически генерируемых файлов с исходным кодом, этот файл не имеет соответствующего заголовочного файла (с расширением .h). Обычно необходимость изменения основного файла проекта возникает редко, за исключением тех случаев, когда нужно выполнить некоторую функцию в момент запуска приложения, например отобразить на экране заставку во время инициализации приложения.

Основной файл ресурсов проекта

Этот файл содержит пиктограмму приложения, номер версии приложения и другую информацию. Не все типы приложений имеют файл ресурсов проекта.

В файлах ресурсов хранятся также рисунки, пиктограммы и изображения указателя мыши проекта. Для создания этих элементов обычно используется редактор изображений Image Editor, который входит в состав C++Builder. Файлы ресурсов могут также содержать строки и другие объекты. Более подробная информация о способах хранения изображений в файле ресурсов приводится в главе 10.

Файлы формы

При создании формы в C++Builder автоматически создаются следующие типы файлов.

- Файл структуры формы *имя_модуля*.dfm.
- Файл кода формы *имя_модуля*.cpp.
- Заголовочный файл формы *имя_модуля*.h.

По умолчанию модуль формы получает имя Unit1 при создании первой формы, Unit2 — при создании второй формы и т.д. Файлы форм можно переименовать при сохранении всего проекта. Расширение .dfm означает Delphi Form, т.е. служит напоминанием о том, что C++Builder частично происходит от программного продукта Delphi фирмы Borland. Файл .dfm содержит значения, которые представляют графическую часть формы, например расположение компонентов, стили шрифтов и т.д.

В версиях 1–4 C++Builder файл .dfm хранится в двоичном формате. В этих версиях его можно просматривать в текстовом формате, щелкая правой кнопкой мыши на форме и выбирая из контекстного меню команду View As Text. В C++Builder версии 5 файл .dfm по умолчанию хранится в текстовом формате, но в случае необходимости его можно сохранить и в двоичном формате. Более подробно об этом будет говориться позже в этой главе. Файл структуры формы .dfm можно редактировать, но такая необходимость возникает редко.

Файл .cpp и соответствующий ему заголовочный файл .h создаются вместе с файлом .dfm каждый раз при создании новой формы. Файл .h содержит определение класса C++ этой формы. Файл .cpp содержит функции — обработчики событий формы и созданных в ней компонентов. В простых приложениях большая часть созданного кода размещается в файле формы .cpp.

Для просмотра содержимого файлов .cpp и .h следует выполнить перечисленные ниже действия.

1. Если проект еще не открыт, выберите команду меню File⇨Open.
2. Выберите команду View⇨Units, затем — нужный файл модуля формы и щелкните на кнопке OK или нажмите клавишу <Enter>. После этого файл .cpp формы будет отображен в окне редактора кода.
3. Для просмотра содержимого заголовочного файла щелкните правой кнопкой мыши на файле формы .cpp в окне редактора кода и выберите из контекстного меню команду Open Source/Header File.

Файлы пакета

Пакетом называется динамически подключаемая библиотека, или DLL-библиотека (dynamic link library). Она может совместно использоваться большим количеством приложений C++Builder, что дает возможность нескольким приложениям совместно использовать классы, компоненты и данные. Например, наиболее часто используемые компоненты C++Builder располагаются в пакете VCL50. Большинство приложений C++Builder использует код этого пакета, который находится в файле пакета vcl50.bpl.

Ниже перечислены некоторые типы файлов пакета.

- Файл пакета с параметрами проекта (bpk).

  Этот файл аналогичен файлу проекта .bpr, но применим только для пакетных файлов. Его можно создать, выбрав команду меню File⇨New и в диалоговом окне New Items — пиктограмму Package.

- Файл пакета библиотеки Borland (bpl).

  Это рабочая библиотека для файла пакета, аналогичная DLL-библиотеке, за исключением того, что содержит некоторые специфические для C++Builder элементы. Ее основное имя совпадает с основным именем кода пакетного файла.

- Файл пакета библиотеки импорта Borland (bpi).

  При каждой компиляции файла с кодом пакета создается файл .bpi. Его имя также совпадает с основным именем кода пакетного файла.

Более подробные сведения о пакетах можно найти ниже в этой главе.

Файлы параметров рабочего стола

В файле параметров рабочего стола проекта хранятся сведения о расположении открытых окон в IDE-среде и открытых файлов в окне редактора кода. Эти параметры сохраняются до следующего открытия проекта.

Файл параметров рабочего стола имеет формат *имя_проекта*.dsk и хранится в той же папке, где хранится сам проект. В C++Builder предусмотрено автоматическое сохранение этого файла при закрытии проекта. Для указания этой опции следует выбрать команду меню Tools⇨Environment Options, а затем в отобразившемся диалоговом окне Environment Options выбрать вкладку Preferences и установить флажок параметра Project Desktop (Параметры рабочего стола) в группе AutoSave Options (параметры автоматического сохранения).

Резервные файлы

При каждом сохранении проекта, за исключением первого, в C++Builder предусмотрено создание резервного файла для всех файлов формата .bpr, .dfm, .cpp и .h. В расширениях всех резервных файлов в качестве префикса используется символ ~, например расширение .bpr в резервном файле будет иметь вид .~bp, расширение .cpp — .~cp и т.д.

Менеджер проектов Project Manager

В окне менеджера проектов Project Manager отображается структура файлов проекта или группы проектов. Ее содержимое можно просмотреть, выбрав команду меню View⇨Project Manager.

На рис. 2.1 показан пример диалогового окна Project Manager.

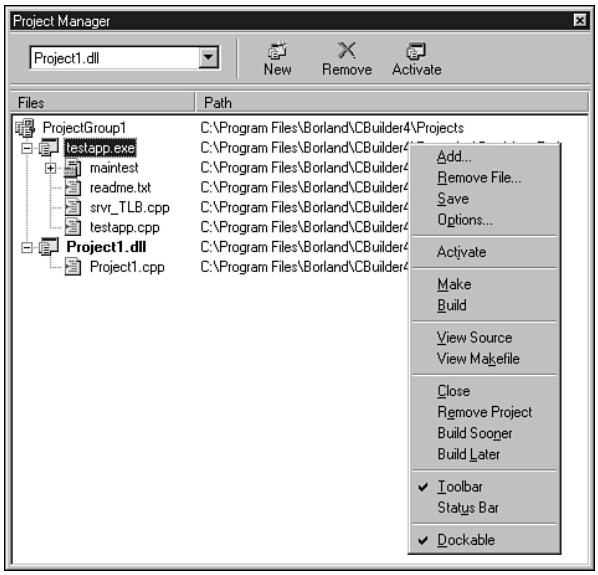


Рис. 2.1. Диалоговое окно Project Manager

Группой проектов называется совокупность проектов, необходимых для создания приложения. Например, приложение может содержать VCL-проект, DLL-проект и консольный проект. Информация о группе проектов хранится в той же папке, где хранится сам проект, т.е. в файле *имя_проекта*.bpg.

Ниже перечислены основные функции менеджера проектов.

- Создание и удаление проектов.
- Создание и удаление форм и модулей.
- Возможность выбора и перетаскивания компонентов.
- Выбор проекта.
- Компиляция или компоновка всех проектов.

Репозитарий объектов

Одним из наиболее мощных инструментов управления проектом в C++Builder является репозитарий объектов (Object Repository). Он позволяет программисту совместно или повторно использовать формы, проекты, модули данных, шаблоны проектов и программы-мастера. Например, при необходимости создания ряда приложений со стандартным диалоговым окном About можно сохранить это диалоговое окно в репозитарии объектов. Это позволит быстро добавлять его в другие проекты, экономя время и избавляя вас от необходимости каждый раз копировать или создавать его заново.

Включение компонентов в репозитарий объектов

В качестве первого примера рассмотрим уже упоминавшееся диалоговое окно About. В качестве его заготовки используем стандартное окно About, предлагаемое в C++Builder. Для его создания и тестирования нужно сначала создать простое приложение, в котором оно используется, выбрав команду меню File⇨New Application. Для добавления в это приложение предлагаемого по умолчанию окна About выберите команду меню File⇨New и во вкладке Forms диалогового окна New Items выберите пиктограмму About box.

Теперь нужно подправить внешний вид этого окна, изменяя надписи, которые будут видны во время работы с готовым приложением. На рис. 2.2 показан один из вариантов внешнего вида нашего окна About. Оно содержит логотип, название и номер версии программного продукта, а также комментарий. Сохраним в нем предлагаемый по умолчанию логотип, хотя при желании его также можно изменить.

Рис. 2.2. Диалоговое окно About после редактирования надписей

При правильном выполнении описанных выше действий по созданию нового окна About можно заметить, что, помимо редактирования надписей, была также удалена надпись об авторских правах. Это было сделано просто для демонстрации возможности внесения любых изменений. Чтобы удалить надпись об авторских правах, нужно выделить надпись `Copyright Label` в окне конструктора форм и нажать клавишу <Delete>.

При правильном выполнении описанных выше действий по созданию нового окна About можно заметить, что, помимо редактирования надписей, была также удалена надпись об авторских правах. Это было сделано просто для демонстрации возможности внесения любых изменений. Чтобы удалить надпись об авторских правах, нужно выделить надпись `Copyright Label` в окне конструктора форм и нажать клавишу <Delete>.

Присвойте окну About имя StdAboutBox, выбрав компонент About Box в окне конструктора форм и введя текст StdAboutBox в поле свойства Name окна Object Inspector. Сохраните проект C++Builder, выбрав команду меню File⇨Save All. Затем в диалоговом окне Save As замените имя Unit2.cpp модуля компонента About Box именем **StdAboutBox.cpp**. Точно так же замените имя Unit1.cpp модуля основной формы приложения именем **AboutTestForm.cpp**, а потом укажите **AboutTestProj.cpp** в качестве имени файла кода проекта.

Для изменения в окне About надписей, которые будут видны во время выполнения программы, следует добавить несколько строк в код обработчика события OnCreate окна About. Для этого выделите окно About в окне конструктора форм, щелкните на вкладке Events окна Object Inspector и дважды щелкните в поле события OnCreate. Затем добавьте показанные в листинге 2.1 строки в код обработчика события TStdAboutBox::FormCreate(), который появится в окне редактора кода.

Листинг 2.1. Обработчик события OnCreate для окна TStdAboutBox

```
void __fastcall TStdAboutBox::FormCreate(TObject *Sender)
{
    struct TransArray
    {
       WORD LanguageID,CharacterSet;
    };
    DWORD VerInfo,VerSize;
    HANDLE MemHandle;
    LPVOID MemPtr,BufferPtr;
    UINT BufferLength;
    TransArray *Array;
    char QueryBlock [40 ];

    //Получить название программы и номер версии.
    String Path(Application->ExeName);
    VerSize = GetFileVersionInfoSize(Path.c_str(), &VerInfo);
    if (VerSize > 0){
        MemHandle = GlobalAlloc(GMEM_MOVEABLE, VerSize);
        MemPtr = GlobalLock(MemHandle);
        GetFileVersionInfo(Path.c_str(),
                           VerInfo, VerSize, MemPtr);
        VerQueryValue(MemPtr, "\\VarFileInfo\\Translation",
                      &BufferPtr, &BufferLength);
        Array = (TransArray *)BufferPtr;
        //Получить название программы.
        wsprintf(QueryBlock,
                 "\\StringFileInfo\\%04x%04x\\ProductName",
                 Array[0].LanguageID, Array[0].CharacterSet);
        VerQueryValue(MemPtr, QueryBlock, &BufferPtr,
                      &BufferLength);
        //Указать название программы.
        ProductName->Caption =(char *)BufferPtr;
        //Получить номер версии.
```

```
        wsprintf(QueryBlock,
                 "\\StringFileInfo\\%04x%04x\\ProductVersion",
                 Array[0].LanguageID, Array[0].CharacterSet);
        VerQueryValue(MemPtr, QueryBlock, &BufferPtr,
                      &BufferLength);
        //Указать номер версии.
        Version->Caption = (char *)BufferPtr;

        GlobalUnlock(MemPtr);
        GlobalFree(MemHandle);
     }else {
        ProductName->Caption = "";
        Version->Caption = "";
     }

     Comments->Caption = "Thank you for trying this "
                         + "fabulous product.\n"
                         + "We hope that you have "
                         + "enjoyed using it.";
}
```

Этот код извлекает название программы и номер ее версии на основании информации о версии проекта. Для указания такой информации в проекте следует выбрать команду меню Project⇨Options, а затем в диалоговом окне Project Options щелкнуть на вкладке Version Info и установить флажок Include Version Information in Project. Затем следует прокрутить список параметров в таблице Key/Value, чтобы найти поле ProductName и ввести в него название программы, например **My Demo Product**. По умолчанию нумерация версий проекта начинается с номера 1.0.0.0. Щелкните на кнопке OK для закрытия диалогового окна Project Options. Впоследствии, после добавления окна About в проект, информация о номере версии будет автоматически отображаться в нем без необходимости изменения кода окна About.

В листинге 2.1 надписи с комментарием Comments присвоена длинная строка. Для ее отображения выберите надпись Comments в окне конструктора форм, щелкните на вкладке Properties в окне Object Inspector, и введите значения **265** и **35** для свойств Width (ширина) и Height (высота), соответственно. Задайте значение **false** для свойства Set AutoSize. На этом создание стандартного окна About завершается.

Для тестирования корректности работы окна About нужно отобразить его в том тестовом приложении, для которого оно создавалось. Для этого щелкните на вкладке файла AboutTestForm.cpp в окне редактора кода. Добавьте строку `#include "StdAboutBoxForm.h"` сразу после строки `#include "AboutTestForm.h"` в верхней части модуля. Поместите кнопку (компонент Button) в центре формы в окне конструктора форм, а потом создайте обработчик события OnClick для этой кнопки, дважды щелкнув на ней. В окно с кодом обработчика события введите следующий код:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    StdAboutBox->ShowModal();
}
```

Полностью код модуля AboutTestForm.cpp приведен в листинге 2.2.

Листинг 2.2. Полный код модуля AboutTestForm.cpp

```
//---------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop
#include "AboutTestFOrm.h"
#include "StdAboutBoxForm.h"
//---------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent*Owner)
  :TForm(Owner)
{
}
//---------------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  StdAboutBox->ShowModal();
}
//---------------------------------------------------------------
```

Во время работы программы этот код отобразит окно About при щелчке на созданной нами кнопке. Вследствие использования функции ShowModal() окно About создано как модальное, т.е. пользователь сможет продолжить работу с формой только после его закрытия. Сохраните все изменения проекта, выбрав команду меню File⇨Save All. Этот пример находится в файле AboutTestProj.bpr в папке ORAboutBox на компакт-диске, который прилагается к книге.

Выполните компиляцию и запустите приложение, выбрав команду меню Run⇨Run или нажав клавишу <F9>. После щелчка на кнопке на экране появится окно About в таком виде, в котором оно показано на рис. 2.2. Для закрытия окна About щелкните на кнопке OK. Возвратившись к форме, закройте приложение.

Теперь, проверив корректность работы окна About, его можно добавить в репозитарий объектов. Элементы, добавленные в репозитарий объектов, будут представлены в списке команд меню File⇨New. Щелкните правой кнопкой мыши на окне About в окне конструктора форм и выберите команду Add To Repository из контекстного меню. В появившемся на экране диалоговом окне укажите его название и описание, например так, как показано на рис. 2.3. В списке Page выберите соответствующую вкладку репозитария объектов, например Forms, для добавления в него окна About. Введите ваше имя в поле Author и, в случае необходимости, укажите пиктограмму, которая будет представлять это приложение. Щелкните на кнопке OK для сохранения окна About в репозитарии объектов. Выбрав команду File⇨New и щелкнув на вкладке Forms, вы сможете увидеть только что созданное стандартное окно About.

Кроме форм, в репозитарий объектов могут быть добавлены целые проекты. Это также можно сделать с помощью команды меню Project⇨Add To Repository.

На заметку

Прежде чем добавлять объекты в репозитарий, убедитесь, что текущая форма или проект сохранены. Дело в том, что несохраненные объекты не могут быть добавлены в репозитарий объектов. Если программист забудет это сделать, то C++Builder предложит сохранить их. Если такой объект уже существует в репозитарии объектов, то C++Builder предложит заменить его.

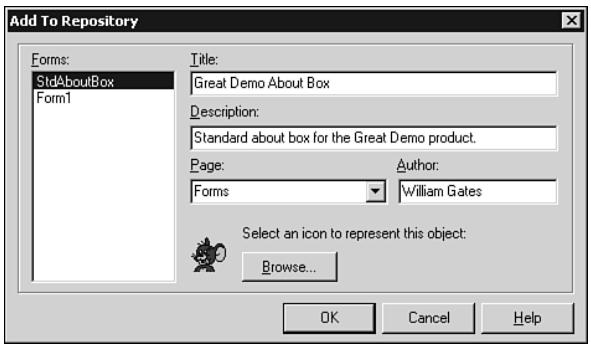


Рис. 2.3. Добавление диалогового окна в репозитарий объектов

Элементы репозитария объектов

Для доступа к элементам, которые были ранее добавлены в репозитарий объектов, выберите команду меню File⇨New. Как видите, репозитарий объектов имеет несколько вкладок, и каждая из них содержит элементы, которые можно использовать повторно.

Элементы, показанные во вкладках репозитария объектов, могут отображаться в различных форматах. Для просмотра списка предусмотренных стилей представления элементов щелкните правой кнопкой мыши на диалоговом окне репозитария объектов. Например, для отображения имени, описания, модифицированной даты и времени, а также автора каждого элемента в табличном формате следует выбрать команду View Details.

В конце каждой вкладки располагаются три переключателя: Copy (копировать), Inherit (наследовать) и Use (использовать). Они демонстрируют мощность и гибкость методов работы с репозитарием объектов.

Параметр Copy позволяет создать дубликат избранного элемента. Изменения дубликата никак не повлияют на оригинальный элемент в репозитарии объектов. Аналогично, изменения оригинального элемента не повлияют на дубликат.

Параметр Inherit аналогичен параметру Copy. Разница заключается в том, что новый объект является производным от оригинального объекта. Следовательно, изменения оригинального объекта в репозитарии объектов также будут внесены в объект-наследник.

Наконец, параметр Use позволяет непосредственно открыть и изменить элемент, выбранный из репозитария объектов. Непосредственное использование элементов не рекомендуется, если только вы не работаете с модулями данных.

Совместное использование в проекте элементов репозитария объектов

Открыв в текущем проекте с помощью команды меню File⇨New диалоговое окно New Items, вы найдете в нем вкладку с именем вашего проекта. В этой вкладке перечислены все формы и модули данных вашего проекта.

Репозитарий объектов позволяет наследовать существующие формы, что особенно удобно при необходимости создания в проекте нескольких аналогичных форм.

Настройка параметров репозитария объектов

Элементы и вкладки репозитария объектов можно настроить нужным образом, изменяя его параметры с помощью диалогового окна Object Repository, показанного на рис. 2.4. Для его отображения выберите команду меню Tools⇨Repository или команду меню File⇨New, а затем щелкните правой кнопкой мыши в появившемся диалоговом окне и в контекстном меню выберите команду Properties.

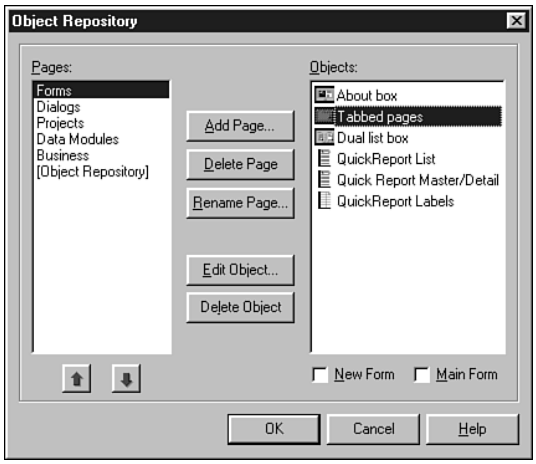


Рис. 2.4. Диалоговое окно Object Repository

В первом поле со списком содержится перечень всех имеющихся вкладок. С помощью расположенных внизу кнопок с изображениями стрелок можно изменить расположение вкладки. Кроме того, вкладки можно добавлять, удалять или переименовывать, используя соответствующие кнопки в центре диалогового окна.

На заметку

Только пустые вкладки могут быть удалены. Поэтому перед удалением страницы следует удалить все ее объекты.

Во втором поле со списком перечислены все элементы выбранной вкладки. Для работы с ними будут доступны только две кнопки: одна — для удаления элемента и другая — для редактирования его описания. В зависимости от типа объекта можно установить флажки следующих параметров.

- New Form (новая форма). При выборе команды меню File⇨New Form будет открыта указанная и принимаемая по умолчанию форма.
- Main Form (основная форма). При создании нового проекта будет открыта указанная и принимаемая по умолчанию основная форма.
- New Project (новый проект). При выборе команды меню File⇨New Application будет открыт указанный и принимаемый по умолчанию проект.

Создание программы-мастера и включение ее в репозитарий объектов

Программы-мастера (которые также известны как программы-эксперты) — это приложения, которые с помощью последовательности диалоговых окон помогают создавать проекты

или фрагменты кода или просто отображают некоторую информацию. В C++Builder содержится несколько программ-мастеров. Например, мастер создания консольных приложений ***Console Wizard*** и мастер создания приложений с помощью MFC-библиотеки классов ***MFC Wizard***, возможно, уже известны читателю.

Выше уже описывались методы создания и добавления форм и проектов в репозитарий объектов. Очевидно, что программам-мастерам требуется обеспечить более тесную интеграцию с IDE-средой C++Builder. Эта цель достигается с помощью класса `TIExpert`, который является компонентом интерфейса Open Tools API.

На заметку

Интерфейс Open Tools API (OTA) состоит из восьми модулей, каждый из которых содержит интерфейсы для взаимодействия с IDE-средой. Хотя OTA-интерфейс официально не документирован, но все же наилучшим способом изучения программы-мастера является ознакомление с его исходным кодом, находящимся в папке `\Source\ToolsAPI` основной папки, в которой инсталлирован C++Builder.

Для создания программы-мастера нужно определить потомка класса `TIExpert`. Среди функций — членов класса `TIExpert` есть функции, предназначенные для сбора информации о новой программе-мастере: `GetName()`, `GetAuthor()`, `GetComment()`, `GetPage()`, `GetGlyph()`, `GetStyle()`, `GetState()`, `GetIDString()`, `GetMenuText()`, для выполнения `Execute()`, а также конструктор и деструктор. Полное описание этих функций можно найти в файлах исходного кода OTA.

Программу-мастер можно компилировать в виде пакета (приемы работы с которыми описываются в следующем разделе) или в виде DLL-модуля. Автор предпочитает использовать пакеты, которые проще инсталлируются и сопровождаются. Сначала для этого нужно создать новый пакет с помощью репозитария объектов, выбрав команду меню File⇨New, а затем — пиктограмму Package из вкладки New. Затем добавьте в файл с исходным кодом, `Package1.cpp`, строку `#include <ExptIntf.hpp>`, описание класса `TWizard`, описание функции `TWizard::Execute()`, а также описание пространства имен, как показано в листинге 2.3. Этот пакет в полном виде под именем `MyWizard.bpk` можно найти в папке `ORWizard` на компакт-диске, прилагаемом к этой книге.

Листинг 2.3. Исходный код файла `MyWizard.cpp`

```
//---------------------------------------------------------------
#include <vcl.h>
#include <ExptIntf.hpp>
#pragma hdrstop
USERES("MyWizard.res");
USEPACKAGE("vcl50.bpi");
//---------------------------------------------------------------
#pragma package(smart_init)
//---------------------------------------------------------------
//Код пакета.
class __declspec(delphiclass)TWizard :public TIExpert
{
public:
   String __stdcall GetName()
      {return "My Wizard";};
   String __stdcall GetAuthor()
      {return "Yoto Yotov";};
   String __stdcall GetComment()
```

```
      {return "A sample wizard";};
   String __stdcall GetPage()
      {return "New";};
   HICON __stdcall GetGlyph()
      {return hIcon;};
   TExpertStyle __stdcall GetStyle()
      {return esProject;};
   TExpertState __stdcall GetState()
      {return TExpertState()<<esEnabled;};
   String __stdcall GetMenuText()
      {return "Wizard";};
   String __stdcall GetIDString()
      {return "Yotov.Wizard";};
   virtual void __stdcall Execute(void);

   __fastcall TWizard()
      {hIcon =LoadIcon(NULL,IDI_APPLICATION);};
   virtual __fastcall ~TWizard(){};
protected:
   HICON hIcon;
};

void __stdcall TWizard::Execute()
{
   Application->MessageBox("Wizard is available!",
                           "Congratulations!", MB_OK);
}

#pragma argsused
int WINAPI DllEntryPoint(HINSTANCE hinst,
                         unsigned long reason, void*)
{
   return 1;
}
//---------------------------------------------------------------

namespace Mywizard
{
   void __fastcall PACKAGE Register()
   {
       RegisterLibraryExpert(new TWizard());
   }
}
```

В этом коде реализованы необходимые функции — члены класса `TIExpert`; например, функция `GetName()` задает имя `My Wizard` для данной программы-мастера в репозитарии объектов, а функция `GetPage()` — вкладку New, в которой эта программа-мастер будет находиться.

Для включения созданной программы-мастера в репозитарий объектов необходимо откомпилировать и инсталлировать ее. Откройте редактор кода с помощью команды меню View⇨Toggle Form/Unit (Вид⇨Переключиться к форме/модулю) или нажмите клавишу

<F12>, находясь в редакторе кода. Затем щелкните на кнопке Compile (Компилировать) в окне редактора кода, а после окончания процесса компиляции — на кнопке OK. Для инсталляции пакета просто щелкните на кнопке Install (Инсталлировать) в окне редактора кода — на экране появится диалоговое окно с сообщением об успешной инсталляции пакета.

Теперь с помощью команды меню File⇨New откройте диалоговое окно New Items и убедитесь, что в нем находится созданная вами программа-мастер `My Wizard`. Конечно, для того чтобы функция-член `Execute()` выполняла более полезные действия, ее код должен быть сложнее. Читателю предоставляется возможность попробовать свои силы в создании такого кода.

Пакеты

В этом разделе рассматриваются способы использования пакетов при работе с приложениями и IDE-средой. Термин "пакет" относится к модулю с исходным кодом с расширением `.bpk` (Borland Package) и, в более общем смысле, — к файлу с расширением `.bpl` (Borland Package Library — BPL), который создается при сборке пакета. BPL-файл очень похож на DLL-модуль и используется с той же целью, т.е. для динамического связывания (или динамической компоновки) его исходного кода с приложением. Более подробную информацию о сходствах и различиях между Borland Package Library и DLL можно найти в главе 15.

Существует три основных типа пакетов: *только для создания* (*designtime-only*), *только для выполнения* (*runtime-only*) или *для создания/для выполнения* (*designtime/runtime*). Единственное отличие между ними заключается в том, что пакеты только для создания могут быть инсталлированы в IDE-среде, а пакеты только для выполнения — нет. Дело в том, что пакеты только для выполнения могут быть использованы только во время работы готового приложения. Пакеты для создания/для выполнения могут использоваться в любой ситуации и часто применяются на начальном этапе создания компонентов. Применение пакетов для инсталляции и распространения компонентов подробно рассматривается в главе 11.

Пакет состоит из двух разделов: `Contains` и `Requires`. В разделе `Contains` содержатся именно те файлы пакета, которые компилируются и компонуются при компиляции и компоновке самого пакета и являются его частью.

Указанные в разделе `Requires` файлы импорта являются ссылками на пакеты для выполнения, к которым данный пакет должен осуществить доступ для корректного выполнения своих функций. Более подробно этот механизм описывается ниже в этом разделе. При сборке пакета компилируются и компонуются все соответствующие файлы раздела `Contains`, и для каждого из них создается объектный файл.

Кроме того, генерируется BPL-файл в виде файла импорта Borland Package Import (BPI) с расширением `.bpi` и в виде статической библиотеки (LIB) с расширением `.lib`. Чтобы исключить генерацию BPI-файла выберите вкладку Linker в диалоговом окне Project Options, которое появляется после выбора команды меню Project⇨Options, и снимите флажок параметра Generate import library (Генерировать библиотеку импорта) в группе параметров Linking (Компоновка). Аналогично, чтобы не генерировать LIB-файл, снимите флажок параметра Generate .lib file (Генерировать статическую библиотеку).

BPI-файл нужно генерировать всегда, поскольку он используется IDE-средой во время компоновки, чтобы выполняемый файл мог использовать соответствующий BPL-файл при выполнении приложения. Исключением является пакет, предназначенный только для создания приложения. Следовательно, для всех типов пакетов создаются одинаковые типы файлов, а отличаются только способы их использования. На рис. 2.5 показана структура пакета и файлы, создаваемые после успешной сборки пакета.

Обратите внимание на то, что на рис. 2.5 под модулями пакета подразумеваются единицы трансляции C++. Следовательно, после успешной компиляции и компоновки для каждой единицы трансляции будет создан объектный файл. В раздел `Contains` пакета также можно добавить другие файлы, обычно это файлы ресурсов и объектные файлы. Они требуются в тех случаях, когда пакет используется для компонентов пакета. Более подробно эта тема рассматривается в главе 11.

На рис. 2.5 показано, что раздел `Requires` содержит файл импорта `vcl50.bpi`. Это значит, что данному пакету *потребуется* выполняемый код, который содержится в файле `vcl50.bpl`.

Размещение файла импорта `vcl50.bpi` в разделе `Requires` пакета позволяет компоновщику разрешить любые внешние ссылки на рабочую библиотеку `vcl50.bpl`. Основная VCL-библиотека нужна почти для всех пакетов, поэтому файл импорта помещается в разделе `Requires` практически всех создаваемых пакетов.

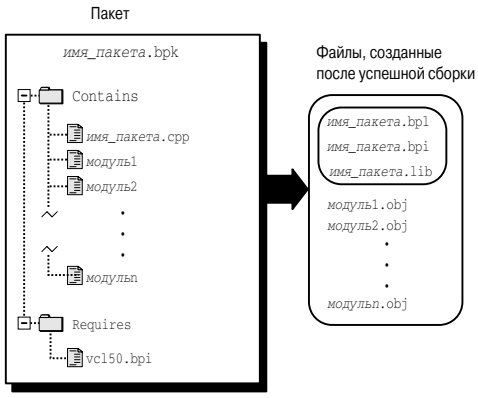


Рис. 2.5. Структура пакета и файлы, создаваемые в результате успешной сборки

На рис. 2.5 показано, что после сборки пакета создаются три файла. Тип BPL-файла зависит от назначения пакета: т.е. создан ли пакет только для создания или только для выполнения. Если пакет имеет двойное назначение, то BPL-файл обладает двойной функциональностью. BPL-файл в пакете только для создания используется для инсталляции этого пакета в IDE-среде. Это делается с помощью команды меню Component⇨Install Packages и поиска нужного BPL-файла для пакета только для создания. BPL-файл в пакете только для выполнения используется только для динамического связывания приложений с функциями и данными, которые содержит пакет.

Для динамического связывания приложения с BPL-файлом компоновщик должен иметь возможность разрешить все ссылки на функции и данные (т.е. найти эти функции и данные), которые содержатся в BPL-файле. Такая ссылка называется *внешней*, потому что она ссылается на нечто внешнее по отношению к приложению. Для разрешения внешней ссылки компоновщик ищет *запись импорта* (*import record*) для функции, вызываемой приложением. Запись импорта содержится в соответствующем файле импорта пакета Borland Package Import (BPI) с расширением `.bpi`.

Для каждой функции, *экспортируемой* из пакета (в основном функции, которая объявлена в модуле, содержащемся в данном пакете), существует запись, содержащая внутреннее имя функции и имя модуля этой функции. В данном случае это имя BPL-файла пакета и другая более подробная информация, которая не имеет отношения к обсуждаемой здесь теме. Эта информация копируется компоновщиком в выполняемый файл приложения. В результате появляется возможность создать динамическую связь с функцией, которая будет установлена операционной системой Windows в каждом случае выполнения приложения. В момент загрузки приложения в оперативную память выполняется поиск файла с именем, указанным в записи импорта, среди

всех файлов в системном каталоге (например, среди файлов в системном каталоге Windows). Если такой BPL-файл найден, он также загружается в оперативную память. Внешняя ссылка в таком случае указывает на расположение функции (или данных) внутри BPL-файла. Если нужный BPL-файл уже загружен в оперативную память, то это еще лучше, потому что с выполнением этой операции не будут связаны дополнительные накладные расходы, а данный BPL-файл будет совместно использоваться несколькими приложениями. Именно таким образом функция или данные будут использованы вызывающим их приложением.

Следует помнить, что BPL-файлы и BPI-файлы, созданные с помощью пакета, предназначенного только для выполнения, используются для поддержки динамического связывания кода, экспортируемого пакетом. А LIB-файл используется для поддержки статического связывания кода, экспортируемого пакетом. По сути, LIB-файл включает OBJ-файлы модулей, которые содержатся в самом пакете, а потому является *объектной библиотекой*. Если необходимо использовать функцию из пакета, соответствующий объектный файл из LIB-файла копируется в целевой выполняемый файл. Каждый такой выполняемый файл имеет собственную копию.

В табл. 2.1 приведены основные сведения о файлах, которые создаются пакетом после его успешной сборки.

Таблица 2.1. Файлы, получаемые после успешной сборки пакета

Расширение	Описание	Тип	Назначение
`.bpl`	Borland Package Library (BPL)	Динамически связываемая библиотека	Содержит выполняемый код пакета и экспортирует функции и данные пакета.
			Доступ к библиотеке пакетов только для выполнения возможен со стороны тех приложений, которые динамически связаны с ней.
			Библиотека пакетов только для создания, которая может быть инсталлирована в IDE-среде для создания новых компонентов или редакторов, доступна только при создании приложений
`.bpi`	Borland Import Library (BPI)	Библиотека импорта	Содержит записи импорта для функций и данных, экспортируемых соответствующим BPL-файлом, который необходим для динамического связывания с BPL-файлом
`.lib`	Static Library File (LIB)	Объектная библиотека	Статическая библиотека, содержащая объектные файлы модулей, которые находятся в пакете. Используется для статического связывания экспортируемых функций и данных в целевом приложении

В табл. 2.1 показано, что для использования BPL-файла библиотеки пакетов только для выполнения, необходимо во время компоновки иметь соответствующий BPI-файл, который используется для разрешения внешних ссылок на BPL-файл. На рис. 2.6 показана связь между файлами пакета и приложением, которое использует этот пакет.

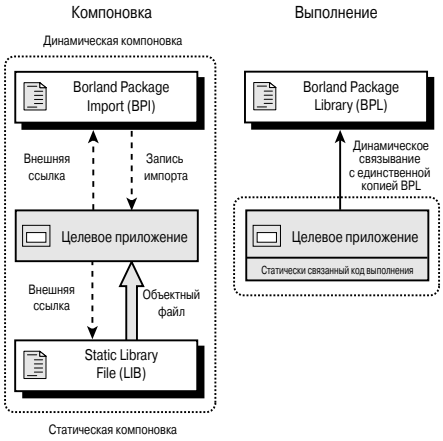


Рис. 2.6. Схема связи между пакетом и приложением

При компиляции и компоновке проекта можно выбрать тип связывания с нужными пакетами, т.е. динамический или статический. По умолчанию используется динамическое связывание. Для изменения этого параметра следует выбрать команду меню Project⇨Options и во вкладке Packages снять флажок параметра Build with runtime packages (Вместе с пакетами для выполнения).

Кроме того, можно выбрать отдельные модули пакета, которые требуется связать статически с модулем проекта. Для этого нужно в начало модуля добавить строку `#pragma link "`*`имя_модуля`*`"`, где *`имя_модуля`* — это имя именного того модуля, который требуется статически связать с заданным модулем внутри проекта. Компоновщик скопирует нужный объектный файл из LIB-файла пакета.

Особенности применения пакетов

Для корректного импорта и экспорта пакета нужно использовать макрос `PACKAGE` сразу после ключевого слова `class` в определении класса, например, так, как показано ниже.

```
class PACKAGE TMyComponent :public Tcomponent
{
    //Определение класса компонента.
};
```

Макрос `PACKAGE` также должен быть указан в объявлении функции `Register()` регистрации этого компонента, как показано ниже.

```
namespace Newcomponent
{
    void __fastcall PACKAGE Register()
    {
        TComponentClass classes[1] = {__classid(TMyComponent)};
        RegisterComponents("Samples", classes, 0);
    }
}
```

Это, конечно, относится и к определениям класса компонента. При использовании программы-мастера ***Component Wizard*** для создания компонентов в C++Builder предусмотрена автоматическая вставка макроса `PACKAGE`.

Для корректного импорта и экспорта функций и данных из содержащегося в пакете модуля, в файле с его исходным кодом после директивы `#include` перед самим кодом следует разместить строку `#pragma package(smart_init)`. В противном случае компиляция этого модуля все же будет выполнена, но пакет не удастся скомпоновать статически. Эта директива предназначена для инициализации модулей пакета в порядке, определенном их зависимостями.

Строку `#pragma package(smart_init, weak)` можно использовать, чтобы исключить модуль из BPL-файла и поместить его в BPI-файле. В случае необходимости этот модуль копируется из BPI-файла и статически связывается с целевым приложением. Такой модуль назы-

вается *слабо запакованным* (*weakly packaged*) и используется для предупреждения конфликтов среди пакетов, которые могут зависеть от той же внешней библиотеки.

Пакеты для выполнения

В C++Builder предусмотрено несколько пакетов для выполнения, которые содержат основную библиотеку VCL, а также другие библиотеки и компоненты. В табл. 2.2 перечислены эти пакеты, а также модули, экспортируемые из каждого пакета.

Таблица 2.2. Пакеты для выполнения

Пакеты для выполнения	Экспортируемые модули	Выполняемые функции
dss50.bpl	Mxarrays,Mxbutton, Mxcommon, Mxconsts, Mxdb, Mxdcube, Mxdsql, Mxgraph, Mxgrid, Mxpbar, Mxpivsrc, Mxqedcom, Mxqparse, Mxstore, Mxtables	Компоненты Decision Cube
ibevnt50.bpl	Ibconsts, Ibctrls, Ibevnts, Ibproc32	События InterBase
inet50.bpl	Copyprsr, Httpapp, Webconst	Компоненты Internet
inetdb50.bpl	Dbweb, Dsprod	Компоненты Internet
nmfast50.bpl	Nmconst, Nmdaytim, Nmecho, Nmextstr, Nmfngr, Nmftp, Nmhttp, Nmmsg, Nmnntp, Nmpop3, Nms_huge, Nms_stream, Nmsmtp, Nmstrm, Nmtime, Nmudp, Nmurl, Nmuue, Psock	Компоненты FastNet Internet
qrpt50.bpl	Qr3const, Qrabout, Qrcomped, Qrctrls, Qrdatawz, Qrenved, Qrexpbld, Qrexport, Qrexpr, Qrexpred, Qrextra, Qrlabled, Qrlablwz, Qrprev, Qrprgres, Qrprnsu, Qrprntr, Qrwizard, Quickrpt	Компоненты Qreport
tee50.bpl	Arrowcha, Brushdlg, Bubblech, Chart, Ganttch, Pendlg, Series, Tecanvas, Teeabout, Teeconst, Teefunci, Teengine, Teeprevi, Teeprocs, Teeshape, Teexport	Класс TChart
teedb50.bpl	Dbchart	Класс TChart (база данных)
teeqr50.bpl	Qrtee	Класс TChart (компонент QReport)
teeui50.bpl	Areaedit, Arrowedi, Axisincr, Axmaxmin, Baredit, Custedit, Dbeditch, Editchar, Flineedi, Ganttedi, Iedi3d, Iediaxis, Iedigene, Iedilege, Iedipage, Iedipane, Iediperi, Iediseri, Ieditcha, Ieditit1, Iediwall, Pieedit, Shapeedi, Teegally, Teelisb, Teepoedi, Teestore	Класс TChart (интерфейс пользователя)

Окончание табл. 2.2

Пакеты для выполнения	Экспортируемые модули	Выполняемые функции
vcl50.bpl	Activex, Actnlist, Axctrls, Buttons, Classes, Clipbrd, Comconst, Comctrls, Commdlg, Comobj, Comstrs, Consts, Contnrs, Controls, Dialogs, Dsgnintf, Dsgnwnds, Editintf, Exptintf, Extctrls, Extdlgs, Fileintf, Flatsb, Forms, Graphics, Grids, Imglist, IniFiles, Istreams, Libhelp, Libintf, Mapi, Mask, Masks, Math, Menus, Mtx, Multimon, Oleconst, Olectnrs, Olectrls, Oleserver, Printers, Proxies, Registry, Scktcomp, Stdactns, Stdctrls, Stdvcl, Svcmgr, Syncobjs, Sysconst, System, Sysutils, Toolintf, Toolsapi, Toolwin, Typinfo, Vclcom, Virtin	Стандартные, дополнительные, системные, Win32 и диалоговые компоненты
vclado50.bpl	Adoconst, Adodb, Adoint, Oledb	Компоненты ADO
vclbde50.bpl	Bde, Bdeconst, Dbinpreq, Dbpwdlg, Dbtables, Smintf	Ядро Borland Database Engine
vcldb50.bpl	Db, Dbactns, Dbcgrids, Dbcommon, Dbconsts, Dbctrls, Dbgrids, Dblogdlg, Dbolectl	Компоненты доступа к данным и управления данными
vcldbx50.bpl	Dblookup, Report, Rsconsts	Компоненты TDBLookup
vclib50.bpl	Ib, Ibbatchmove, Ibblob, Ibcustomdataset, Ibdatabase, Ibdatabaseinfo, Ibevents, Ibexternals, Ibheader, Ibinstall, Ibintf, Ibquery, Ibservices, Ibsql, Ibsqlmonitor, Ibstoredproc, Ibtable, Ibupdatesql, Ibutils	Компоненты InterBase
vclie50.bpl	Mshtml, Shdocvw	Internet Explorer
vcljpg50.bpl	Jconsts, Jpeg	Работа с JPEG-файлами
vclmid50.bpl	Corbacon, Corbaobj, Corbardm, Corbcnst, Corbastd, Databkr, Dbclient, Dsintf, Mconnect, Midas, Midascon, Midconst, Mtsrdm, Objbrkr, Orbpas, Provider, Sconnect	Компоненты MIDAS
vclsmp50.bpl	Calender, Diroutln, Gauges, Spin	Компоненты-примеры
vclx50.bpl	Checklst, Colorgrd, Ddeman, Filectrl, Mplayer, Outline, Tabnotbk, Tabs	Компоненты Win 3.1 и другие системные компоненты
webmid50.bpl	Compprod, Miditems, Midprod, Scrptmgr, Pagitems, Wbmconst, Webcomp, Xmlbrokr	Компоненты Internet Express

19 пакетов из 22 пакетов, перечисленных в табл. 2.2, входят в состав профессиональной версии C++Builder. Пакет ADO (`vclado50.bpl`), MIDAS (`vclmid50.bpl`) и пакет Internet Express Package (`webmid50.bpl`) входят в состав только корпоративной версии C++Builder. Эту таблицу можно использовать как справочное руководство для получения сведений о пакетах, которые нужно вставить в раздел `Requires` создаваемого вами пакета.

Утилита `tdump`

Входящая в состав C++Builder (и находящаяся в каталоге `Bin`) утилита `tdump` используется для проверки содержимого файлов BPL, BPI и LIB. Действительно, эта утилита позволяет проверить состав любого выполняемого или библиотечного файла. Для этого нужно открыть окно сеанса DOS и ввести показанную ниже строку, используя фактические имена файла вместо имен-заменителей, представленных в наклонном начертании.

`tdump` *`проверяемый_файл файл_вывода_результатов`*`.dmp`

После нажатия клавиши <Enter> файл *`файл_вывода_результатов`*`.dmp` будет содержать в текстовом формате содержимое файла *`проверяемый_файл`*. Эта очень полезная утилита, поэтому рекомендуется внимательно изучить ее возможности. Например, для просмотра содержимого BPL-файла в командной строке следует использовать ключ `-ee`, как показано ниже.

`tdump -ee` *`проверяемый_файл`*`.bpl` *`файл_вывода_результатов`*`.dmp`

А для просмотра только тех модулей, которые импортируются в BPL-библиотеку, в командной строке нужно использовать ключ `-em` согласно приведенному ниже синтаксису.

`tdump -em` *`проверяемый_файл`*`.bpl` *`файл_вывода_результатов`*`.dmp`

Список ключей этой утилиты можно получить, введя в командной строке `tdump` и нажав клавишу <Enter>.

Новые компоненты IDE-среды в C++Builder 5

В C++Builder версии 5 включено много новых компонентов IDE-среды. Здесь кратко рассматриваются несколько наиболее полезных новинок — категории свойств и изображения свойств в окне Object Inspector, новый формат расширенного языка маркировки или XML-формат Extensible Markup Language (XML) файла проекта, способы сохранения файлов форм в текстовом формате, параметры проекта на уровне узлов, списки задач (To-Do list) для проекта и определенного фрагмента кода, а также новая программа-мастер ***Console Wizard*** для создания консольных приложений.

Категории свойств окна Object Inspector

В главе 1 уже описывались способы отображения и редактирования свойств компонентов и функций-обработчиков событий в окне Object Inspector IDE-среды. В C++Builder 5 вводится новое понятие категорий свойств. Все свойства (включая события, которые также являются свойствами) теперь распределены по категориям в окне Object Inspector и упорядочены в алфавитном порядке. Основная функция категорий — собрать связанные по смыслу свойства в отдельные группы. Причем свойство может находиться сразу в нескольких группах. Кроме того, свойство можно скрыть в окне Object Inspector за счет *фильтрации* (*filtering*), т.е. скрытия его категории.

В следующих разделах будет показано, как использовать категории свойств для группирования связанных по смыслу свойств в окне Object Inspector. Это в равной степени относится и к событиям. Сведения о способах регистрации категорий для свойств в настраиваемых компонентах можно найти в главе 10.

Категории свойств

По умолчанию свойства отображаются в окне Object Inspector в алфавитном порядке. Для просмотра их по категориям нужно щелкнуть правой кнопкой мыши на рабочей области окна Object Inspector и выбрать из контекстного меню команду Arrange⇨By Category. После этого в окне Object Inspector будут отображены категории свойств. Категории можно развернуть или свернуть, щелкнув на пиктограмме с изображением знака + (развернуть) или знака - (свернуть) (по умолчанию в исходном состоянии все категории свернуты). Результат этих действий остается неизменным при выполнении других операций с категориями и свойствами. Например, если некая категория развернута до перехода к просмотру свойств в алфавитном порядке, то после возврата к просмотру свойств по категориям она снова будет развернута. Наконец, если выбран другой компонент и он имеет свойства в той же категории, то эта категория также будет развернута.

Для фильтрации категорий свойств, отображаемых в окне Object Inspector, снова щелкните правой кнопкой мыши и выберите в контекстном меню команду View. Каждая категория может быть отмечена или не отмечена флажком, причем неотмеченные категории будут скрыты.

При выборе команды меню View⇨All будут автоматически отмечены все категории, соответственно, при выборе View⇨None все категории станут неотмеченными (а значит, скрытыми), а при выборе команды меню View⇨Toggle состояние категории будет заменено обратным. Таким образом, отмеченные категории можно сделать неотмеченными, и наоборот. Изменение значения свойства, которое входит в несколько категорий, приводит к тому, что оно будет изменено во всех этих категориях.

Установленные категории свойств

Установлено 13 категорий свойств (и событий), 12 из которых используются библиотекой VCL в C++Builder. Они перечислены в табл. 2.3 вместе с кратким описанием типа свойств, содержащихся в каждой категории и примерами этих свойств.

Таблица 2.3. Категории свойств в C++Builder

Название категории	Назначение категории
`Action`	Содержит свойства, которые управляются действиями и поведение которых связано с функциями периода выполнения приложения. В этой категории находятся свойства `Hint` и `Checked` компонента `TMenuItem`
`Data`	Содержит свойства, которые управляют данными компонента. Эта категория в настоящее время библиотекой VCL не используется. Сначала она использовалась для свойств `Text`, `EditMask` и `Tag`, но теперь они располагаются в категориях `Localizable`, `Localizable` и `Miscellaneous`, соответственно
`Database`	Содержит свойства, поведение которых связано с операциями над базой данных. Например, в этой категории находятся свойства `DatabaseName`, `MasterSource` и `OnCalcFields` компонента `TTable`
`Drag, Drop, and Docking`	Содержит свойства, связанные с операциями перетаскивания и опускания элементов, а также их закрепления. Например, в этой категории находятся свойства `OnDragOver` и `DockSite` компонента `TForm`

Окончание табл. 2.3

Название категории	Назначение категории
Help and Hints	Содержит свойства (и события), связанные с получением справки, совета или другого вида помощи. Например, в этой категории находятся свойства OnHint и HelpContext компонента TStatusBar
Input	Содержит свойства, связанные с управлением ввода в компонент. В этой категории находятся свойства OnKeyPress, OnClick и Enabled компонента TForm
Layout	Содержит свойства, связанные со структурой и внешним видом элемента управления во время его создания. В этой категории находятся свойства OnResize и Width компонента TForm
Legacy	Содержит устаревшие свойства. В этой категории находятся свойства Ctl3D и OldCreateOrder компонента TForm
Linkage	Содержит свойства, относящиеся к связыванию компонентов. В этой категории находятся свойства PopupMenu компонента TForm и DataSource компонента TDBGrid
Locale	Содержит свойства, связанные с локализацией программы или совместимостью с локализованными версиями операционных систем. В этой категории находятся свойства BiDiMode и ParentBiDiMode компонента TForm
Localizable	Содержит свойства, которые могут изменяться в зависимости от места инсталляции приложения. В этой категории находятся свойства BiDiMode, Hint и Font компонента TForm
Miscellaneous	Содержит свойства, которые не подпадают ни под одну категорию или которые нет необходимости приписывать к какой-либо категории. В этой категории находятся свойства Tag и Name компонента TForm
Visual	Содержит свойства, связанные со структурой и внешним видом элемента управления во время работы приложения. В этой категории находятся свойства BorderStyle, Color и Width компонента TForm

Эта таблица категорий указывает на способы применения категорий библиотекой VCL. Однако, истинное представление о свойствах, содержащихся в этих категориях, можно получить только в ходе постоянной работы с ними в окне Object Inspector.

Изображения в списках прокрутки окна Object Inspector

В C++Builder 5 класс TPropertyEditor расширен за счет включения шести новых функций (более подробная информация по этому поводу приводится в главе 10), которые поддерживают создание изображений в списках прокрутки окна Object Inspector. Это позволяет создать в нем интуитивно понятный интерфейс для соответствующих свойств.

Редакторы свойств, которые перегружают эти свойства, приведены в табл. 2.4, вместе со свойствами библиотеки VCL, которые зарегистрированы для совместного использования с ними. Эти свойства имеют соответствующие изображения, связанные со значением каждого свойства.

Таблица 2.4. Редакторы свойств со встроенной поддержкой изображений

Класс редактора свойств	*Свойство библиотеки VCL*
TBrushStyleProperty	TBrush::Style
TColorProperty	Tcolor
TCursorProperty	Tcursor
TFontNameProperty	TFont::Name
TPenStyleProperty	TPen::Style
TComponentImageIndexPropertyEditor	TImageIndex
TListViewColumnImageIndexPropertyEditor	TListColumn:: ImageIndex
TMenuItemImageIndexPropertyEditor	TMenuItem::ImageIndex
TPersistentImageIndexPropertyEditor	TImageIndex

Если глобальная переменная FontNamePropertyDisplayFontNames (в файле $(BCB)\Include\Vcl\dsgnintf.hpp) имеет принимаемое по умолчанию значение false для редактора свойств TFontNameProperty, то имена шрифтов не будут показаны в окне Object Inspector. Вместо этого будет использован фактически используемый шрифт, например принимаемый по умолчанию шрифт Microsoft Sans Serif. Это обусловлено тем, что перерисовка таких изображений очень медленно выполняется на маломощных компьютерах или на компьютерах, где установлено много шрифтов. Для просмотра таких изображений следует задать для этого свойства значение true. Некоторая задержка возникнет только при первом отображении этого свойства, а потому этот режим приемлем для большинства компьютеров. Установка значения true для свойства FontNamePropertyDisplayFontNames — не совсем простая задача, для выполнения которой потребуется применить функции Open Tools API. Альтернативным решением является инсталляция редактора свойств, который подменяет свойство TFontNameProperty. Этот подход подробно описывается в главе 10.

Свойства, зарегистрированные с помощью редакторов свойств TComponentImageIndexPropertyEditor, TListViewColumnImageIndexPropertyEditor, TMenuItemImageIndexPropertyEditor и TPersistentImageIndexPropertyEditor, будут отображать на экране изображения только в том случае, если соответствующее свойство типа TImageList имеет компонент-наследник TCustomImageList, содержащий необходимые изображения.

Редактор свойств TComponentImageIndexPropertyEditor предназначен для свойств TImageIndex компонентов-наследников компонента TComponent, причем родительский компонент содержит свойство TCustomImageList под именем Images. Этот редактор свойств используется для настраиваемых (пользовательских) компонентов, которые удовлетворяют этим критериям. Если в родительском компоненте для свойства TCustomImageList требуется использовать другое имя, то необходимо создать новый редактор свойств, производный от данного. Это не очень практично по следующим причинам. Во-первых, нельзя включить заголовочный файл $(BCB)\Include\Vcl \stdreg.hpp, в котором объявлены эти четыре редактора свойств TImageIndex. Во-вторых, код перерисовки не совсем корректно отображает и размещает на экране большие изображения, а потому для этого лучше использовать подмену редактора свойств, как показано в главе 10.

Редакторы свойств TListViewColumnImageIndexPropertyEditor и TMenuItemImageIndexPropertyEditor применяются для свойств TImageIndex с родительским компонентом, имеющим компонент типа TCustomImageList под именем SmallImages.

Редактор свойств TPersistentImageIndexPropertyEditor используется для свойств TImageIndex в классах-потомках компонента TPersistent (а не TComponent) с родительским

компонентом, имеющим компонент типа `TCustomImageList` с именем `Images`. Этот редактор свойств используется для настраиваемых компонентов, которые отвечают этим критериям. Если в родительском компоненте для свойства `TCustomImageList` требуется использовать другое имя, то и в этом случае необходимо создать новый редактор свойств, производный от данного. Для этого лучше использовать подмену редактора свойств, как показано в главе 10.

На рис. 2.7 показан способ применения свойства `ImageIndex` компонента `TMenuItem` вместе с компонентом `TImageList` для отображения списка изображений `TImageList` в списке прокрутки этого свойства. В этом примере для использования списка изображений нужно задать значение `ImageList1` для свойства `Images` компонента `TMainMenu`.

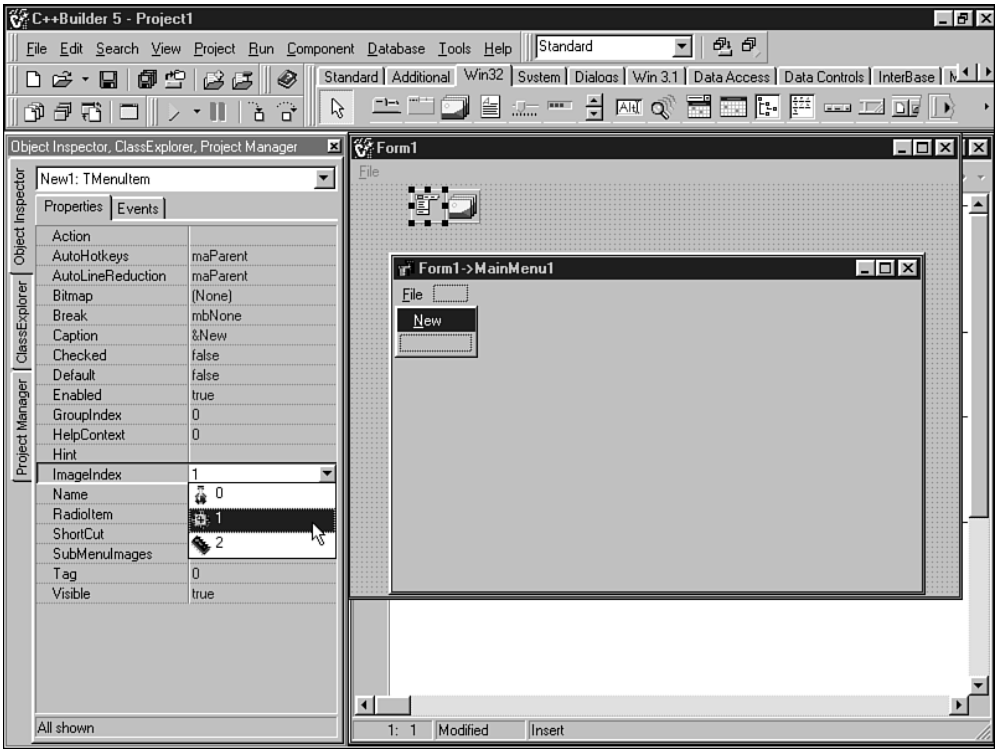


Рис. 2.7. Использование изображений для свойства `ImageIndex` в компоненте `TMenuItem`

На рис. 2.8 показано несколько редакторов свойств, которые содержат изображения в окне Object Inspector, с использованием компонента `TShape`, добавленного в качестве примера в форму, показанную на рис. 2.7.

Более подробная информация о способах использования списков прокрутки с изображениями в настраиваемых компонентах приводится в главе 10. В ней также приводится иерархия классов, производных от класса `TPropertyEditor`.

Файл проекта в формате XML

В C++Builder 5 формат Makefile для файлов проекта (`.bpr` и `*.bpk`) заменен XML-форматом (Extensible Markup Language — XML). Просматривать и редактировать файл проекта в формате XML в IDE-среде можно в окне редактора кода, которое отображается на эк-

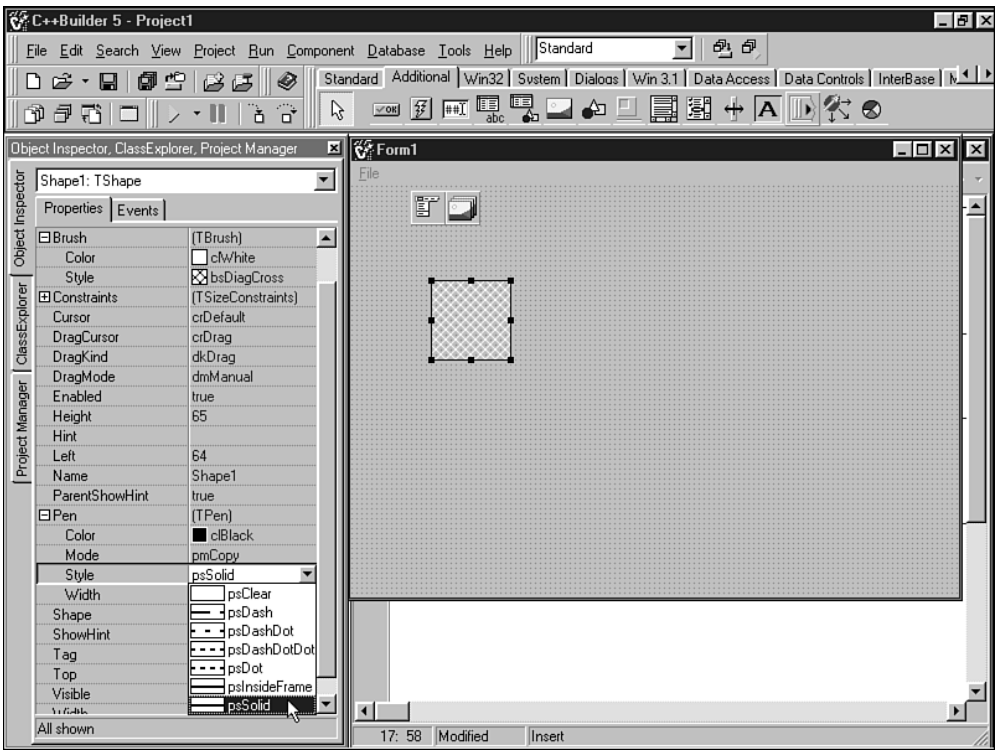


Рис. 2.8. Раскрывающиеся списки с изображениями для свойства `Style` компонента `TShape`

ране после выбора команды меню Project⇨Edit Option Source. В предыдущих версиях C++Builder для этого нужно было выбрать команду меню Project⇨View Makefile. Для совместимости с уже существующими проектами файл проекта можно отформатировать в формате Makefile с помощью шаблона и экспортировать его. Для этого нужно просто выбрать команду меню Project⇨Export Makefile. Можно также выбрать команду меню Export Makefile для отдельного проекта из контекстного меню в окне менеджера проектов Project Manager. Кроме того, файл проекта в формате Makefile можно экспортировать с помощью консольного инструмента `BPR2MAK.EXE`.

При загрузке в C++Builder 5 уже существующего проекта в формате Makefile формат его файла автоматически преобразуется в новый формат XML. При этом на экране отображается диалоговое окно с информацией о таком преобразовании.

В листинге 2.4 приводится упрощенная версия файла проекта в формате Makefile для программы Doodle, которая поставляется в составе C++Builder 4. В листинге 2.5 этот же файл показан в новом формате XML, т.е. в том виде, в каком он поставляется с C++Builder 5. Оба эти файла приведены с существенными упрощениями только для иллюстрации излагаемого материала и не являются рабочими файлами проекта.

Листинг 2.4. Упрощенный пример файла проекта Doodle в формате Makefile, используемом в C++Builder 4

```
PROJECT = doodle.exe
OBJFILES = doodle.obj main.obj palette.obj

CFLAG1 = -I$(BCB)\include;$(BCB)\include\vcl \
         -Od -Hc -H=$(BCB)\lib\vcl40.csm \
```

```
        -w -Ve -r--k -y -v -vi \
        -D$(SYSDEFINES);$(USERDEFINES)-c -b -w-par \
        -w-inl -Vx -tW

[Version Info ]
MajorVer=1
MinorVer=0
```

Листинг 2.5. Упрощенный пример файла проекта Doodle в формате XML, используемом в C++Builder 5

```
<?xml version='1.0' encoding=='utf-8' ?>
<PROJECT>
  <MACROS>
    <PROJECT value="doodle.exe"/>
    <OBJFILES value="DOODLE.obj main.obj palette.obj"/>
  </MACROS>
  <OPTIONS>
    <CFLAG1 value="-Od -H=$(BCB)\lib\vcl50.csm -Hc -Vx \
                  -Ve -r--a8 -k -y -v -vi--c -b- -tW"/>
  </OPTIONS>
  <IDEOPTIONS>
[Version Info ]
MajorVer=1
MinorVer=0
  </IDEOPTIONS>
</PROJECT>
```

Для большинства пользователей не имеет значения, в каком формате находится файл проекта — в формате Makefile или формате XML. Дело в том, что файл проекта автоматически создается на основе модулей проекта и параметров, заданных в диалоговом окне Project Options после выбора команды меню Project⇨Options.

Чтобы для компиляции проекта можно было применить консольную утилиту MAKE, файл проекта придется экспортировать в формате Makefile.

Сохранение форм в текстовом формате

Новинкой в C++Builder версии 5 является возможность сохранения форм в текстовом формате, а не в двоичном формате, как это было в предыдущих версиях. Этот параметр используется по умолчанию, однако в случае необходимости формы могут быть сохранены также в двоичном формате.

Щелкните правой кнопкой мыши на любой форме проекта для открытия контекстного меню. Как показано на рис. 2.9, по умолчанию возле команды меню Text DFM будет стоять флажок.

При сохранении модуля формы или проекта в целом эта форма будет сохранена в текстовом формате в DFM-файле, как показано в листинге 2.6.

Листинг 2.6. Пример сохранения DFM-файла в текстовом формате

```
object Form1:TForm1
  Left = 192
  Top = 107
  Width = 311
  Height = 158
```

```
  Caption = 'Text Form'
  Color = clBtnFace
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  OldCreateOrder = False
  PixelsPerInch = 96
  TextHeight = 13
    object Label1:TLabel
    Left = 13
    Top = 24
    Width = 277
    Height = 20
    Caption = 'This form is saved as text (default)'
    Font.Charset = DEFAULT_CHARSET
    Font.Color = clWindowText
    Font.Height = -16
    Font.Name = 'MS Sans Serif'
    Font.Style = [fsBold]
    ParentFont = False
  end
  object Button1:TButton
    Left = 114
    Top = 72
    Width = 75
    Height = 25
    Caption = 'OK'
    TabOrder = 0
    OnClick = Button1Click
  end
end
```

Снятие флажка параметра формы Text DFM приведет к тому, что форма будет сохранена в двоичном формате. Для того чтобы этот флажок устанавливался по умолчанию для всех новых форм нужно, как показано на рис. 2.10, снять флажок параметра New Forms as Text option во вкладке Preferences диалогового окна Environment Options, которое отображается после выбора команды меню Tools⇨Environment Options.

В папке TextForms на прилагаемом к этой книге компакт-диске содержится два примера проектов: в проекте FormText.bpr параметр Text DFM используется для сохранения формы в текстовом формате, а в проекте FormBinary.bpr этот параметр не используется, и форма сохраняется в двоичном формате.

Параметры на уровне узлов

Благодаря новому компоненту Node-Level Options интегрированной среды разработки теперь можно подменять параметры проекта для отдельных узлов (например, для файлов с исходным кодом) проекта. Дело в том, что, помимо набора глобальных параметров, существуют также локальные параметры для каждого узла проекта, которые по умолчанию имеют те же значения, что и глобальные параметры.

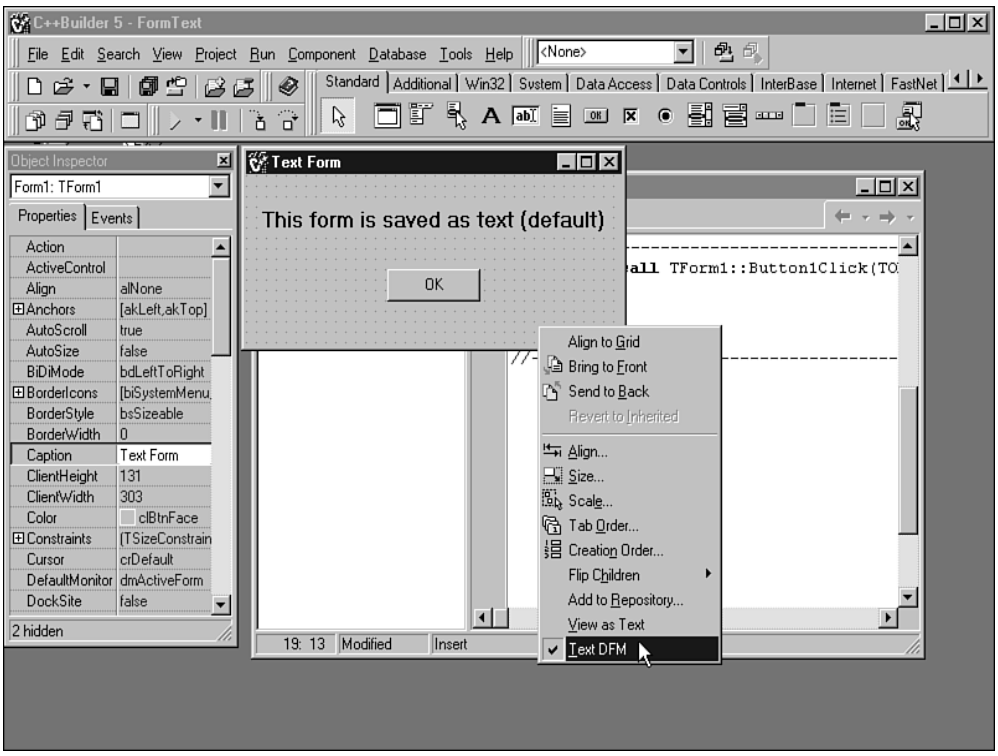


Рис. 2.9. Параметр Text DFM *(сохранить в текстовом формате) в контекстном меню формы*

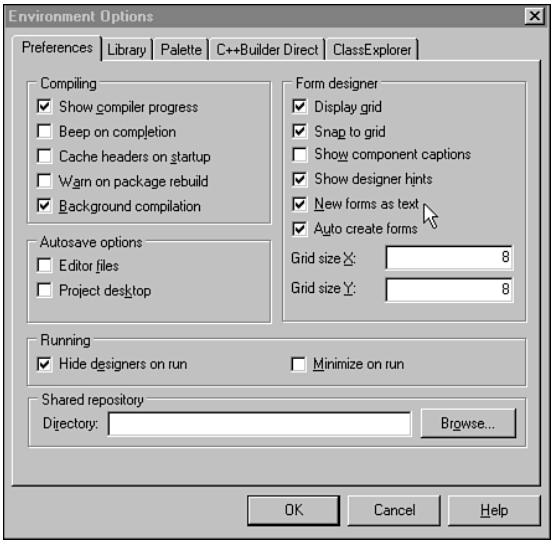


Рис. 2.10. Вкладка Preferences *диалогового окна* Environment Options

Установка глобальных и узловых параметров

Глобальные параметры можно установить в диалоговом окне Project Options, которое отображается на экране после выбора команды меню Project⇨Options. По внешнему виду и функциональным возможностям они аналогичны параметрам проекта в предыдущих версиях C++Builder. Установленные здесь значения параметров принимаются по умолчанию для всех узлов проекта.

Рассмотрим их использование на примере приложения Doodle, которое поставляется с C++Builder 5. Предположим, что по умолчанию нам требуется отключить оптимизацию кода для всех модулей проекта, но хотелось бы включить ее для модуля `main.cpp`.

На рис. 2.11 показано диалоговое окно Project Options с глобальными параметрами приложения Doodle, в котором отключена оптимизация кода.

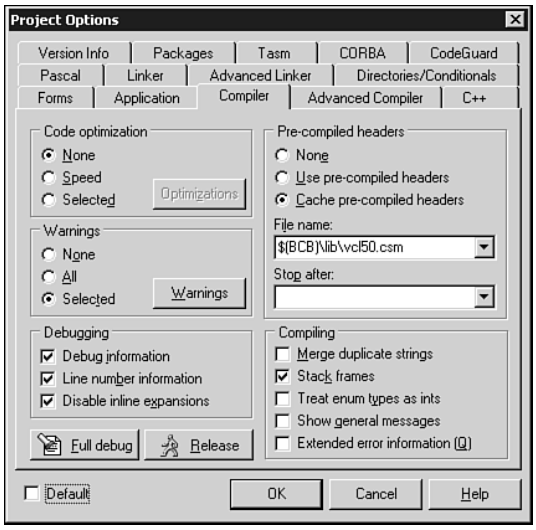


Рис. 2.11. Диалоговое окно Project Options с глобальными параметрами приложения Doodle с отключенной оптимизацией кода

Для подмены заданных по умолчанию параметров проекта можно установить значения локальных параметров отдельного узла с помощью диалогового окна Local Options. Это окно отображается на экране компьютера с помощью команды Edit Local Options контекстного меню окна Project Manager. На рис. 2.12 показано окно Project Manager для приложения Doodle вместе с контекстным меню модуля `main.cpp`, в котором выбрана команда Local Options.

На рис. 2.13 показано диалоговое окно Local Options этого узла с включенным параметром оптимизации по скорости, который подменяет принимаемое по умолчанию значение параметра проекта. Обратите внимание на то, что при установке локальных параметров они подсвечиваются другим цветом для указания подмены принимаемых по умолчанию значений параметров проекта.

После подмены параметра узла в диалоговом окне Local Options, этот узел отмечается в окне Project Manager красной галочкой, как показано на рис. 2.14 на примере узла `main.cpp`.

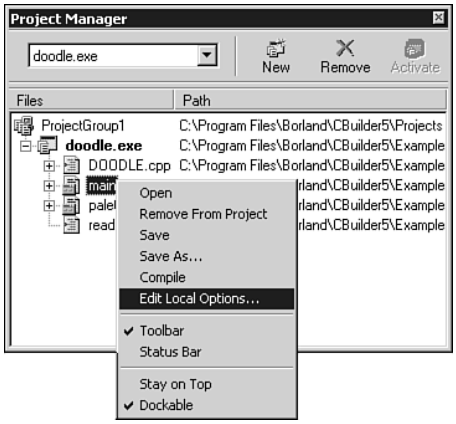


Рис. 2.12. Окно Project Manager и команда Edit Local Options контекстного меню для узла модуля main.cpp приложения Doodle

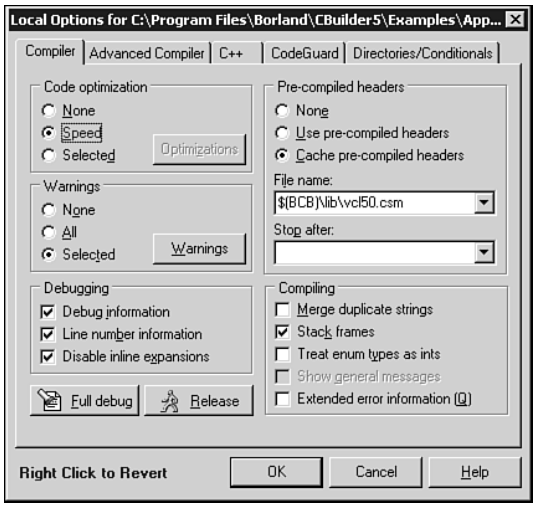


Рис. 2.13. Диалоговое окно Local Options и параметры узла main.cpp

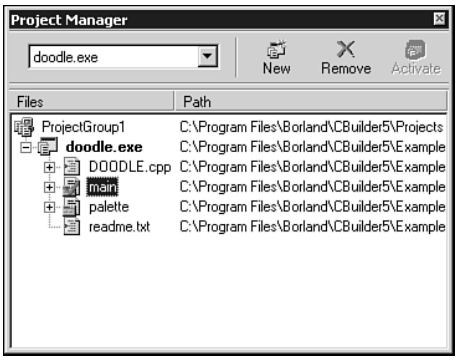


Рис. 2.14. Узел main.cpp отмечен галочкой для обозначения подмены значения параметра

Обращение значений узловых параметров

Для обратной замены локального значения параметра значением, заданным по умолчанию для всего проекта, следует выбрать команду Revert из контекстного меню нужного параметра. Для обратной замены локальных значений всех параметров вкладки локальных параметров следует выбрать команду Revert из контекстного меню всей вкладки. Для обратной замены локальных значений параметров целого узла следует выбрать команду Revert All из контекстного меню, которое отображается после щелчка правой кнопкой мыши на надписи Right Click to Revert. После завершения установки значений щелкните на кнопке OK.

Способы применения узловых параметров

Возможность подмены параметров проекта на уровне узлов повышает гибкость процесса создания проекта. Хотя этой возможностью придется пользоваться не слишком часто, в некоторых случаях она может оказаться очень полезной. Ниже перечислены примеры подмены принимаемых по умолчанию значений параметров проекта.

- Для оптимизации по размеру, применяемой по умолчанию для всего проекта, за исключением модулей, для которых необходимо включить оптимизацию по скорости.
- Для использования в модуле другой версии включаемого файла за счет указания пути к этому файлу.
- Для применения особых условных директив `define` с целью изменения поведения модуля.
- Для отключения выдачи некоторых предупреждений для определенного модуля, чтобы исключить появление предупреждающих сообщений для корректно работающего кода.
- Для использования других технических условий языка C++ для модуля, поставленного или созданного в соответствии с другими стандартами.

Помимо описанных случаев, вы столкнетесь со множеством других ситуаций, когда может потребоваться подмена значений узловых параметров.

Новый список неотложных задач

Существенным усовершенствованием C++Builder 5 является список неотложных задач (To-Do List). Каждый элемент этого списка (некоторая задача) может быть добавлен в него глобально для всего проекта или локально для фрагмента кода, что означает связь этого элемента с отдельным модулем или фрагментом кода.

Этот список имеет очень большое значение в работе программиста, поскольку практически каждый проект представляет собой список задач, выполненный с помощью специализированного проектирующего программного обеспечения или написанный вручную. Новый компонент To-Do List в C++Builder 5 позволяет непосредственно соединить его с кодом, что позволяет как упростить процесс документирования, так и быстрее найти эти задачи впоследствии.

Проект Crozzle (который более подробно рассматривается в главе 6, посвященной компиляции и оптимизации приложения) будет использован в приведенных ниже примерах для демонстрации составных элементов компонента To-Do List. Его можно найти в папке главы 6 на прилагаемом к этой книге компакт-диске.

Элементам списка неотложных задач может быть присвоен приоритет Priority (от 0 до 5), владелец Owner (персона или группа лиц), а также категория Category. Это помогает управлять содержимым списка, который может содержать огромное количество элементов даже в проекте небольшого размера.

Просмотр элементов списка неотложных задач

Для просмотра списка неотложных задач следует выбрать команду меню View⇨To-Do List (Вид⇨Список неотложных задач) в IDE-среде C++Builder. В нем перечислены три типа элементов: глобальные (Global) элементы показаны обычным шрифтом, локальные (Local) элементы в открытом и активном модуле — полужирным, а локальные элементы в других модулях — шрифтом серого цвета. Имена модулей для локальных элементов списка неотложных задач показаны в диалоговом окне To-Do List.

На рис. 2.15 показан список неотложных задач для проекта Crozzle, содержащий несколько элементов. В этом разделе мы рассмотрим общие вопросы управления элементами с помощью диалогового окна To-Do List. А в следующих разделах приводятся более подробные сведения об использовании глобальных и локальных элементов.

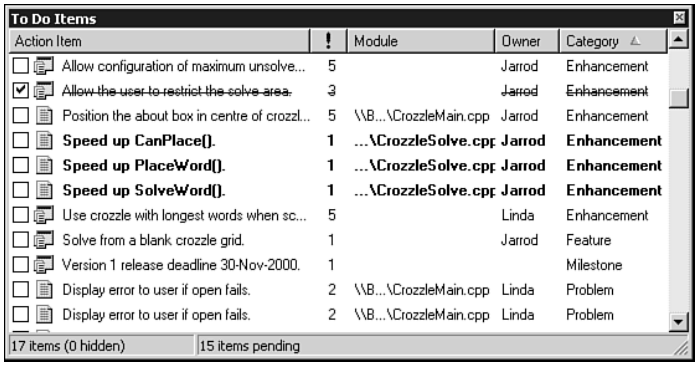


Рис. 2.15. Список неотложных задач проекта Crozzle в диалоговом окне To-Do List

В диалоговом окне To-Do List большинство задач выполняется с помощью команд контекстного меню. Например, с их помощью можно добавить в список глобальные элементы, удалить или изменить элементы, отсортировать их по любому полю, а также фильтровать по категории (Category), владельцу (Owner), типу (Type) и статусу (Status).

Список неотложных задач можно скопировать в буфер обмена в виде текста или HTML-таблицы для последующего использования в другой документации или программном обеспечении (например, вставить его в Web-страницу с описанием текущего состояния в сети intranet). Свойства HTML-таблицы можно задать с помощью команды Table Properties контекстного меню.

Для отметки выполненной задачи нужно установить флажок параметра Status. В таком виде элемент останется в списке до удаления.

Глобальные элементы списка неотложных задач

Глобальные элементы списка неотложных задач хранятся в файле *имя_проекта*.todo, который автоматически создается и добавляется в проект при создании глобального элемента списка неотложных задач. Ниже перечислены примеры использования глобальных элементов списка.

- Напоминания о датах выполнения ключевых этапов проекта.
- Список компонентов, которые предстоит создать.
- Список компонентов, которые следует добавить в следующей версии.
- Ошибки, которые не были устранены в каком-либо фрагменте кода.

При загрузке файла CrozzleProj.todo проекта Crozzle в окно текстового редактора (например, Notepad) список неотложных задач будет заключен в фигурные скобки ({}). Он состоит из не-

скольких полей и описаний, которые образуют список глобальных неотложных задач проекта. Для управления ими следует использовать диалоговое окно To-Do List IDE-среды C++Builder.

Для вставки глобального элемента в проект нужно щелкнуть правой кнопкой мыши и выбрать команду меню Add из контекстного меню в диалоговом окне To-Do List. После этого в соответствующие поля следует ввести описание, приоритет, имя владельца и категорию, а затем щелкнуть на кнопке OK. При этом все ранее использованные в проекте значения полей Owner и Category можно будет выбрать в списках прокрутки.

Локальные элементы списка неотложных задач

Локальный элемент списка неотложных задач имеет вид комментария в коде приложения. Он автоматически добавляется в исходный код при добавлении метода в объект с помощью диалогового окна Add Method из контекстного меню обозревателя классов Class Explorer.

Локальные элементы можно добавить и вручную, указав место в коде, куда они должны быть вставлены. Их можно размещать практически в любом месте программы (включая заголовочные файлы), за исключением верхней части модуля над объявлениями включаемых файлов. Ниже перечислены возможные варианты использования локальных элементов списка неотложных задач.

- Сообщение об ошибке.
- Замечание о том, что данная часть кода нуждается в улучшении.
- Отметка кода, вызывающего подозрения.
- Указатель на определенное место.
- Указание на изменения, которые необходимо внести в руководство пользователя, как следствие изменения кода.

Локальные элементы списка неотложных задач объявляются в коде в том же формате, что и комментарии языка C++. По внешнему виду в файле *имя_проекта*`.todo` они аналогичны глобальным элементам. В листинге 2.7 показана часть кода из проекта Crozzle, которая содержит локальный элемент списка неотложных задач.

Листинг 2.7. Локальный элемент списка неотложных задач в файле `CrozzleSolve.cpp`

```
//---------------------------------------------------------------------------
void __fastcall
   TCrozzleForm::HelpAboutItemClick(TObject *Sender)
{
  TAboutForm *AboutForm = new TAboutForm(Application);
  /*TODO 5 -oJarrod -cEnhancement
           : Position the about box in centre of
             crozzle window.*/
  AboutForm->ShowModal();
}
```

Комментарий основан на нескольких полях, причем некоторые из них необязательны и могут отсутствовать, если их значения не заданы. Приоритет 0 в комментарии также не показывается. Вот как выглядит синтаксис такого комментария:

```
/* статус [приоритет] [-o владелец] [-c категория] : описание */
```

Для него можно использовать и такой вариант синтаксиса:

```
// статус [приоритет] [-o владелец] [-c категория] : описание
```

Простейший способ добавления локального элемента заключается в выборе команды Add To-Do Item из контекстного меню, которое отображается на экране после щелчка правой кнопкой мыши в соответствующем месте окна редактора кода. Для вставки локального элемента в место расположения указателя мыши можно также нажать комбинацию клавиш <Shift+Ctrl+T>. Кроме того, локальные элементы можно ввести вручную с помощью редактора кода, используя приведенный выше синтаксис (после синтаксического анализа комментариев они будут автоматически преобразованы в локальные элементы списка To-Do List).

Так как текст описания локального элемента размещается в комментарии к коду, то в описании неотложных задач следует избегать применения символов /* или */. Дело в том, что компоненты To-Do List, Class Explorer, Code Insight, а также компилятор могут неправильно интерпретировать этот комментарий.

Из диалогового окна To-Do List можно непосредственно перейти к тому месту кода, к которому относится данный локальный элемент списка, дважды щелкнув мышью или выбрав команду Open из контекстного меню.

Если локальный элемент отмечен в диалоговом окне To-Do List как завершенный (Complete), то такой комментарий останется в коде, но его статус TODO (необходимо сделать) в комментарии будет заменен статусом DONE (сделано). Чтобы иметь возможность просмотра выполненных задач, не удаляйте завершенные элементы списка, оставляя их в коде. Для удаления незавершенного или завершенного элемента следует выбрать команду Delete из контекстного меню диалогового окна To-Do List или удалить его комментарий в коде.

Читателю предлагается самостоятельно поэкспериментировать с компонентом To-Do List, учитывая, что его нужно использовать аккуратно, т.е. не допуская устаревания его элементов.

Программа-мастер *Console Wizard* для создания консольных приложений

Программа-мастер ***Console Wizard*** для создания консольных приложений является одной из нескольких программ-мастеров C++Builder версии 5. Она имеет несколько простых инструментов, упрощающих создание консольных приложений, которые являются приложениями Win32, но не имеют графического интерфейса пользователя.

Запуск программы-мастера *Console Wizard*

Программу-мастера ***Console Wizard***, как и другие программы-мастера создания приложений, можно запустить, выбрав команду меню File⇨New. Затем нужно выбрать пиктограмму ***Console Wizard*** из диалогового окна New Items, показанного на рис. 2.16.

На экране будет отображено диалоговое окно этой программы-мастера со следующими параметрами.

- Source Type (тип исходного кода). Указывает язык (C или C++), на котором будет создан код приложения.
- Use VCL (использовать библиотеку VCL). Позволяет использовать компоненты библиотеки VCL. Этот параметр становится доступным только в том случае, когда в качестве языка исходного кода указан язык C++.
- Multi Threaded (многопоточность). Возможность выполнения нескольких потоков. Этот параметр автоматически выбирается, если выбран параметр Use VCL.
- Console Application (консольное приложение). Выбор этого параметра приводит к созданию консольного окна для этого приложения.

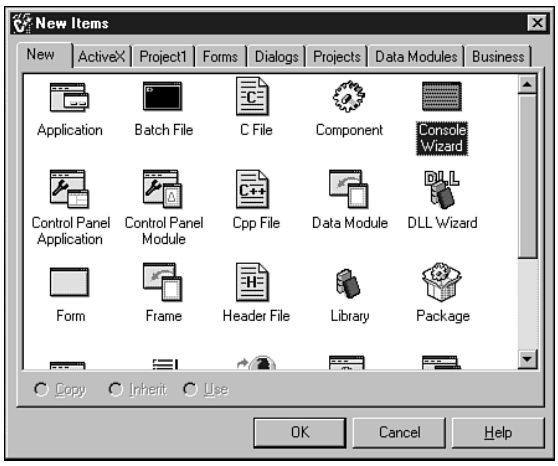


Рис. 2.16. Пиктограмма программы-мастера **Console Wizard** *в диалоговом окне* New Items

Кроме того, в текстовом окне в нижней части этого диалогового окна можно указать расположение уже существующего файла с исходным кодом, который будет автоматически загружен. Диалоговое окно с параметрами программы-мастера ***Console Wizard*** показано на рис. 2.17.

После указания всех параметров и щелчка на кнопке OK в окне редактора кода будет создан каркас программы. Например, выбор только параметров C++ и Console Application приведет к созданию скелета функции `main()`, показанной в листинге 2.8.

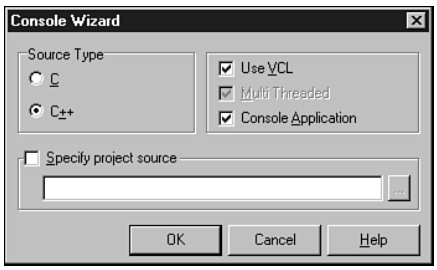


Рис. 2.17. Диалоговое окно с параметрами программы-мастера **Console Wizard**

Листинг 2.8. Каркас кода консольного приложения

```
//---------------------------------------------------------

#pragma hdrstop

//---------------------------------------------------------

#pragma argsused
int main(int argc,char*argv [])
{
    return 0;
}
//---------------------------------------------------------
```

Пример создания простого консольного приложения

В этом кратком примере для создания простого консольного приложения используются параметр C из группы параметров Source Type и параметр Console Application для генерации консольного окна. Добавление в функцию-заготовку `main()` показанного в листинге 2.9 кода по-

сле запуска этой программы приведет к результату, показанному на рис. 2.18. Полностью этот проект под именем `MyConsole.bpr` хранится в папке `Console` на прилагаемом к этой книге компакт-диске.

Листинг 2.9. Код простого консольного приложения в файле `MyConsoleCode.c`

```
//---------------------------------------------------------------

#pragma hdrstop

#include <stdio.h>
#include <conio.h>
//---------------------------------------------------------------

#pragma argsused
int main(int argc,char*argv [])
{
    printf("Hello World!\n");
    getch();

    return 0;
}
//---------------------------------------------------------------
```

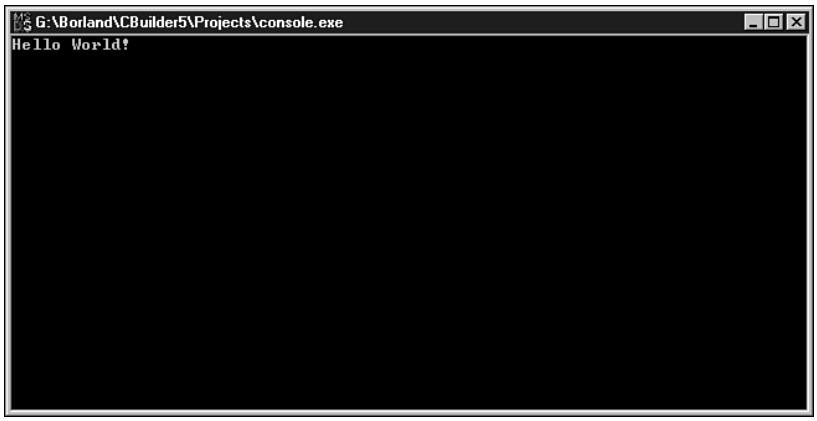


Рис. 2.18. Результат работы простого консольного приложения

Резюме

В этой главе подробно рассмотрены проекты C++Builder, описаны способы повторного использования проектов и других программируемых элементов с помощью репозитария объектов, а также представлены пакеты и способы их применения. Помимо этого рассмотрены новые компоненты IDE-среды C++Builder 5.

IDE-среда — очень мощное и действенное средство повышения производительности труда, а потому описанные в этой главе компоненты IDE-среды могут существенно упростить процесс программирования. В следующих главах эти компоненты будут рассмотрены более подробно.

Глава

3

Стили и методы программирования в среде C++Builder

Джэйми Оллсоп

В этой главе представлены наиболее важные сведения о стилях и методах программирования в среде C++Builder.

В первом разделе основное внимание уделяется методам повышения читабельности кода, поскольку оптимизация зрительного восприятия кода имеет очень большое значение. Более простой для чтения код легче воспринимать и сопровождать, что положительно сказывается на стоимости всего проекта. Оптимизация читабельности заключается не просто в улучшении внешнего вида кода при его размещении на странице, но и в описании кода, которое адекватно отображает его назначение, содержание и поведение.

Создание определенного стиля кодирования и неуклонное следование ему — вероятно, наиболее простой способ улучшения читабельности кода. Однако, следует учесть, что по некоторым параметрам одни стили превосходят другие. Именно поэтому здесь рассматривается несколько стилей программирования, чтобы читатель мог выбрать для себя наиболее подходящий. Кроме того, здесь предлагаются рекомендации по выбору наиболее эффективных стилей, а также изложены предостережения в отношении применения некоторых стилей. Действительно, некоторые эффективные методы работы при чрезмерном применении могут существенно снизить читабельность. Описанию именно таких ситуаций в этом разделе уделяется особое внимание. Основные элементы стиля кодирования рассматриваются параллельно с описанием степени их влияния на читабельность и с рекомендациями по поводу наиболее эффективного их использования. Кроме того, в некоторых случаях описаны способы эффективного использования IDE-среды.

Во втором разделе рассматриваются вопросы, касающиеся создания кода на языке C++ в приложениях C++Builder. В нем охвачен широкий круг тем: от использования операторов `new` и `delete` до применения в программах ключевого слова `const`. В этом разделе описаны аспекты кодирования, которые часто неправильно понимаются или используются программистами. Некоторые аспекты кодирования часто недооцениваются, особенно теми программистами, которые не имеют достаточного опыта программирования на языке C++. Однако излагаемый материал предназначен в основном для тех, кто имеет такой опыт программирован. Они смогут найти здесь разъяснение некоторых вопросов использования языка C++ в C++Builder, т.е. извлечь пользу из этого материала. Сложные темы начинаются с описания базовых понятий. Обсуждаются также вопросы, возникшие в результате того, что VCL-библиотека унаследовала многие элементы библиотеки Object Pascal.

Наконец, в заключительном разделе приведен список рекомендуемой литературы вместе с кратким описанием. Читателю настоятельно рекомендуется просмотреть этот список, поскольку чтение представленных в нем материалов может оказаться очень полезным.

Стили кодирования для улучшения читабельности кода

В этом разделе рассматривается необходимость повышения читабельности кода и предлагаются методы для достижения этой цели. Независимо от выбранных стилей и методов, разработчики должны строго придерживаться избранного стиля. Строгое следование одному стилю имеет очень большое значение (код в этом разделе записан с использованием разных стилей только для того, чтобы проиллюстрировать излагаемый в этой главе материал).

Краткий и простой код

Хотя это очевидно, но всегда старайтесь создавать максимально краткий и простой код. Эта цель достигается двумя способами.

Во-первых, это значит, что код в целом является набором большого количества небольших частей кода, каждая из которых служит отдельной и понятной цели. Это значит, что сложность кода определяется только уровнем абстракции, а не его количеством. Рассмотрим пример кода с двумя функциями, который показан в листинге 3.1.

Листинг 3.1. Сложность кода и уровень абстракции

```
#include <vector>

double GetMaximumValue(const std::vector<double>& Vector)
                       throw(std::out_of_range)
{
   double Maximum = Vector.at(0);

   for(int i=0; i<Vector.size(); ++i)
   {
      if(Vector[i] > Maximum)
      {
         Maximum = Vector[i];
      }
   }

   return Maximum;
}

void NormalizeVector(std::vector<double>& Vector)
{
   if(!Vector.empty())
   {
      double Maximum = GetMaximumValue(Vector);

      for(int i=0; i<Vector.size(); ++i)
      {
          Vector[i] -= Maximum;
      }
   }
}
```

Обе приведенные выше функции имеют одинаково сложный код, но вторая функция, `NormalizeVector`, выполняет более сложную задачу, поскольку ее уровень абстракции выше.

Используйте оператор пред-инкремента вместо оператора пост-инкремента. Например, в листинге 3.1 оператор `++i` используется вместо оператора `i++`. В обоих случаях значение переменной `i` будет увеличено на единицу, но пост-инкрементный оператор вернет прежнее значение переменной `i`. Так как в этом нет необходимости, то эти действия будут выполнены зря. Обычно, пост-инкрементный оператор реализуется на основе пред-инкрементного оператора, а потому пред-инкрементный оператор выполняется быстрее, чем пост-инкрементный.

При работе со встроенными типами, например `int`, эта разница не очень заметна, но в случае определенных пользователем типов она может быть существенной. В любом случае необоснованное использование пост-инкрементного оператора приведет к снижению производительности программы.

Во-вторых, это значит, что при чтении кода не следует удаляться от объявлений локальной переменной и параметров функции (которые содержат информацию о типах). С этой целью при написании больших фрагментов кода, которые выполняют сразу несколько задач, рассмотрите возможность разбиения кода на меньшие концептуально отличные блоки и перепишите их соответственно.

Выделение фрагментов программы

Простейший способ улучшения расположения кода заключается в размещении скобок ({ }) в отдельных строках с выравниванием соответствующих пар. Код внутри скобок следует отметить отступом с постоянным шагом, который будет использоваться везде в вашей программе. Обычно для отступа используются два или четыре пробела, хотя одни программисты могут использовать более длинные, а другие более короткие отступы длиной всего в один пробел. (Остерегайтесь применения очень длинных отступов, потому что они могут существенно ухудшить читабельность программ.)

Снимите флажок параметра Use Tab Character (Использование табуляции) во вкладке General диалогового окна Editor Properties, которое можно отобразить на экране с помощью команды меню Tools⇨Editor Options. В этом случае для отступа будут использоваться пробелы, а не табуляция.

Почему на это следует обращать внимание? Во-первых, благодаря таким ухищрениям упрощается визуальное считывание кода. При этом код может быть быстро разбит на составные функциональные блоки, где сущность каждого блока понятна по его первой строке. Во-вторых, очевидна область действия кода, т.е. становится ясно, какие переменные находятся в рамках или за пределами области в данный момент, что позволяет найти и устранить возможные проблемы. Эта особенность данного подхода имеет очень большое значение. Следует помнить, что область действия блока содержит выражение заголовка блока (например, ключевое слово `for` или заголовок функции), а функциональная часть блока заключена в скобки. Выравнивание скобок по уровню заголовка блока и отступ функционального кода явно показывают такую логическую связь.

Эффективность такого расположения заключается в том, что гораздо легче найти соответствие между двумя парами скобок, чем между концевой скобкой и ключевым словом (или другими их перестановками), поскольку эти символы отличаются друг от друга. Расположение фрагментов кода будет легко находить при последующем сопровождении кода, потому что каждая открывающая скобка имеет соответствующую концевую скобку в том же столбце.

В IDE-среде предусмотрен способ создания и отмены отступа для выбранного блока кода с помощью комбинаций клавиш <Ctrl+Shift+I> и <Ctrl+Shift+U>, соответственно. Количество пробелов, которые используются для отступа выбранного фрагмента, определяется значением параметра Block indent (Отступ блока) во вкладке General диалогового окна Editor Properties, которое можно отобразить на экране с помощью команды меню Tools⇨Editor Options. Наибольшую гибкость в работе обеспечивает установка значения 1 для этого параметра.

Выделение отступом скобок и кода в них не так эффективно, потому что в таком случае скобки тяжело найти. Для простых блоков кода это несложно, но при работе с вложенными блоками это может привести к путанице. Однако известно, что некоторые программисты предпочитают использовать именно такой стиль. Более подробное обсуждение этого вопроса можно найти в книге *A Practical Handbook of Software Construction* , McConnell, 1993, p. 399. Учтите, что большая часть материала в этой книге относится к языкам, в которых для указания структуры блока используются такие ключевые слова, как, например, `begin` и `end`. По-

этому смысл сказанного немного отличается, хотя это различие не принимается во внимание в упомянутой книге, и об этом не следует забывать.

Код будет легко и просто читаться, если расположить его согласно указанному выше правилу, т.е. скобки находятся в отдельных строках, в одном столбце, а код внутри скобок отмечен отступом. Циклы можно отметить как блоки, и тогда вложенные конструкции станут простыми и ясными без какой-либо двусмысленности. Это особенно удобно при использовании вложенных конструкций на основе операторов `if-else`. Например, приведенный ниже код не очень прост для восприятия.

```
#include <ostream>
// Здесь предполагается, что int A <= int B <= int C
if(A + B > C)
  if((A==B) || (B==C))
  if((A==B) && (B==C))
     std::cout << "Это РАВНОСТОРОННИЙ треугольник";
else
if( (A*A + B*B) == C*C )
  std::cout << "Это ПРЯМОУГОЛЬНЫЙ РАВНОБЕДРЕННЫЙ треугольник";
  else std::cout << "Это РАВНОБЕДРЕННЫЙ треугольник";
else
    if( (A*A + B*B) == C*C ) std::cout << "Это ПРЯМОУГОЛЬНЫЙ треугольник";
    else std::cout << "Это треугольник";
    else std::cout << "Это НЕ треугольник";
```

Смысл этого кода будет более понятен, если записать его в следующем виде.

```
#include <ostream>

// Здесь предполагается, что int A <= int B <= int C

if(A + B > C)
{
   if((A==B) || (B==C))
   {
      if((A==B) && (B==C))
      {
         std::cout << "Это РАВНОСТОРОННИЙ треугольник";
      }
      else if( (A*A + B*B) == C*C )
      {
         std::cout <<
            "Это ПРЯМОУГОЛЬНЫЙ РАВНОБЕДРЕННЫЙ треугольник";
      }
      else
      {
         std::cout << "Это РАВНОБЕДРЕННЫЙ треугольник";
      }
   }
   else if( (A*A + B*B) == C*C )
   {
      std::cout << "Это ПРЯМОУГОЛЬНЫЙ треугольник";
   }
```

```
   else std::cout << "Это треугольник";
}
else std::cout << "Это НЕ треугольник";
```

При этом всегда следует стараться найти более простой и ясный способ написания кода, особенно, если приходится создавать большие вложенные конструкции на основе операторов `if` и `if-else`. При создании большого блока `if-else` рассмотрите возможность его замены выражением с оператором `switch`. Однако это не всегда возможно. В случае больших вложенных конструкций на основе оператора `if` попытайтесь перестроить код в виде последовательных конструкций `if-else`.

Старайтесь также создавать как можно более короткие строки. Это улучшает их читабельность в окне редактора и упрощает печать.

Часто при печати кода длинные строки не помещаются на странице полностью. В таком случае часть длинной строки может быть перенесена на следующую строку или просто обрезана. Оба варианта очень нежелательны, потому что ухудшают читабельность кода.

Для предотвращения таких последствий следует либо избегать длинных строк, либо аккуратно разбивать их на несколько строк. При создании кода с помощью окна редактора кода правый край можно использовать как напоминание о ширине печатаемой области страницы. Расположение правого края определяется значением параметра Right margin (Правый край) во вкладке Display диалогового окна Editor Properties, которое можно отобразить на экране с помощью команды меню Tools⇨Editor Options. Символы, выходящие за этот край, не будут печататься на странице или будут автоматически перенесены на следующую строку, если установлен флажок параметра Wrap Lines (Перенос строк) в диалоговом окне Print Selection, которое можно отобразить на экране с помощью команды меню File⇨Print при работе с редактором кода. Общее количество печатаемых символов зависит от ширины страницы и положения левого края, которое задается в этом же окне с помощью параметра Left Margin (Левый край).

Для страницы формата A4 с положением левого края в позиции 0 правый край должен находиться в позиции 94. Это значит, что строка кода, длиннее этой величины, не будет напечатана. Чтобы отобразить правый край кода в окне редактора кода установите флажок параметра Visible Right Margin (Показать правый край кода) во вкладке Display диалогового окна Editor Properties, которое можно отобразить на экране с помощью команды меню Tools⇨Editor Options.

Наиболее распространенными причинами чрезмерного увеличения строк кода являются глубоко вложенные циклы или условия выбора на основе оператора `switch` со сложным кодом внутри, выражения с операторами `for` и `if` в одной строке, функции с длинным списком параметров, длинные логические выражения внутри выражений на основе оператора `if`, а также конкатенация строк с помощью оператора `+`.

Для упрощения глубоко вложенных циклов и условий выбора, рекомендуется сократить шаг отступа или перестроить код таким образом, чтобы определенная часть работы выполнялась с помощью функций.

Например, показанное ниже выражение на основе оператора `switch` можно переписать так, как показано в следующем примере.

```
switch(Key)
{
  case 'a' : // очень длинная строка кода, который не виден
```

```
            // в пределах окна редактора...
            break;
  case 'b' : // еще одна длинная строка кода ...
            break;
  default  : // выполняемые по умолчанию действия
            break;
}
```

Такой код можно записать иначе.

```
switch(Key)
{
  case 'a'
  : // очень длинная строка кода, который теперь виден
    // в пределах окна редактора...
    break;
  case 'b'
  : // еще одна длинная строка кода ...
    break;
  default
  : // выполняемые по умолчанию действия
    break;
}
```

Строку с выражениями на основе операторов `for` и `if` следует разбить на отдельные строки, например, так, как показано ниже для оператора `for`.

```
for(int i=0; i<10; ++i) // очень длинная строка кода, который
                        // не виден в пределах окна редактора
```

Этот код можно записать иначе.

```
for(int i=0; i<10; ++i)
{
  // очень длинная строка кода, который теперь виден
  // в пределах окна редактора...
}
```

Аналогично можно поступить с длинным выражением на основе оператора `if`.

```
if( Key == 'a' || Key == 'A' ) // очень длинная строка кода,
                               // который не виден в
                               // пределах окна редактора...
```

Этот код можно заменить приведенным ниже кодом.

```
if( Key == 'a' || Key == 'A' )
{
  // очень длинная строка кода, который теперь виден
  // в пределах окна редактора...
}
```

Действительно, несколько однострочных выражений лучше записывать по отдельности, так как в этом случае с помощью построчной отладки можно проследить за ходом выполнения каждого выражения программы.

Длинные логические операторы внутри выражений на основе оператора `if` следует разбить на несколько строк, например так, как показано ниже.

```
if(Key == VK_UP || Key == VK_DOWN || Key == VK_LEFT...
{
  // Код
}
```

Это выражение лучше записать в следующем виде.

```
if(Key == VK_UP
   || Key == VK_DOWN
   || Key == VK_LEFT
   || Key == VK_RIGHT
   || Key == VK_HOME
   || Key == VK_END)
{
  // Код
}
```

В этом примере поднят еще один важный вопрос, касающийся расположения оператора || или другого аналогичного оператора. Здесь он располагается слева, потому что строка читается слева направо. Размещение справа замедляет и усложняет чтение, а также искажает смысловую нагрузку выражения. Дело в том, что размещение этого оператора справа только для того, чтобы указать на наличие кода после него, необязательно, поскольку люди обычно считывают код блоками, а не построчно, как это делает компьютер.

Аналогично можно поступить с длинным перечнем параметров функции, т.е. размещая каждый параметр на новой строке. Рассмотрим показанный ниже пример такой функции.

```
void DrawBoxWithMessage(const AnsiString &Message, int...
```

Его можно переписать, как показано в листинге 3.2.

Листинг 3.2. Рекомендуемый способ записи длинного списка параметров функции

```
void DrawBoxWithMessage(const AnsiString &Message,
                        int Top,
                        int Left,
                        int Height,
                        int Width);
```

Здесь следует заметить, что запятую важно помещать в конце строки, так как расположение ее в начале следующей строки, не соответствующее правилам записи знаков пунктуации в обычном тексте, усложняет чтение. Здесь запятая используется только для того, чтобы сообщить компилятору о разделении параметров и не имеет другого смысла. Использование запятых не должно затруднять чтение кода. Обратите внимание на то, что расположение запятой резко отличается от расположения логических операторов, который рассматривались выше.

Такой же способ улучшения читабельности следует применять и для конкатенации длинных строк.

```
AnsiString FilePath = "";
AnsiString FileName = "TestFile";

FilePath = "C:\\RootDirectory"+"\\"+"Branch\\Leaf"+FileName+".txt";
```

Такой код рекомендуется записывать так, как показано в листинге 3.3.

Листинг 3.3. Рекомендуемая форма записи длинных конкатенаций

```
AnsiString FilePath = "";
AnsiString FileName = "TestFile";

FilePath = "C:\\RootDirectory"
           + "\\"
           + "Branch\\Leaf"
           + FileName
           + ".txt";
```

Код в листингах 3.2 и 3.3 предельно ясен, но его, возможно, не так просто вводить из-за большого количества отступов. Тем, кому кажется хлопотным введение множества пробелов (это справедливо в отношении большинства программистов!), можно предложить альтернативный подход на основе использования стандартного отступа для каждой строки выражения. Например, при использовании отступа с тремя пробелами, код в листинге 3.2 будет иметь следующий вид:

```
void DrawBoxWithMessage(const AnsiString &Message,
   int Top,
   int Left,
   int Height,
   int Width);
```

Это ухудшает читабельность, но позволяет упростить сопровождение. Такой выигрыш может показаться сомнительным, но иногда это бывает удобно.

Для экономии времени во время создания кода следует применять контекстные шаблоны кодирования, использовав комбинацию клавиш <Ctrl+J>, а затем выбрав нужный шаблон в списке. Однако при этом следует придерживаться единого стиля кодирования. Для этого список используемых шаблонов нужно настроить соответствующим образом с помощью вкладки Code Insight (Шаблоны кода) диалогового окна Editor Properties, которое можно отобразить на экране с помощью команды меню Tools⇨Editor Options. Обратите внимание, что символ | обозначает расположение курсора после вставки шаблона. Код шаблона можно также отредактировать вручную в файле `$(BCB)\Bin\Bcb.dci`, где `$(BCB)` — каталог, в котором инсталлирован пакет C++Builder.

Имена со смысловой нагрузкой

Одним из способов улучшения читабельности кода является использование осмысленных имен для переменных, типов и функций. *Имена типов* (*type name*) должны включать названия фактически используемых типов; например, `int` — это имя для значений целочисленного типа (integer), `TFont` — имя класса шрифтов библиотеки VCL. *Имена переменных* (*variable name*) — это названия объявленных переменных особого типа. Например, в приведенном ниже коде `NumberOfPages` — это имя переменной типа `int`.

```
int NumberOfPages;
```

Имена функций (*function name*) присваиваются функциям для описания их назначения. Рассмотрим сначала имена переменных, хотя большая часть сказанного в равной степени относится и к именам типов и функций.

Выбор имен переменных с указанием их назначения

Обычно имя переменной выбирается таким, чтобы оно отображало характер и назначение переменной. Если это возможно, то в имени переменной должен содержаться намек на тип переменной. Например, объявление переменной `String EmployeeName;` предпочтительнее, чем объявление переменной `String S;`.

Встретив переменную `S` в коде, довольно трудно будет вспомнить ее назначение. Что это такое: количество страниц в книге или строка с именем человека? Кроме того, довольно трудно вспомнить, какой тип имеет эта переменная: `int`, `double` или `String`? А имя переменной `EmployeeName` говорит само за себя: оно содержит имя сотрудника и, вероятно, имеет строковый тип.

Использование осмысленных имен не составляет никакого труда для разработчика и упрощает жизнь всем, кому впоследствии придется читать код. При каждом объявлении новой переменной, следует задать себе вопрос: “Для чего эта переменная?”. После сокращения до минимума ответа на этот вопрос можно получить осмысленное имя переменной.

В качестве имени переменной часто используется слово или короткая фраза. Для улучшения читабельности принято использовать прописную букву в начале каждого слова имени переменной. Хотя одни разработчики предпочитают использовать только строчные буквы с символами подчеркивания для отделения слов, а другие — некую смесь этих двух вариантов. Для иллюстрации этого рассмотрим следующий пример:

```
String EmployeeName;
String employee_name;
String Employee_Name;
String employeeName;
```

Недостатком использования символов подчеркивания является несколько б*о*льшая длина переменной. С технической точки зрения никаких возражений против этого способа нет, но при этом код перегружается, а имена переменных увеличиваются. Одни разработчики настаивают на том, чтобы все переменные начинались со строчной буквы, например `employeeName`. Эта запись обычно используется для разделения имен переменных и имен типов, которые предлагается начинать с прописной буквы. Другие разработчики используют начальную строчную букву только для имен временных переменных, например для переменной `temp` в функции `swap`. Компромиссное решение находится где-то посередине, т.е. имя переменной должно быть осмысленным и кратким. При этом следует учитывать, что осмысленность имени переменной имеет б*о*льшее значение.

Модификация имен переменных для указания типа

Знание назначения переменной часто имеет большее значение для понимания смысла кода, чем знание типа переменной. Однако иногда знание типа переменной требуется для того, чтобы применить особые правила. В листинге 3.4 приводится простой пример, иллюстрирующий эту ситуацию.

Листинг 3.4. Необходимость знания типа переменной

```
int Sum = 0;
int* Numbers = new int[20];

for(int i=0; i<20; ++i)
{
   Numbers[i] = i*i;
```

```
   Sum += Numbers[i];
}

double Average = Sum/20; // Переменная Sum имеет тип int,
                         // который нужно привести к типу double
```

Результатом выполнения кода в листинге 3.4 должно быть значение **123.5**, но на самом деле значение переменной Average будет равно 123. Дело в том, что Sum имеет тип int, и при делении на 20 (которое также рассматривается как значение типа int) будет получен результат типа int, который затем присваивается переменной Average.

Старайтесь объявлять переменные непосредственно перед их использованием. Благодаря этому переменные будут создаваться только по мере необходимости. Если код содержит условные выражения (или инициирует исключительную ситуацию), может случиться так, что некоторые объявленные переменные не использованы. В таком случае имеет смысл не увеличивать затраты, создавая и удаляя переменные, только когда это действительно необходимо. Будьте осторожны при размещении объявлений внутри циклов и прибегайте к этому только в случае крайней необходимости. При объявлении переменных рекомендуется сразу же их инициализировать. Не следует объявлять указатели и переменные в одной и той же строке. В лучшем случае это приведет к путанице.

Лучше всего объявлять переменные в таком виде:

```
Тип ИмяПеременной;
```

Тогда объявление указателя на int будет выглядеть так:

```
int* pointerToInt;
```

Однако в объявлении

```
int* pointerToInt, isThisAPointerToInt;
```

PointerToInt является указателем на int, а isThisAPointerToInt — не указателем на int, а переменной типа int. Схему этого объявления можно записать следующим образом:

```
int *pointerToInt, notPointerToInt;
```

Однако такое объявление также не очень удобно и отличается от рекомендованного выше формата Тип ИмяПеременной;. Поэтому для устранения двусмысленности его лучше переписать в отдельных строках в следующем виде:

```
int *pointerToInt = 0;

int notPointerToInt;
```

При этом следует явно инициализировать указатели либо значением NULL, либо адресом. Это позволяет предотвратить случайное использование указателей, содержащих неопределенный адрес.

Чтобы получить правильный результат **123.5** для значения переменной Average, в листинге 3.4 следует отредактировать последнюю строку, как показано ниже.

```
double Average = static_cast<double>(Sum)/20; // Правильный вариант
```

Переменная Sum приводится к типу double еще до выполнения операции деления на число 20 (которое теперь рассматривается как значение типа double). Теперь переменной Average присваивается значение **123.5** типа double.

Следите за корректным использованием четырех способов приведения типов в языке C++ и всегда используйте именно стиль языка C++, а не C. Стиль языка C++ более заметен среди другого кода и указывает на способ приведения.

В этом тривиальном примере явное указание типа для имени переменной напоминает применение кувалды для раскалывания орехов. Более эффективное решение можно получить, объявляя `Sum` как переменную типа `double`.

Иногда в имени переменной необходимо отразить информацию о ее типе. Одним из методов решения этой задачи является применение одной (или нескольких) букв в начале или в конце имени переменной. Например, букву `b` можно использовать для указания логического типа (bool), `s` — для строкового (string) и т.д. В таком случае переменные многих типов будут содержать избыточную информацию о типе переменных. При этом ударение будет делаться на типе информации, а не на назначении переменной. Кроме того, количество возможных типов может значительно превысить количество букв в алфавите. В результате использование подобных обозначений может привести к путанице и ненужному усложнению. Печально известный пример такого соглашения об именах называется венгерской нотацией, которая используется для кода с использованием интерфейса Win32 API.

На заметку

Венгерская нотация появилась при создании Win API-функций и основана на добавлении к именам переменных определенных символов для обозначения их типа. Так как для всех типов букв не хватает, то используются довольно странные сокращения. Например, логические переменные обозначаются префиксом `f`, строки — префиксом `sz`, указатели — префиксом `p` и т.д. Мы не будет приводить здесь их полный перечень. Достаточно сказать, что эта нотация создает больше проблем, чем решает. Они возникают в основном из-за имеющихся противоречий (например, переменная `wParam` является 32-битовым целым числом без знака, а не 16-битовым целым числом без знака, как можно предположить на основе ее префикса) и трудно читаемых имен. При кратком знакомстве со справочными файлами о Win32 API-фунцкиях можно найти множество других примеров использования этой нотации. Обозначения, отягощенные информацией о типе, крайне опасны, потому что ударение в них делается на типе переменной и часто мало что говорит о самой переменной.

При чтении такого кода не всегда легко мысленно разбить строку на составные части, что ухудшает восприятие кода. Если вы все же вынуждены использовать префиксную (или даже постфиксную) запись символов для обозначения типа, то компромиссное решение заключается в использовании дополнительных символов только для ограниченного круга особых типов.

Модификация имен переменных для обозначения характеристик или ограничений

В имени переменной может также отражаться информация о какой-либо характеристике или ограничении для этой переменной. Часто необходимо знать о том, что переменная является указателем, т.к. некорректная работа с указателями может стать причиной возникновения ошибки в программе. Например, если к указателю на переменную типа `int` случайно прибавить число 5 без предварительного разыменования, то это может привести к возникновению проблемы. Указатели рекомендуется использовать только в тех случаях, когда нельзя использовать никакой другой тип, например ссылку (для которой разыменование выполняется неявно).

Иногда важно знать, что переменная является статичной (`static`), если она — член класса или параметр функции. В таких случаях наиболее подходящими будут следующие префиксы: `p_` — для указателя, `s_` — для статичной переменной, `d_` или `m_` — для члена класса и `a_` —

для параметра функции. Символ `a_` используется для отделения параметра функции от указателя и его следует интерпретировать как "параметр `a_...` представляет аргумент (argument), переданный функции". Теперь смысл символа `a_` становится более понятным.

Для четкого отделения префикса от имени переменной часто используется символ подчеркивания. Это существенно упрощает восприятие префикса при чтении кода. Если символ подчеркивания используется вместе с информацией о некоторой характеристике переменной, то лучше использовать постфиксную запись. В таком случае имя переменной читается первым, что является более естественным. Например, при чтении имени переменной `s_NumberOfObjects` оно, вероятно, будет воспринято как "статичная переменная, которая содержит информацию о количестве объектов (the number of objects)". А при чтении имени переменной `NumberOfObjects_s` оно будет воспринято как "переменная с информацией о количестве объектов (the number of objects), которая является статичной". В таком случае ударение делается на назначении переменной (т.е. сохранении информации о количестве объектов). Каждый разработчик может выбрать предпочтительный для себя вариант. Проблема возникает при использовании обоих подходов, поскольку такой синтаксис не является аккуратным. Использование *префиксов* `p_` для указателей и *постфиксов* для других символов может разрешить большую часть проблем, но не все.

Другая распространенная проблема связана с указанием имен *констант*, которые часто обозначаются с помощью прописных букв.

При объявлении переменных некоторых классов часто требуется включить в них имя класса (без префикса, например T или C, если таковой присутствует) как часть имени переменной. Это обычно можно сделать аккуратно с помощью префикса, несущего определенную информацию, либо грубо в виде постфиксного номера, как это делается в IDE-среде в C++Builder для имен классов, хотя вообще для этого следует проявить большую изобретательность. Рассмотрим в качестве примера следующие объявления переменных.

```
TComboBox* CountryComboBox;
TLabel* NameLabel;
String BookTitle;
```

В имена переменных типа `TComboBox` и `TLabel` было бы разумно включить имя класса как часть имени переменной. Однако в случае переменной `BookTitle` совершенно очевидно, что это строка, а потому добавление слова `String` совсем необязательно и может принести больше вреда, чем пользы.

Особого упоминания заслуживает присвоение имен C++Builder для закрытых членов-переменных с помощью объявления `__property`. По соглашению такие переменные начинаются с префикса `F` (от Field). Поэтому рекомендуется следовать этому соглашению и не использовать прописную букву `F` в других целях. Основное назначение префикса в этом случае заключается в том, чтобы имя такого свойства использовалось неизменным.

Выбор имен типов

Как уже говорилось выше, выбирать имена типов следует так же, как имена для переменных. Однако при этом следует учитывать определенные соглашения.

Для классов такое соглашение означает, что если класс является производным от класса `TObject`, то имя класса должно начинаться с префикса `T`. Это указывает пользователю класса на то, что данный класс создан в стиле классов VCL-библиотеки и удовлетворяет соглашению об именах для других классов библиотеки VCL. Эта особенность имеет побочный благоприятный эффект. Для переменных этого класса имя класса может использоваться без префикса, что делает очевидным их назначение. В классах, отличных от VCL-классов (т.е., в *обычных* классах C++), также могут использоваться префиксы, но для этого лучше исполь-

зовать не префикс T, а, например, префикс C для указания того, что используется обычная объектная модель C++ и класс не является производным от класса TObject. Это отличие может иметь очень большое значение: VCL-классы должны создаваться динамически; а на другие классы, отличные от VCL-классов, такое ограничение не накладывается.

Имена для таких типов, как enum, Set, struct и union, можно задавать, придерживаясь тех же правил. Согласно соглашению об именах в C++Builder, префикс T используется для имен перечислимых типов и типов набора, хотя совсем необязательно придерживаться этого правила, которое не имеет какого-то особого значения. Не рекомендуется использовать в качестве префикса символ E, так как в C++Builder он используется для классов исключительных ситуаций.

Так как типы enum и Set широко используются в C++Builder, то стоит упомянуть некоторые особенности их именования. Класс Set — это шаблонный класс, объявленный в файле $(BCB)\Include\VCL\sysset.h, как показано ниже.

```
template<class T, unsigned char minEl, unsigned char maxEl>
   class RTL_DELPHIRETURN Set;
```

Игнорируя метку RTL_DELPHIRETURN, которая используется для совместимости с библиотекой VCL, нетрудно заметить, что шаблон Set принимает три параметра: тип и два ограничения диапазона. Следовательно, класс Set можно объявить таким способом:

```
Set<char, 'A', 'Z'> CapitalLetterMask;
```

Если класс Set используется несколько раз, то для упрощения его представления обычно используется ключевое слово typedef, как показано ниже.

```
typedef Set<char, 'A', 'Z'> TCapitalLetterMask;
// далее в коде
TCapitalLetterMask CapitalLetterMask;
```

Тип Set часто используется для создания масок (как показано в этом примере), поэтому слово Mask было использовано в именах приведенного выше примера. Тип enum обычно используется для реализации содержимого класса Set. Например, в файле $(BCB)\Include\VCL\graphics.hpp можно найти показанные ниже определения.

```
enum TFontStyle {  fsBold, fsItalic, fsUnderline, fsStrikeOut } ;
typedef Set<TFontStyle, fsBold, fsStrikeOut> TFontStyles;
```

Типы перечисления в библиотеке VCL используются по большей части для организации работы с типами множеств. Для удобства они объявляются в области действия целого файла. Включая файл в проект, можно легко получить доступ к типу Set. Это означает высокую вероятность возникновения конфликта имен. Для исключения таких конфликтных ситуаций "инициалы" имени перечислимого типа используются в качестве префикса для всех переменных этого типа. Это позволяет свести к минимуму возможность возникновения конфликта имен. Этот очень удачный прием рекомендуется использовать при создании собственного кода. Например, инициалами типа TFontStyle (за исключением символа T) являются символы fs, которые образуют префикс для каждой переменной этого перечислимого типа. Другим методом исключения конфликта имен является размещение объявлений перечислимых типов и объявлений типов с помощью typedef внутри определений классов, которые их используют. Это значит, что при вызове этих переменных за пределами класса потребуется уточнить имя класса. Эти замечания в равной степени касаются и констант.

Использование ключевого слова typedef иногда позволяет улучшить читабельность кода; например, в приведенном выше примере, а также при объявлении указателей функции (и особенно для событий __closures, которые более подробно рассматриваются в главе 9, посвя-

щенной созданию настраиваемых компонентов). За исключением этих ситуаций, ключевое слово `typedef` следует использовать с большой осторожностью, так как оно способствует скрытию типа переменной. Вследствие этого чрезмерное использование ключевого слова `typedef` может привести к большой путанице.

Выбор имен функций

Чтобы адекватно отражать назначение функций, их имена должны быть точными и достаточно информативными. Если невозможно точно дать такое описание, то должна быть изменена сама функция. При этом разные типы функций должны иметь разные имена.

Функция, которая не возвращает никакого значения (функция типа `void`) или возвращает только информацию об успешном завершении, т.е. функция, которая не возвращает никаких данных, должна содержать в имени *глагол выполняемого действия* и *имя объекта действия*: `CreateForm()`, `DisplayBitmap()` и `OpenCommPort()`. Функция-член такого типа часто не требует указания объекта, потому что эту роль может, как правило, выполнить сам объект, вызывающий эту функцию.

Если функция возвращает какие-то данные, то в ее имени должен содержаться намек на природу возвращаемого значения: `GetAverage()`, `log10()` и `ReadIntervalTimeOut()`.

Некоторые разработчики предпочитают использовать строчную букву для первого слова в имени функции. Это зависит от их личных предпочтений, но и в этом случае следует придерживаться единого стиля.

Вообще имена функций могут быть гораздо длиннее имен переменных, а потому по этому поводу не следует чрезмерно беспокоиться. Однако длинное имя функции не должно быть результатом плохо спроектированной функции, которая пытается выполнить много слабо связанных операций. Постарайтесь создать функцию, выполняющую одну хорошо определенную задачу, которая адекватно отражена в имени функции. Если ваша функция должна выполнять несколько задач, то ее имя должно отражать эту особенность.

Соглашения об именах

Способы именования переменных, типов и функций — очень обширная тема, которая охватывает мириады понятий. Поэтому все их описать просто невозможно. В любом случае рекомендуется придерживаться следующих правил.

- При чтении имен переменной или функции должно становиться очевидным ее назначение. Если для этого потребуется увеличить длину имени, это необходимо сделать.
- При чтении имени типа должно быть очевидно его предполагаемое использование. Постарайтесь не использовать имен, которые могут использоваться переменными этого типа. Для имени типа лучше использовать префиксную букву.
- Будьте последовательны! Из опыта известно, что даже самое сложное соглашение об именах становится понятным, если оно последовательно применяется.

Конструкции кода

Одним из наилучших способов улучшения читабельности кода является соответствующее использование конструкций кода, т.е. использование соответствующего инструмента в соответствующее время для соответствующей работы. Следовательно, для этого нужно иметь правильное представление о том, когда использовать константы (`const`), когда применять ссылки вместо указателей, когда уместно использовать циклы, следует ли применить несколько выражений `if-else` или достаточно одного выражения `switch`, когда поместить объ-

явление внутри, а когда вне класса, когда передать управление блоку обработки исключительной ситуации, как описать спецификацию исключительной ситуации и т.д. Большая часть этих вопросов относится к этапу проектирования приложений, но некоторые из них достаточно просты, чтобы их можно было учесть при создании кода, улучшив не только его читабельность, но и надежность. Использование ключевого слова `const` и ссылок подробно рассматривается в следующем разделе. Это именно та область программирования, которую можно совершенствовать постепенно.

Итак, если при создании кода вы четко представляете себе цель и способы ее достижения, то созданный вами код будет понятным. В этом случае человек, читающий код, найдет в нем именно то, что предполагает найти, и никакой путаницы не возникнет.

Комментарии

Основное назначение комментариев — краткое описание кода для улучшения его читабельности. Эту цель легко достичь при благоразумном использовании комментариев, но следует помнить о том, что размещение комментариев без четкой стратегии затрудняет восприятие кода.

Комментарии размещают в создаваемом коде для отметки особых мест, в которых может возникнуть путаница. При размещении комментариев следует использовать только комментарии в стиле языка C++, которые начинаются с символов `//`. Это предотвратит возможный конфликт при неожиданном появлении комментария в стиле языка C на основе символов `*/`.

Комментарии для документирования кода

Способ размещения комментариев в большой степени сказывается на восприятии кода. Читабельность кода можно существенно улучшить, если следовать приведенным ниже рекомендациям.

При размещении комментариев важно не разрывать код. Комментарии должны быть отделены от кода пустой строкой, другим стилем (например, наклонным начертанием) или цветом. Комментарии для кода с отступом также следует располагать с отступом. Это особенно важно, если в комментариях рассматривается функциональная часть кода. Комментарии, созданные таким образом, могут читаться независимо от кода, а код может восприниматься независимо от комментариев. Некоторые программисты предпочитают не отделять строку с комментарием от строки с кодом, так как считают, что использования цвета и начертания достаточно для выделения комментариев.

Изменяя параметры во вкладке Colors диалогового окна Editor Properties (которое можно отобразить на экране с помощью команды меню Tools⇨Editor Options), можно скрывать или выделять комментарии. Для скрытия комментария в окне редактора кода нужно изменить цвет комментария, чтобы он слился с фоновым цветом окна редактора кода; по умолчанию это белый цвет. Для выделения комментария измените фоновый цвет на синий, а цвет текста — на белый. Поэкспериментируйте и найдите наиболее подходящий для себя вариант.

Если комментарий относится к строке, в которой есть место для комментария, то его рекомендуется располагать в одной строке с кодом. Если необходимо создать комментарий для нескольких строк, то их необходимо начинать с тех же столбцов. Преимущество таких комментариев заключается в том, что они не будут сливаться с кодом. Ниже приводится пример кода с такими комментариями.

```
double GetMaximumValue(const std::vector<double>& Vector)
throw(std::out_of_range)
{
```

```
   // Инициализация переменной Maximum; если вектор пуст, то
   // будет начата обработка
   // исключительной ситуации std::out_of_range

   double Maximum = Vector.at(0);

   for(int i=0; i<Vector.size(); ++i)
   {
      if(Vector[i] > Maximum)  // Если i-ый элемент больше, чем
      {                        // текущее значение переменной
         Maximum = Vector[i];  // Maximum, то для Maximum
      }                        // задать значение i-го элемента
   }

   return Maximum;
}
```

Комментарии для этого выражения на основе оператора `if` излишни — они приведены здесь лишь для иллюстрации способа расположения комментариев. Комментарии также могут располагаться после замыкающей скобки цикла или выражения с условиями выбора, как показано ниже.

```
}// конец if
}// конец for
}// ! for
}// конец for(int i=0; i<Vector.size(); ++i)
```

Последний пример комментария кажется чрезмерно длинным, но иногда он бывает необходим при создании кода или в том случае, когда цикл или выражение на основе условий выбора может занимать несколько страниц кода.

Комментарии необходимы для документирования кода. Они особенно удобны для подытоживания важной информации про характеристики функции. Для представления такой информации разработано несколько способов, но всегда следует придерживаться какого-то одного метода. Если требуется указать важную информацию о функции, то эта информация должна быть представлена в заголовочном файле, в котором эта функция объявлена, а также в файле, в котором она реализована. Кто бы ни занимался сопровождением кода, он должен обладать той же информацией о нем, что и создатель кода. Перед реализацией и объявлением функции всегда следует иметь краткое описание назначения функции. Кроме того, рекомендуется перечислять требования, которым должна удовлетворять функция, чтобы она работала должным образом (эти требования иногда называются *предусловиями*), а также преимущества, которые она несет пользователю (они иногда называются *постусловиями*). Условие, которое приводит к неопределенному поведению, должно быть явно указано и включено в список требований. Представленная таким образом информация является своего рода "контрактом" с функцией: если все реализовано в соответствии с описанием, то функция выполнит все должным образом.

При этом документировать нужно только необходимую и достаточную информацию. Короче говоря, простые функции можно совсем не документировать. И наоборот, для больших и сложных функций может потребоваться создание достаточно длинных комментариев. Если это так, то учтите следующее. Пользователям функции необязательно знать о деталях внутренней реализации функции или как функция действует на закрытые или защищенные данные (если функция является членом класса). Такую информацию не нужно размещать в заголовочном файле. Также следует помнить, что комментарии наиболее эффективны при максимально близком расположении возле комментируемого места. При этом случается, что какая-то информация чересчур подробна, а иную лучше бы разместить в другом месте. Ниже приводится пример комментария функции в файле, где она реализована.

```
//-------------------------------------------------------------//
//
// НАЗНАЧЕНИЕ : Возвращает максимальное значение для
//              вектора чисел типа double
//
// ТРЕБОВАНИЯ : Получаемый вектор не должен быть пустым
//
// ПРЕИМУЩЕСТВА: Функция возвратит значение наибольшего элемента
//
//-------------------------------------------------------------//

double GetMaximumValue(const std::vector<double>& Vector)
throw(std::out_of_range)
{
   double Maximum = Vector.at(0);

   for(int i=0; i<Vector.size(); ++i)
   {
      if(Vector[i] > Maximum)
      {
         Maximum = Vector[i];
      }
   }

   return Maximum;
}
```

Аналогично, заголовочный файл будет выглядеть, как показано ниже.

```
double GetMaximumValue(const std::vector<double>& Vector)
hrow(std::out_of_range)
    // НАЗНАЧЕНИЕ : Возвращает максимальное значение для
    //              вектора чисел типа double
    //
    // ТРЕБОВАНИЯ : Получаемый вектор не должен быть пустым
    //
    // ПРЕИМУЩЕСТВА: Функция возвратит максимальное значение
```

Напомним, что заголовочные файлы программы часто содержат только текущую информацию об интерфейсе, а потому всегда должны содержать четкую и точную информацию (на текущий момент). В противном случае комментарии могут вскоре стать бесполезными. При изменении кода следует соответствующим образом изменять связанные с ним комментарии.

Если для какой-то части кода приходится писать обширные комментарии, то это значит, что сам код следует изменить.

Комментарии для игнорирования кода

Еще одним вариантом использования комментариев в C++Builder является исключение с их помощью имен неиспользуемых параметров в обработчиках событий, генерируемых IDE-средой. Часто бывает так, что некоторые или даже все параметры не используются функцией. Комментируя имена параметров, программист не изменяет сигнатуру функции, а лишь явно указывает на фактически используемые параметры. Если параметры записаны в одной строке, то с этой целью следует использовать комментарии в стиле языка C.

В конце концов список параметров можно переупорядочить и использовать комментарии в стиле языка C++, но полученная в итоге запись может быть не совсем аккуратной. В таких случаях необходимо проявлять осторожность. Ниже приводится фрагмент кода для обработчика события расположения мыши MouseUp над кнопкой TButton, который иллюстрирует это наблюдение.

```
void __fastcall
   TMainForm::Button1MouseUp(TObject* /*Sender*/,
                             TMouseButton /*Button*/,
                             TShiftState /*Shift*/,
                             int X,
                             int Y)
{
   // Показать координаты указателя мыши на кнопке Button1
   Label1->Caption.sprintf("%d,%d", X, Y);
}
```

Обработчик события MouseUp использует член-функцию sprintf() типа AnsiString для изменения свойства Caption ярлыка Label1. Этот прием сработает, потому что член-функция sprintf() возвращает по ссылке значение AnsiString(*this). Однако, в общем случае это свойство следует изменять, только присвоив ему новое значение с помощью оператора присваивания.

Здесь не стоит удалять неиспользуемые имена параметров, так как это может внести путаницу в заголовок функции, особенно если впоследствии все-таки придется использовать какой-то такой параметр. Очевидное решение заключается в комментировании неиспользуемых имен параметров и расположении закомментированной запятой и замыкающей скобки *перед* символами комментария.

```
void
   __fastcall TMainForm::Button1MouseUp(TObject* , //Sender,
                                        TMouseButton , //Button,
                                        TShiftState , //Shift,
                                        int X,
                                        int Y)
{
   // Показать координаты указателя мыши на кнопке Button1
   Label1->Caption.sprintf("%d,%d", X, Y);
}
```

Это достаточно эффективный способ, который не усложняет сопровождение кода. Если впоследствии потребуется использовать этот параметр, то вы просто удалите символы // и дополнительные запятые и скобку. Альтернативный вариант этой операции выглядит так:

```
void __fastcall TMainForm::Button1MouseUp(TObject* ,//Sender
                                          TMouseButton ,//Button
                                          TShiftState ,//Shift
                                          int X,
                                          int Y)
{
   // Показать координаты указателя мыши на кнопке Button1
   Label1->Caption.sprintf("%d,%d", X, Y);
}
```

Он отличается от предыдущего только удалением закомментированных запятых в конце каждого комментария. Это можно считать улучшением, поскольку запятая (или замыкающая скобка) в конце комментария может ввести в заблуждение.

Комментарии для улучшения внешнего вида кода

Еще один вариант использования комментариев заключается в улучшении внешнего вида кода на экране или для печати. Это достигается благодаря размещению прямоугольников вокруг заголовков, разделительных линий между функциями и т.д. При использовании комментариев таким образом следует позаботиться о том, чтобы код не потерялся среди большого количества звездочек (*) или других декоративных символов. При создании такого оформления также следует придерживаться единого стиля.

C++Builder автоматически помещает разделительную линию между генерированными им блоками кода. Это улучшает внешний вид кода, т.к. вносится дополнительный элемент визуального разделения. То же самое рекомендуется делать при создании собственного кода.

В C++Builder 5 можно изменить предлагаемую по умолчанию разделительную линию, которая располагается между разделами генерируемого кода. Это можно сделать, отредактировав показанную ниже строку в файле BCB.BCF в каталоге $(BCB)\BIN (если такой строки нет, ее нужно создать).

```
Formatting                                                              ]
Divider Line=//---- My Custom Divider Line ----//
```

Как видите, ее формат аналогичен формату конфигурационного файла с расширением *.ini. Разделительная линия должна начинаться как строка с комментарием, т.е. с символов //. Рекомендуемый метод добавления разделительной линии заключается в том, чтобы сначала создать ее в окне редактора кода, затем вырезать и вставить файл BCB.BCF. В таком случае можно четко представить себе ее внешний вид. Это позволяет добиться единого стиля в оформлении кода, генерированного IDE-средой, и кода, созданного вручную. Также рекомендуется добавить эту разделительную линию в список шаблонов кода.

```
[Code
```

Заключительные замечания по поводу улучшения читабельности кода

В конечном итоге самым лучшим описанием назначения и принципов работы кода является сам код. Улучшая читабельность кода, программист может таким образом улучшить его описание.

Усовершенствованные приемы программирования в C++Builder

В этом разделе рассматриваются некоторые способы более эффективного программирования на языке C++ с помощью C++Builder. Описанию эффективного программирования на языке C++ посвящено огромное количество книг, с которыми читатель обязательно должен познакомиться для углубления своих знаний языка C++. В конце этой главы приводится список такой рекомендованной литературы. Здесь рассмотрены темы, имеющие непосредственное отношение к C++Builder, а также те, которые часто неправильно интерпретируются или неправильно используются неопытными пользователями C++Builder.

Применение класса String вместо типа char*

При работе со строками не стоит использовать тип char*. Для этого рекомендуется использовать либо строковый класс из стандартной библиотеки C++, либо собственный строковый класс библиотеки VCL AnsiString (который переименован в String), либо оба этих класса. Доступ к строковому классу из стандартной библиотеки C++ можно получить, поместив приведенную ниже строку в начале кода.

```
#include <string>
```

Именно этот класс следует использовать для создания переносимых приложений. В противном случае рекомендуется использовать класс AnsiString, который предпочтительнее потому, что именно с его помощью представлены строки во всей библиотеке VCL. В этом случае созданный код будет беспрепятственно взаимодействовать с библиотекой VCL. Для более близкого знакомства с классом AnsiString следует внимательно изучить его методы. Так как строки используются достаточно часто, то затраты на обучение быстро окупятся за счет упрощения и повышения эффективности кода.

В тех случаях, когда требуется использовать строку типа char*, например для передачи параметра функции Win32 API-интерфейса, в обоих строковых классах предусмотрен метод c_str(), который возвращает такую строку. Кроме того, в классе AnsiString в виде методов также сохранены такие популярные и старомодные функции, как sprintf() и printf() (для конкатенации строк). Они предлагаются в двух версиях: стандартной и версии с префиксом cat_. Эти версии отличаются тем, что функции с префиксом cat_ добавляют конкатенируемую строку к существующей строке AnsiString, а стандартная функция заменяет содержимое существующей строки AnsiString. Различие между методами sprintf() и printf() заключается в том, что функция sprintf() возвращает ссылку на AnsiString, а функция printf() — длину окончательно сформатированной строки (или длину конкатенированной строки в случае функции cat_printf). Ниже приведены примеры объявления таких функций.

```
int __cdecl printf(const char* format, ...);
int __cdecl cat_printf(const char* format, ...);
AnsiString& __cdecl sprintf(const char* format, ...);
AnsiString& __cdecl cat_sprintf(const char* format, ...);
```

Все эти методы в конечном итоге содержат вызовы функций vprintf() и cat_vprintf(). В качестве второго параметра они принимают список va_list, а не список аргументов-переменных. Для этого в код потребуется добавить строку с выражением #include <stdarg.h>. Объявления этих функций имеют следующий вид:

```
int __cdecl vprintf(const char* format, va_list paramList);
int __cdecl cat_vprintf(const char* format, va_list paramList);
```

Соответствующие им функции printf() и sprintf()выполняют ту же задачу, отличаясь только возвращаемым результатом. И именно это является единственным критерием выбора одной из двух функций.

Обратите внимание на то, что функции-члены printf() и sprintf() типа AnsiString в C++Builder версии 4 идентичны функциям cat_printf() и cat_sprintf() в версии 5, *а не* членам-функциям printf() и sprintf() типа AnsiString. При преобразовании кода для указанных версий на это следует обратить особое внимание.

Ссылки и их эффективное использование

Ссылки часто неверно воспринимаются, а потому применяются не так, как должно. Указатели часто можно заменить ссылками, что позволяет сделать код более интуитивно понятным и простым для сопровождения. В этом разделе рассматриваются основные характеристики ссылок и приемы эффективной работы с ними. Кроме того, в ней обсуждаются причины, по которым в библиотеке VCL указатели не используются в качестве параметров.

Ссылка всегда относится только к одному *объекту ссылки*, на который она ссылается; она не может быть перенаправлена для ссылки на другой объект (под "объектом" в данном контексте подразумеваются все типы). Ссылка должна быть инициализирована при создании, к тому же она не может не иметь объекта ссылки (`NULL`). Указатели же могут указывать на неопределенный объект (`NULL`), могут быть перенаправлены и не должны инициализироваться при создании. Ссылку следует рассматривать как альтернативное имя объекта, тогда как указатель следует рассматривать как объект в себе. Все действия, которые выполняются со ссылкой, также выполняются с объектом, на который она ссылается, и наоборот. Дело в том, что ссылка — это всего лишь альтернативное имя объекта ссылки, т.е. они представляют собой одно и то же. Следовательно, ссылки, в отличие от указателей, разыменованы неявным образом.

Ниже приводится пример объявления ссылки.

```
int X = 12; // Объявление переменной int X
            // и инициализация значением 12

int& Y = X; // Объявление Y как ссылки на число типа int
            // и инициализация ее ссылкой на X
```

При изменении значения переменной `Y` или `X`, будет также изменено значение переменной `X` или `Y`, соответственно, потому что `X` и `Y` — это два имени одного и того же объекта. Ниже показан еще один пример объявления ссылки на динамически создаваемую переменную.

```
TBook* Book1 = new TBook(); // Объявление и создание
                            // объекта TBook

TBook& Book2 = *Book1;      // Объявление переменной Book2
                            // как ссылки на переменную типа
                            // TBook и ее инициализация ссылкой
                            // на объект с указателем Book1
```

Объект, заданный указателем `Book1`, является объектом ссылки `Book2`.

Ссылки в основном используются для передачи определенных пользователем типов в качестве параметров функции. Рассмотрим, например, функцию `swap` с двумя параметрами типа `int`.

```
void swap(int& X, int& Y)
{
   int temp;
   temp = X;
   X = Y;
   Y = temp;
}
```

Эта функция может быть вызвана таким образом:

```
int Number1 = 12;
int Number2 = 68;
```

```
Swap(Number1, Number2);

// Number1 == 68 и Number2 == 12
```

Значения переменных `Number1` и `Number2` передаются функции `swap` по ссылке, а потому `X` и `Y` внутри этой функции становятся альтернативными именами переменных `Number1` и `Number2`, соответственно. Все, что происходит с переменной `X`, происходит и с переменной `Number1`, а то. что происходит с переменной `Y`, происходит с переменной `Number2`. Такой преопределенный тип, как `int`, следует передавать по ссылке только в том случае, если предполагается изменить ее значение. В противном случае эффективнее было бы передавать ее по значению. Однако с определенными пользователем типами (классами, структурами и т.д.) следует поступать иначе. Вместо передачи функциям переменных такого типа по значению, эффективнее было бы передавать их по ссылке с указанием ключевого слова `const`. А если тип передаваемой переменной предполагается изменить, то следует передавать ее по ссылке без указания ключевого слова `const` или с помощью указателя. Рассмотрим следующий пример:

```
void DisplayMessage(const AnsiString& message)
{
   // Отображение сообщения.
   // message - это псевдоним для аргумента типа AnsiString,
   // передаваемого функции. При этом копирование значений
   // не выполняется, а ключевое слово const указывает на то,
   // что эта функция не сможет изменить передаваемое сообщение.
}
```

Приведенная выше форма записи этой функции гораздо лучше, чем показанная ниже.

```
void DisplayMessage(AnsiString message)
{
   // Отображение сообщения.
   // message - это копия аргумента типа AnsiString,
   // передаваемого функции.
}
```

Первый вариант предпочтительнее по двум причинам. Во-первых, параметр типа `AnsiString` передается по ссылке. Это значит, что при вызове функции в качестве вызывающего аргумента используется параметр типа `AnsiString`, так как этой функцией используется только ссылка. При этом не потребуется вызывать конструктор копирования для параметра типа `AnsiString` (как это происходит при вызове второго варианта функции), как не потребуется вызывать деструктор (что обычно делается при вызове второго варианта функции после выхода за пределы области действия). Во-вторых, ключевое слово `const` используется в первом варианте функции для указания того, что сообщение нельзя изменить. При этом вызовы обоих вариантов функции будут выглядеть одинаково.

```
AnsiString Message = "Hello!";

DisplayMessage(Message);
```

Однако в этом случае рекомендуется использовать первый вариант функции, поскольку он более безопасен и быстр. Обратите внимание на то, что при этом не придется вносить изменений в код вызова.

Функции также могут возвращать ссылки, обладающие таким побочным эффектом, как возврат lvalue-значения (т.е. значения, которое может появится в левой части выражения). Кроме того, в этом случае можно создавать операторы, располагая их в левой части выражения, например оператор, возвращающий элемент массива по индексу. Рассмотрим следующий пример классов Book и ArrayOfBooks:

```
class Book
{
   public:
          Book();
          int NumberOfPages;
} ;

class ArrayOfBooks
{
   private:
          static const unsigned NumberOfBooks = 100;
   public:
          Book&  operator[] (unsigned i);
} ;
```

В этом случае экземпляр класса ArrayOfBooks может использоваться как обычный массив. Тогда, используя оператор, возвращающий элемент массива по индексу, можно присваивать и считывать значения его элементов, как показано ниже.

```
ArrayOfBooks ShelfOfBooks;
unsigned PageCount = 0;

ShelfOfBooks[0].NumberOfPages = 45;          // Маленькая книга!
PageCount += ShelfOfBooks[0].NumberOfPages; //PageCount = 45
```

Это возможно, потому что возвращаемое оператором значение фактически является ссылкой, а не копией объекта, на который делается ссылка.

Обычно, рекомендуется использовать ссылки, а не указатели, потому что они обеспечивают большую безопасность: их нельзя перенаправить, для них не требуется выполнять проверку на наличие неопределенного значения (NULL), т.к. они должны ссылаться на вполне определенный объект. Кроме того, для них не требуется выполнять явное разыменование, что позволяет создавать интуитивно понятный код.

Почему в таком случае указатели активно используются в библиотеке VCL в C++Builder? Причина заключается в том, что библиотека VCL создана с помощью Object Pascal, где используются ссылки Object Pascal. Ссылка Object Pascal ближе к указателю C++, чем к ссылке C++. Побочный эффект такой связи состоит в том, что при использовании библиотеки VCL вместе с C++, указатели должны применяться как замена для ссылок Object Pascal. Дело в том, что ссылка Object Pascal (в отличие от ссылки C++) может ссылаться на неопределенное значение (NULL) и ее можно перенаправить на другой объект. В некоторых случаях параметры-ссылки можно использовать вместо параметров-указателей. Но так как все объекты на основе библиотеки VCL создаются динамически в свободной области памяти и потому указываются с помощью указателей, то сначала должны быть разыменованы эти указатели. Так как при создании библиотеки VCL были учтены особенности ссылок Object Pascal, для передачи и возврата параметров объекта применяются указатели. Напомним, что параметр-указатель передается по значению, а потому переданный указатель не будет изменен функцией. Предотвратить изменение указанного объекта можно с помощью ключевого слова const.

Избегайте применения глобальных переменных

Глобальные переменные рекомендуется использовать только в случаях крайней необходимости. Помимо "загрязнения" глобального пространства имен (и повышения вероятности возникновения конфликта имен), это также повышает вероятность появления зависимости между единицами трансляции, в которых используются глобальные переменные. Такой код усложняет сопровождение и использование единиц трансляции в других программах. Объявление переменных в разных местах усложняет восприятие и понимание кода в целом.

Опытный и проницательный программист C++Builder среди прочего сразу же сможет заметить, что в каждом модуле формы присутствуют глобальные указатели формы. Это создает впечатление, что применение глобальных переменных вполне допустимо, поскольку они применяются в C++Builder. Однако C++Builder использует их по особой причине, которая описывается в конце этого раздела. Здесь же мы рассмотрим некоторые альтернативные варианты объявления глобальных переменных.

Допустим, что глобальные переменные жизненно необходимы. Как же их использовать, избежав свойственных им побочных эффектов? В таком случае нужно применить нечто, что ведет себя как глобальная переменная, но таковой не является. Для этого следует использовать класс с методом, который возвращает значение ссылки на соответствующую статическую переменную, которая представляет нашу глобальную переменную. В зависимости от назначения глобальных переменных (например, глобальной для программы или для библиотеки), может потребоваться (или не потребоваться) получить доступ к переменным с помощью статических методов. Иначе говоря, можно в случае необходимости инициализировать объект класса, который содержит статические переменные. Рассмотрим сначала пример, когда действительно необходимо получить доступ к статическим переменным (представляющим наши глобальные переменные) посредством статических методов. Такой вариант класса впредь будем называть модулем.

Для улучшения представления глобальных переменных поместим их в класс как закрытые статические переменные, а для доступа к ним используем статические функции получения значения (getter) и передачи значения (setter). (Более подробную информацию по этому поводу можно найти в книге Лакоса (Lakos) *Large-Scale C++ Software Design*, 1996, p. 69). Это позволяет предотвратить "загрязнение" глобального пространства имен и в определенной степени получить управление над способом доступа к глобальным переменным. Обычно такой класс называют `Global`. Таким образом, две глобальные переменные

```
int Number;
double Average;
```

можно заменить приведенным ниже классом `Global`.

```
class Global
{
 private:
   static int Number;
   static double Average;

   //ЗАКРЫТЫЙ КОНСТРУКТОР
   Global(); //не реализован, инициализация невозможна

  public:
   // Методы, присваивающие значения - SETTERS
   static void setNumber(int NewNumber) {  Number = NewNumber; }
   static void setAverage(double NewAverage) {  Average = NewAverage; }
```

```
   // Методы, извлекающие значения - GETTERS
   static int getNumber() {  return Number; }
   static double getAverage() {  return Average; }
} ;
```

Доступ к переменной `Number` теперь можно получить, вызвав `Global::getNumber()` и `Global::setNumber()`. Аналогичным образом организован доступ к переменной `Average`. Класс `Global` на самом деле является модулем, доступ к которому можно получить в любом месте программы и для которого не требуется инициализация (потому что члены-данные и методы являются статическими).

Очень часто такая реализация не требуется, а потому можно создать класс с глобальной точкой доступа, который конструируется только при первом доступе к нему. Этот способ обладает преимуществом по сравнению с другим способом, потому что позволяет управлять порядком инициализации переменных (объекты должны быть сконструированы до их первого использования). Используемый для этого метод заключается в размещении нужных переменных в классе, который не может быть непосредственно инициализирован, а доступ к ним может быть осуществлен только с помощью статического метода, который возвращает ссылку на класс. При этом гарантируется, что класс, содержащий переменные конструируется только один раз при его первом использовании.

Этот подход часто называют *одноэлементным шаблоном* (*Singleton pattern*) (более подробное его описание можно найти в книге *Design Patterns: Elements of Reusable Object-Orientated Software*, Gamma et al., 1995, p. 127). Шаблон — это способ представления периодически повторяющихся проблем и их решений в объектно-ориентированных программах. Более подробную информацию о шаблонах можно найти в главе 4 об усовершенствованных приемах программирования в C++Builder.

Основной код, необходимый для создания класса `Singleton` (именно так он чаще всего называется), выглядит следующим образом.

```
class Singleton
{
    public:
           static Singleton& Instance();

    protected:
           Singleton(); // Не реализован,
                        // инициализация не возможна
} ;
```

Реализация экземпляра `Instance` класса `Singleton` выглядит так, как показано ниже.

```
Singleton& Singleton::Instance()
{
   static Singleton* NewSingleton = new Singleton();
   return *NewSingleton;
}
```

При первом вызове экземпляра `Instance` будет создан новый класс `Singleton` и возвращена ссылка на него. Последующие вызовы класса `Singleton` будут просто возвращать эту ссылку. Однако деструктор класса `Singleton` не будет вызван, поэтому объект просто останется в свободной области памяти. Если деструктор должен выполнить обработку данных, то можно применить показанный ниже способ, чтобы убедиться в том, что объект `Singleton` удален.

```
Singleton& Singleton::Instance()
{
   static Singleton NewSingleton;
   return NewSingleton;
}
```

Но этот способ также имеет недостатки, т.к. другой статический объект может предпринять попытку доступа к `Singleton` после его удаления. Одно из решений этой проблемы заключается в применении *метода тонкого учета* (*nifty counter technique*) (более подробную информацию об этом можно найти в книгах *C++ FAQ*, Second Edition, Cline et al., 1999, p. 235, и *Large-Scale C++ Software Design*, Lakos, 1996, p. 537). В этом методе статический счетчик применяется для учета событий создания и удаления объектов. Если возникнет необходимость воспользоваться этим методом, то следует продумать альтернативные варианты изменения кода. Возможна ситуация, когда незначительное перепроектирование кода поможет исключить такую зависимость.

Теперь читателю должно быть ясно, что статические переменные подобны глобальным и почти всегда могут быть использованы вместо них. Хотя тут снова следует напомнить, что по возможности следует избегать применения глобальных переменных.

Использование в C++Builder глобальных переменных

Почему же все-таки в C++Builder используются глобальные указатели форм? Их применение дает возможность использовать немодальные формы. В течение всего времени своего существования такая форма должна иметь глобальную точку доступа. IDE-среде удобно автоматически создать такую точку после создания формы. По умолчанию IDE-среда добавляет вновь созданные формы в автоматически создаваемый список. При этом в функцию `WinMain` файла проекта (с расширением `.cpp`) добавляется следующая строка (здесь *X* является числом):

```
Application->CreateForm(__classid(TFormX), &FormX);
```

Для модальных форм это не требуется, потому что метод `ShowModal()` возвращается после закрытия формы, что позволяет удалить ее в той же области действия, в которой она была создана. По этой причине следует привести некоторые общие рекомендации в отношении использования форм.

Для изменения поведения IDE-среды таким образом, чтобы формы не добавлялись автоматически в список Auto-create forms вкладки Forms диалогового окна Project Options, выберите команду меню Tools⇨Environment Options и установите флажок параметра Auto create forms в группе Forms designer вкладки Preferences диалогового окна Environment Options. Формы будут добавляться в список Available forms вкладки Forms диалогового окна Project Options.

Во-первых, следует решить, какой будет форма: модальной или немодальной.

Если форма является модальной, то ее можно создать и удалить в той же области действия. Именно в этом случае не используйте глобальный указатель формы, а саму форму не создавайте автоматически. Удалите строку `Application->CreateForm` из кода функции `WinMain` либо удалите форму из списка Auto-create forms вкладки Forms диалогового окна Project Options, которое появляется на экране после выбора команды меню Project⇨Options. Удалите или закомментируйте указатель этой формы во всех файлах с расширениями `.h` и `.cpp`, а потом укажите явно в заголовочном файле, что эта форма является модальной и должна использоваться только вместе с методом `ShowModal()`. То есть, из файла с расширением `.cpp` удалите строку

```
TFormX* FormX;
```

а из файла с расширением .h, удалите следующую строку:

```
extern PACKAGE TFormX* FormX;
```

Добавьте при этом такой комментарий:

```
//Это модальная форма и ее следует вызывать только методом ShowModal()
```

Для использования этой формы нужно просто записать показанную ниже строку с вызовом.

```
TFormX* FormX = new TFormX(0);
try
{
   FormX->ShowModal();
}
__finally
{
   delete FormX;
}
```

Поскольку указатель формы должен иметь определенное значение, то его следует объявить как постоянный с помощью ключевого слова const.

```
TFormX* const FormX = new TFormX(0);
try
{
   FormX->ShowModal();
}
__finally
{
   delete FormX;
}
TFormX(this);
FormX->ShowModal();
delete FormX;
```

Использование блока try/__finally гарантирует защищенность кода при возникновении исключительных ситуаций. Альтернативный вариант заключается в использовании шаблона класса auto_ptr из стандартной библиотеки.

```
auto_ptr<TFormX> FormX(new TFormX(0));
FormX->ShowModal();
```

Какой бы метод не использовался, в любом случае гарантируется, что если выполнение кода преждевременно завершится из-за возникновения исключительной ситуации, то форма FormX будет удалена автоматически. При использовании первого метода это произойдет в блоке __finally; а при использовании второго метода — когда auto_ptr выйдет за пределы области действия. Второй метод можно усовершенствовать, применив ключевое слово const для шаблона класса auto_ptr, как показано в следующем коде. (Более подробную информацию можно найти в книге *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*, Sutter, 2000, p. 158.)

```
const auto_ptr<TFormX> FormX(new TFormX(0));
FormX->ShowModal();
```

В этих фрагментах кода следует особо отметить, что в качестве аргумента для параметра AOwner формы FormX передается число 0. Дело в том, что мы сами вызвались обработать событие удаления формы.

Шаблон класса auto_ptr представляет собой эффективный способ управления памятью для VCL-объектов. Он безопасен по отношению к исключительным ситуациям и прост в использовании. VCL-объекту, конструктор которого принимает параметр с правами владения, может просто принять значение 0, поскольку известно, что этот объект будет удален при выходе auto_ptr за пределы области действия.

Если форма является немодальной, то следует выбрать один из двух способов ее создания: автоматический или неавтоматический. В последнем случае следует удалить соответствующую строку из функции WinMain. Для ее создания следует использовать глобальный указатель формы и оператор new. Для отображения формы на экране следует применить метод Show(). Еще раз напомним, что модальную форму нельзя удалить, потому что метод Show() возвращает значение при отображении формы, а не при ее закрытии. Следовательно, она еще может использоваться. Например, если форма создается автоматически, то для ее отображения на экране можно использовать такой код:

```
FormX->Show();
```

В противном случае ее нужно создать и лишь затем отобразить.

```
FormX = new TFormX(this);
FormX->Show();
```

Очень важно избежать повторного создания формы после ее автоматического создания, т.к. в этом случае будет переписана ссылка автоматически созданной формы. Это значит, что автоматически созданный экземпляр будет недоступен для приложения и может привести к сбою приложения при попытке разыменования глобального указателя. Следовательно, для устранения таких ситуаций рекомендуется проверять значение указателя формы (на равенство нулю) до создания формы:

```
if(FormX == 0) FormX = new TFormX(this);
FormX->Show();
```

Это вполне возможно, поскольку указатель формы является глобальной переменной, которая гарантированно инициализируется значением 0. Использование этого метода позволяет быть уверенным, что глобальный указатель всегда будет указывать на истинное расположение объекта и не будет ошибочно переписан.

Отступая от этой темы, заметим, что практика объявления переменных или функций как статических только для того, чтобы они имели область действия в пределах единицы трансляции, в которой они объявлены, является порочной. Вместо этого такие переменные и функции следует располагать в *неименованном* (*unnamed*) пространстве имен. (Более подробную информацию по этому поводу можно найти в книге *ANSI/ISO C++ Professional Programmer's Handbook: The Complete Language*, Kalev, 1999, p. 157.)

Эффективное применение ключевого слова const

Ключевое слово const следует использовать по назначению, а не как вспомогательное средство. Объявление переменной как постоянной с помощью ключевого слова const позволяет обнаружить попытки изменения значения этой переменной во время компиляции (что в результате могло бы привести к ошибке), а также указывает на желание программиста не изменять значение этой переменной. Более того, отсутствие ключевого слова const означает

намерение программиста изменить эту переменную. Ключевое слово const может использоваться несколькими различными способами.

Во-первых, его можно использовать для объявления константы.

```
const double PI = 3.141592654;
```

Именно этот способ используется в языке C++ для объявления констант. Не следует для этого использовать выражения на основе оператора #define. Обратите внимание на то, что константы должны быть инициализированы. В приведенном ниже примере показаны различные варианты объявления констант. Учтите, что объявления указателей и ссылок считываются справа налево.

```
int Y = 12;
const int X = Y;   // X равно Y, которое равно 12, значит, X = 12
                   // X изменять нельзя, а Y – можно

// В следующем объявлении сам указатель является константой

int* const P = &Y;   // P – это указатель на значение типа int,
                     // т.е. Y можно изменить посредством P,
                     // но сам указатель P изменить нельзя

// Следующие два определения эквивалентны:

const int* P = &Y;   // P – это указатель на константу типа int,
int const* P = &Y;   // Y нельзя изменить посредством P


// Следующие два определения эквивалентны:

const int* const P = &Y;   // Нельзя изменить ни P, ни Y
int const* const P = &Y;   // посредством P

// Следующие два определения эквивалентны:

const int& R = Y      // R – это указатель на константу типа int
int const& R = Y      // т.е. Y нельзя изменить посредством R
```

При ознакомлении с этими примерами полезно еще раз вспомнить способы применения ключевого слова const в объявлениях указателя. Как уже говорилось, объявление указателя считывается справа налево, поэтому в объявлении int * const ключевое слово const относится к указателю *. Следовательно, константой является указатель, хотя указанное им значение типа int может быть изменено. В объявлении int const * ключевое слово const относится к переменной типа int. В этом случае переменная типа int является константой, а указатель — нет. Наконец, в объявлении int const * const константами являются и переменная типа int, и указатель *. Кроме того, напомним, что объявления int const и const int эквивалентны, поэтому объявление const int * const эквивалентно объявлению int const * const.

Для объявления строки символов рекомендуется использовать одно из следующих объявлений:

```
const char* const LiteralString = "Hello World";
char const * const LiteralString = "Hello World";
```

Обе эти строки и их указатели являются константами.

Параметры функции следует объявлять как константы в тех случаях, когда функция не должна изменять аргумент, передаваемый функции. Например, в приведенном ниже примере указано, что переданные функции аргументы не должны изменяться.

```
double GetAverage(const double* ArrayOfDouble,
                  int LengthOfArray)
{
   double Sum = 0;

   for(int i=0; i<LengthOfArray; ++i)
   {
      Sum += ArrayOfDouble[i];
   }

   double Average = Sum/LengthOfArray;
   return Average;
}
```

Еще один способ интерпретации отсутствия ключевого слова `const` заключается в том, что аргумент этого параметра предполагается изменять внутри функции, за исключением случая, когда этот параметр передается по значению (передается копия параметра, а не сам параметр). Обратите внимание на то, что вряд ли стоит передавать параметр `int LengthOfArray` как константу, потому что он передается по значению. `LengthOfArray` фактически является копией, и ее объявление константой никак не повлияет на аргумент, передаваемый функции. Аналогично, параметр `ArrayOfDouble` объявлен как указатель на постоянное значение типа `double`.

```
const double* ArrayOfDouble
```

Объявление указателя как константы

```
const double* const ArrayOfDouble
```

не имеет смысла, потому что сам указатель является копией, а потому необходимо привести ключевое слово `const` только для данных, на которые он указывает.

Возвращаемое функцией значение также может быть константой. Обычно не рекомендуется объявлять возвращаемые значения как константы, за исключением тех случаев, когда необходимо организовать вызов перегруженного постоянного члена-функции. В таких случаях постоянными следует объявлять возвращаемые указатели и ссылки.

Члены-функции могут быть объявлены как постоянные. Постоянным членом-функцией класса называется функция, которая не модифицирует этот объект (`*this`). Следовательно, она может внутри своего тела вызывать другие члены-функции, только если они также являются постоянными. Для объявления постоянного члена-функции, поместите ключевое слова `const` в конце объявления и определения функции. Вообще, все функции-члены, предназначенные для чтения значения (getter member function), следует объявлять как постоянные, потому что они не изменяют объект `*this`. Рассмотрим следующий пример класса:

```
class Book
{
    private:
            int NumberOfPages;
     public:
            Book();
```

```
            int GetNumberOfPages() const;
} ;
```

Определение члена-функции класса `GetNumberOfPages()` может выглядеть так, как показано ниже.

```
int Book::GetNumberOfPages() const
{
    return NumberOfPages;
}
```

Наконец, еще один распространенный вариант использования ключевого слова `const` обычно встречается при перегрузке операторов в классе, когда требуется предоставить доступ к постоянным и непостоянным переменным. Например, если класс `ArrayOfBooks` создается для хранения объектов `Book`, то разумно было бы предположить, оператор `[]` будет перегружен (для того чтобы класс действовал как массив). Однако при этом следует рассмотреть вопрос о применимости оператора `[]` для постоянных и непостоянных объектов. Решение заключается в том, чтобы перегрузить этот оператор с учетом его применения для постоянных значений, как показано ниже.

```
class ArrayOfBooks
{
    public:
           Book&       operator[] (unsigned i);
           const Book& operator[] (unsigned i) const;
} ;
```

Теперь класс `ArrayOfBooks` может использовать оператор `[]` для постоянных и непостоянных объектов `Book`. Например, если объект `ArrayOfBooks` передается функции по ссылке на постоянное значение, то в этом случае нельзя будет присвоить значение какому-либо элементу этого массива с помощью оператора `[]`. Дело в том, что значение с индексом `i` будет постоянным, а потому постоянное состояние переданного массива будет сохранено.

Хорошенько запомните эти советы по поводу применения ключевого слова `const` и используйте его в случае необходимости.

Принципы обработки исключительных ситуаций

Исключительной ситуацией называется механизм обработки ошибок во время выполнения программы. Среди нескольких методов такой обработки следует назвать возвращение кода ошибки, установку глобальных флагов ошибки, а также завершение работы программы. Во многих случаях исключительная ситуация — это единственный подходящий и достаточно эффективный метод, например, для устранения ошибки в работе конструктора. (Более подробную информацию по этому поводу можно найти в книге *ANSI/ISO C++ Professional Programmer's Handbook: The Complete Language* , Kalev, 1999, p. 113.)

В программах C++Builder обработка исключительных ситуаций обычно выполняется на основе стандартных средств языка C++ и библиотеки VCL. Основные принципы этих средств практически идентичны, за исключением некоторых отличий.

В языке C++ для обработки исключительных ситуаций используются три ключевых слова: `try`, `catch` и `throw`. В C++Builder, кроме них, применяется также ключевое слово `__finally`.

Ключевые слова `try`, `catch` и `__finally` используются как заголовки для блоков кода (т.е., кода, заключенного в скобках). Кроме того, для каждого блока `try` обязательно должен существовать один или несколько блоков `catch` или один блок `__finally`.

Ключевое слово try

Ключевое слово try можно использовать двумя способами. Первый и самый простой способ заключается в создании заголовка блока try внутри функции. Второй способ заключается в создании блока *имя_функции* try с помощью размещения ключевого слова try перед первой скобкой, открывающей тело функции, либо — в случае конструкторов — перед тем столбцом, который обозначает начало списка инициализации.

На заметку

В C++Builder в настоящее время не поддерживается работа с блоками *имя_функции* try. Однако так как это имеет значение только при работе с конструкторами, причем очень незначительное, то их отсутствие вряд ли будет замечено. Здесь же отметим, что в версию 6 компилятора предполагается включить поддержку блока *имя_функции* try.

Ключевое слово catch

Обычно, вслед за блоком try следует по крайней мере один блок catch (или блок *функция* try). Блок catch всегда содержит ключевое слово catch, за которым следуют скобки, содержащие одну спецификацию типа исключительной ситуации, возможно, с именем переменной. Такой блок catch, который обычно называется *обработчиком исключительной ситуации* (*exception handler*), может перехватывать только те исключительные ситуации, тип которых точно соответствует типу исключительной ситуации, указанной в блоке catch. Однако используя блок catch с символом многоточия, catch(...), можно указать на необходимость перехвата *всех* исключительных ситуаций.

Обычно способ использования конструкции try/catch выглядит следующим образом.

```
try
{
   // Код, который может вызвать исключительную ситуацию
}
catch(exception1& e)
{
   // Код обработчика исключительных ситуаций типа exception1
}
catch(exception2& e)
{
   // Код обработчика исключительных ситуаций типа exception2
}
catch(...)
{
   // Код обработчика исключительных ситуаций любого типа
}
```

Ключевое слово __finally

Ключевое слово __finally позволяет выполнять операции очистки, а также гарантирует выполнение некоторого кода независимо от того, возникла или нет исключительная ситуация. Это возможно благодаря тому, что код внутри блока __finally выполняется всегда, даже когда исключительная ситуация уже помещена в соответствующий блок try. Это позволяет создавать безопасный и корректно работающий код даже при наличии исключительных ситуаций. Типичный способ использования конструкции try/__finally имеет следующий вид:

```
try
{
   // Код, который может вызвать исключительную ситуацию
}
__finally
{
   // Код в этом месте выполняется всегда,
   // даже если исключительная ситуация возникла в
   // предыдущем блоке try
}
```

Следует отметить, что конструкции try/catch и try/__finally могут быть вложены в другие конструкции try/catch и try/__finally.

Ключевое слово throw

Ключевое слово throw может использоваться двумя способами. Во-первых, для вызова (или повторного вызова) исключительной ситуации, а во-вторых, для указания типа исключительных ситуаций, которые функция может вызвать. В первом случае (вызов или повторный вызов исключительной ситуации) за ключевым словом throw могут следовать скобки, содержащие одну переменную исключительной ситуации (часто это некий объект), или только переменную исключительной ситуации сразу после пробела, аналогично выражению с оператором return. Если такая переменная исключительной ситуации не используется, то интерпретация ключевого слова throw зависит от его расположения. При размещении внутри блока catch выражение throw повторно вызывает ту исключительную ситуацию, которая обрабатывается в настоящий момент. При размещении его в другом месте, например при отсутствии исключительной ситуации, которую можно было вызвать повторно, вызывается функция terminate(), завершающая выполнение программы. Его нельзя использовать в коде VCL для вызова или повторного вызова исключительной ситуации. Второй способ использования ключевого слова throw предназначен для указания спецификации исключительных ситуаций, которые может вызвать функция. Ниже показан синтаксис использования этого ключевого слова.

```
throw(<exception_type_list>)
```

Список типов исключительных ситуаций exception_type_list приводится по желанию, а его отсутствие означает, что функция не вызывает исключительных ситуаций. Они указываются через запятую и являются единственными исключительными ситуациями, которые могут быть вызваны этой функцией.

Необработанные и неожиданные исключительные ситуации

Помимо описанных выше трех ключевых слов, в языке C++ предусмотрены способы работы с необработанными и неожиданными исключительными ситуациями. Это может произойти при вызове исключительной ситуации внутри функции с несовместимой спецификацией исключительной ситуации.

Если вызванная исключительная ситуация не обрабатывается, то вслед за этим вызывается функция terminate(). Она вызывает используемую по умолчанию функцию обработки завершения программы, которая, в свою очередь, по умолчанию вызывает функцию abort(). Рекомендуется избегать таких действий по умолчанию, поскольку при вызове функции abort() не гарантируется вызов деструкторов локальных объектов. Для предотвращения вызова функции terminate() в результате неперехваченной исключительной ситуации, следует заключить всю программу в блок конструкции try/catch(...) в функции WinMain() (или

main() для консольных программ). Таким образом гарантируется, что любая исключительная ситуация будет перехвачена. При вызове функции terminate() ее заданное по умолчанию поведение можно изменить, указав собственную функцию обработки завершения работы программы. Для этого нужно передать имя собственной функции обработки завершения программы в качестве аргумента функции std::set_terminate(). Кроме того, в текст кода следует включить заголовочный файл <stdexcept>. Допустим, что для этого используется показанная ниже функция.

```
void TerminateHandler();
```

Код вызова этого обработчика вместо основной функции обработки завершения работы программы terminate() будет выглядеть, как показано ниже.

```
#include <stdexcept>

std::set_terminate(TerminateHandler);
```

При появлении неожиданной исключительной ситуации вызывается функция unexpected(). По умолчанию она вызывает функцию terminate(). В этом случае также можно определить собственную функцию для обработки этой ситуации. Для этого нужно вызвать функцию std::set_unexpected(), передавая ей в качестве параметра имя собственной функции обработки ситуации. В код программы следует включить заголовочный файл <stdexcept>.

Использование исключительных ситуаций

В этом разделе рассматриваются исключительные ситуации, которые могут быть вызваны и которые следует вызывать с помощью функции, а также, то где их следует перехватывать. Эти вопросы следует рассмотреть во время создания кода, а не после окончания работы над ним. С этой целью программист должен учесть несколько важных и сложных особенностей, описание которых выходит за рамки этой книги. Более подробное описание этой проблемы можно найти в разделе о рекомендуемой литературе в конце этой главы.

Прежде всего необходимо решить, сколько исключительных ситуаций может появиться в вашей программе: одна или несколько. Затем нужно определить диапазон перехвата и обработки одной или нескольких исключительных ситуаций: будет ли он ограничен только текущей областью действия или распространяться за ее пределы. Если диапазон перехвата и обработки исключительных ситуаций не требуется распространять за пределы текущей области действия, то нужно разместить код в блоке try, а вслед за ним — один или несколько соответствующих блоков catch для перехвата необходимых исключительных ситуаций (или сразу всех исключительных ситуаций с помощью блока catch(...)). При этом нужно учесть возможность обработки исключительных ситуаций, предусмотренную в языке, стандартной библиотеке C++ и в библиотеке VCL, а также место их возможного появления. Например, при неудачной попытке выделения достаточного объема памяти во время выполнения оператора new будет вызван обработчик этой исключительной ситуации std::bad_alloc.

Вызывать исключительную ситуацию в функции следует только в случаях необходимости, например, если функция не достигает своей цели. (Более подробное обсуждение этих вопросов можно найти в разделе об использовании комментариев выше в этой главе. См. также *C++ FAQs, Second Edition*, Cline et al., 1999, p. 137.)

Перехват исключительной ситуации следует выполнять по ссылке и только тогда, когда вам точно известно, как эту ситуация следует обрабатывать. (Более подробные сведения по этой теме можно найти в *More Effective C++: 35 New Ways to Improve Your Programs and Designs* , Meyers, 1996, p. 68.) Исключительные ситуации в библиотеке VCL нельзя перехватить по значе-

нию. Кроме того, не всегда возможно полностью восстановить состояние программы после появления исключительной ситуации; в таком случае ее обработчик должен выполнить все необходимые операции очистки, а затем повторно вызвать исключительную ситуацию.

При этом следует четко представлять себе, когда и как применять спецификации исключительных ситуаций для функций, а также стараться избегать появления некорректных спецификаций. В противном случае внутри вашей функции будет вызвана функция `unexpected()`, которая не будет обработана внутри этой функции.

При этом следует гарантировать создание безопасного кода по отношению к возможным случаям возникновения исключительных ситуаций. Например, приведенный ниже код не является безопасным в этом смысле.

```
TFormX* const FormX = new TFormX(0);
FormX->ShowModal();
delete FormX;
```

Если исключительная ситуация возникнет между событиями создания и удаления формы, то эта форма никогда не будет удалена, а потому такой код не будет работать должным образом при наличии исключительных ситуаций. Альтернативный вариант создания безопасного кода по отношению к возможным случаям возникновения исключительных ситуаций описывается выше в этой главе в разделе о том, как можно и нужно избегать применения глобальных переменных.

При создании классов-контейнеров следует стремиться к созданию *нейтрального по отношению к исключительным ситуациям кода* (*exception-neutral code*). В таком коде появляющиеся исключительные ситуации будут передаваться объекту, вызвавшему функцию с этим кодом. (Более подробную информацию по этому поводу можно найти в книге *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions* , Sutter, 2000, p. 25.)

Ни в коем случае не выполняйте вызов исключительной ситуации в деструкторе, потому что деструктор может быть вызван в результате очистки стека после вызова обработчика для предыдущей исключительной ситуации. В итоге он вызывает функцию `terminate()`. Деструкторы должны содержать спецификацию исключительной ситуации в виде `throw()`.

Заключительные замечания по поводу исключительных ситуаций

Наконец, следует отметить отличия между способами обработки исключительных ситуаций в библиотеке VCL и языке C++. В библиотеке VCL обрабатываются исключительные ситуации как операционной системы, так и генерированные самой программой. Такие исключительные ситуации должны перехватываться по ссылке. В библиотеке VCL исключительные ситуации, генерированные внутри программы, не могут перехватываться по значению. Преимущество исключительных ситуаций в библиотеке VCL заключается в том, что они могут быть вызваны и перехвачены в IDE-среде.

Применение операторов `new` и `delete` для управления памятью

В библиотеке VCL все классы, унаследованные от класса `TObject`, должны создаваться динамически в свободной области памяти. Свободная область памяти часто называется кучей (heap), но в контексте выделения и освобождения памяти с помощью операторов `new` и `delete` следует использовать более точный термин “свободная область памяти”. А термин “куча” следует оставить для случаев выделения и освобождения памяти с помощью операторов `malloc()` и `free()`. (Более подробное обсуждение этого вопроса можно найти в книге *Exceptional C++: 47*

Engineering Puzzles, Programming Problems, and Solutions , Sutter, 2000, p. 142.) Программы в C++Builder могут содержать большое количество вызовов операторов new и delete, а потому понимание принципов их работы имеет очень большое значение.

Объект типа non-POD (non-plain old data) далеко не самый тривиальный класс, потому что он должен иметь собственную память, выделяемую с помощью оператора new. Для этого недостаточно использовать эквивалентный оператор malloc() языка C (его поведение будет неопределенным). По окончании работы с объектом эту память следует освободить с помощью оператора delete, а не оператора free(). Операторы new и delete гарантируют не только выделение/освобождение памяти, но и вызов конструктора/деструктора, соответственно.

Оператор new также возвращает указатель созданного объекта, а не только указатель типа void, который следует привести к нужному типу. Операторы выделения/освобождения памяти operator new и operator delete могут быть перегружены для особых классов. Это позволяет нужным образом настроить процедуры выделения/освобождения памяти, что невозможно при использовании malloc() и free().

(Более подробную информацию по этому поводу можно найти в книге *ANSI/ISO C++ Professional Programmer's Handbook: The Complete Language*, Kalev, 1999, p.221).

Успешный вызов оператора new позволяет выделить достаточное количество памяти в свободной области памяти, приводит к вызову конструктора объекта и возвращает указатель-на-тип-созданного-объекта. Результатом этих действий будет корректно инициализированный объект. Соответственно, вызов оператора delete приводит к вызову деструктора объекта и освобождает память, полученную ранее с помощью вызова оператора new.

Для удаления объекта библиотеки VCL всегда используйте метод delete, а не метод Free() . При этом будет гарантировано вызван деструктор объекта и освобождена память, которая предварительно была выделена оператором new. Метод Free() это не гарантирует, а потому его не стоит использовать в этих целях.

Если вызов new оказался неудачным, то возникнет исключительная ситуация std::bad_alloc. Обратите внимание, что исключительная ситуация bad_alloc определена в стандартном библиотечном файле <new>. Следовательно, в код программы необходимо включить строку #include <new>, которая будет находиться в пространстве имен std. Эта исключительная ситуация не возвращает NULL, а потому возвращаемый указатель не следует проверять на равенство NULL. В программе следует организовать перехват исключительной ситуации std::bad_alloc и если функция, которая вызывает оператор new не перехватывает ее, то ее следует передать за пределы функции, чтобы код вызова функции мог перехватить ее. Эту задачу можно выполнить, например, так:

```
void CreateObject(TMyObject* MyObject) throw()
{
   try
   {
      MyObject = new TMyObject();
   }
   catch(std::bad_alloc)
   {
      // Печать сообщения
      // "Недостаточно памяти для создания MyObject";
```

```
      // Далее можно разместить код обработки этой ситуации
      // или деликатного выхода.
   }
}
```

Можно использовать и такой способ:

```
void CreateObject(TMyObject* MyObject) throw(std::bad_alloc)
{
   MyObject = new TMyObject();
}
```

Использование исключительных ситуаций позволяет организовать централизованную обработку исключительных ситуаций, благодаря чему создаваемый код является более безопасным и интуитивно понятным. Ключевое слово `throw` в заголовке функции называется *спецификацией исключительной ситуации*. Это делается для того, чтобы указать исключительные ситуации, которые могут возникнуть при выполнении этой функции. Более подробно эта тема рассматривается в разделе о принципах работы с исключительными ситуациями выше в этой главе. В первом примере функции `CreateObject()` спецификатор исключительной ситуации `throw()` используется для обозначения того, что никакая исключительная ситуация не будет вызвана функцией. Это допустимо, поскольку единственная возможная исключительная ситуация `std::bad_alloc` перехватывается и обрабатывается самой функцией. Во втором примере функции `CreateObject()` спецификатор исключительной ситуации `throw(std::bad_alloc)` используется для обозначения того, что единственной исключительной ситуацией при выполнении этой функции может быть `std::bad_alloc`. Ее следует перехватить и обработать с помощью одной из вызываемых процедур.

Кроме того, можно создать собственный обработчик исключительной ситуации “нехватки памяти”, связанной с неудачным выделением памяти. Для создания функции-обработчика “нехватки памяти” при использовании оператора `new` следует вызвать функцию `set_new_handler()` (она также определена в библиотеке `<new>`) и передать ей в качестве параметра имя функции, которая будет использована как обработчик “нехватки памяти”. Например, для создания функции (не члена класса или статического члена класса) `OutOfMemory` для обработки таких ситуаций можно создать такой код:

```
#include <new>

void OutOfMemory()
{
   // Попытайтесь освободить какую-то часть памяти.
   // Если памяти достаточно, то такая функция
   // НЕ будет вызвана в следующий раз,
   // в противном случае установите новый обработчик или
   // вызовите новую исключительную ситуацию.
}

// Где-нибудь в начале основного кода следует указать:
std::set_new_handler(OutOfMemory);
```

К этому коду нужно добавить несколько пояснений, потому что последовательность событий, которые возникают при неудачном выполнении оператора `new` диктует способ организации функции `OutOfMemory`. Если при выделении необходимого количества памяти выполнение оператора `new` закончилось неудачно, то вызывается функция-обработчик `OutOfMemory`.

Функция-обработчик OutOfMemory пытается освободить некоторую часть памяти (как этого добиться, будет описано позднее). Затем оператор new снова попытается выделить необходимый объем памяти. При успешном выполнении этих действий обработка ситуации завершается. В противном случае только что описанный процесс будет повторен. Он будет повторяться бесконечное количество раз до тех пор, пока не будет выделено достаточное количество памяти или функция OutOfMemory сама не прекратит этот процесс.

Прекратить этот процесс с помощью функции OutOfMemory можно несколькими способами: вызвать появление исключительной ситуации (например, std::bad_alloc()), установить другой обработчик исключительной ситуации, который позволит выделить больше памяти, передать значение NULL для set_new_handler (std::set_new_handler(0)) или завершить выполнение программы (не рекомендуется). При создании нового обработчика такая же цепь событий произойдет и для нового обработчика (который будет вызван при следующей неудачной попытке выделения памяти). Если обработчику передано значение NULL (0), то обработка выполняться не будет, а будет вызвана исключительная ситуация std::bad_alloc().

При выделении дополнительного объема памяти следует учитывать структуру программы и причины, вызвавшие сокращение доступной памяти. Если программа резервирует большое количество памяти в целях повышения производительности, но не всегда использует ее, то при необходимости память можно освободить принудительно. Найти такую память — чрезвычайно трудная задача. Если в программе память используется иначе, то нехватка памяти будет результатом внешних факторов (например, вследствие работы программного обеспечения с интенсивным использованием оперативной памяти или из-за физических ограничений). В последнем случае ничего нельзя сделать, кроме отображения на экране предупреждения для пользователя о возможной нехватке памяти с рекомендацией закрыть приложения, интенсивно использующие оперативную память.

Хитрость в данном случае заключается в том, чтобы отобразить такое сообщение еще до того как память будет полностью исчерпана. Один подход для решения этой проблемы заключается в предварительном выделении некоторого объема памяти в начале работы программы. Если при выделении необходимого количества памяти выполнение оператора new закончится неудачей, то эта память будет освобождена. В таком случае пользователь получает предупреждение о недостаточном объеме памяти с рекомендацией освободить ее для продолжения работы с приложением. Если предварительно выделенный блок памяти имел достаточно большой размер и пользователю удастся освободить его для увеличения объема свободной памяти, то программа сможет продолжить работу. Этот "вытесняющий" подход достаточно просто реализовать. О других возможных подходах для решения этой задачи читатель сможет узнать из материалов, которые перечислены в списке рекомендованной литературы ниже в этой главе.

Следует отметить, что, если нужно выделить определенный объем неструктурированной памяти, то вместо new и delete удобнее использовать operator new и operator delete. (Более подробно эта тема описывается в книге *More Effective C++: 35 New Ways to Improve Your Programs and Designs*, Meyers, 1996, p. 38.) Это используется в тех ситуациях, когда нужно динамически выделить память, размер которой определяется с помощью вызова функции еще до динамического выделения памяти. Это довольно обычная ситуация при программировании с использованием Win32 API-функций.

```
DWORD StructureSize = APIFunctionToGetSize(SomeParameter);

WIN32STRUCTURE* PointerToStructure;
PointerToStructure = static_cast<WIN32STRUCTURE*>(operator new(StructureSize));
// Какие-то действия с этой структурой
operator delete(PointerToStructure);
```

Ясно, что здесь нет необходимости использовать функции malloc() и free().

Наконец, рассмотрим использование операторов new и delete для динамического выделения и освобождения памяти для массивов. Задача выделения и освобождения памяти для массивов решается с помощью операторов new[] и delete[], соответственно, которые отличаются от операторов new и delete. Дело в том, что если оператор new используется для создания массива объектов, то он сначала выделяет память для них (с помощью оператора new[]), а затем инициализирует каждый объект, вызывая его используемый по умолчанию конструктор. Удаление массива с помощью оператора delete выполняет обратную задачу: сначала для каждого объекта вызывается деструктор, а потом освобождается выделенная ранее для массива память (с помощью оператора delete[]). Таким образом, в данном случае вместо оператора delete вызывается оператор delete[], а скобки [] размещаются между ключевым словом delete и указателем на массив, который следует удалить.

```
delete [] SomeArray;
```

Выделение памяти для одномерного массива выполняется очень просто, например, как показано ниже.

```
TBook* ArrayOfBooks = new TBook[NumberOfBooks];
```

Удаление такого массива также не вызывает никаких трудностей. При этом, однако, следует помнить о том, что должна быть использована правильная форма оператора delete, т.е. delete [], как показано в нижеследующем примере.

```
delete [] ArrayOfBooks;
```

Помните: присутствие скобок [] указывает компилятору, что используется массив, а не один элемент заданного типа. Если массив необходимо удалить, то нужно использовать оператор delete [], а не оператор delete. Если по ошибке будет использован оператор delete, то в лучшем случае будет удален только первый элемент массива. Как известно, при создании массива объектов применяется используемый по умолчанию конструктор. Это значит, что для своих потребностей вам придется определить собственный используемый по умолчанию конструктор. Напомним, что генерируемый компилятором конструктор по умолчанию не инициализирует члены-данные класса. Кроме того, вам, возможно, придется перегрузить оператор присваивания (=) для того, чтобы упростить присваивание значений объектам массива. Например, двумерный массив можно создать с помощью показанного ниже кода.

```
TBook** ShelvesOfBooks = new TBook*[NumberOfShelves];

for(int i=0; i<NumberOfShelves; ++i)
{
   ShelvesOfBooks[i] = new TBook[NumberOfBooks];
}
```

А для удаления этого массива следует применить такой код:

```
for(int i=0; i<NumberofShelves; ++i)
{
   delete [] ShelvesOfBooks[i];
}

delete [] ShelvesOfBooks;
```

Следует сделать еще одно важное замечание: для создания массива объектов лучше использовать вектор объектов на основе шаблона из стандартной библиотеки шаблонов STL. Это позволяет использовать любой конструктор, а также автоматически управлять выделени-

ем и освобождением памяти. Кроме того, при нехватке памяти ее можно перераспределить. Это значит, что вам больше не придется использовать функцию realloc() из языка C. Более подробную информацию о шаблоне класса vector можно найти в разделе о стандартных библиотеках и шаблонах языка C++ в главе 4.

Динамическое выделение памяти в определенной области памяти и возвращение значения NULL вместо перехвата исключительной ситуации при неудачном выделении памяти здесь не рассматриваются, т.к. они выходят за рамки этого раздела. Однако более подробную информацию по этой теме можно найти в материалах, приведенных в списке рекомендуемой литературы.

Стили приведения типов в C++

В табл. 3.1 перечислены четыре стиля приведения типов в языке C++.

Таблица 3.1. Стили приведения типов в языке C++

Стиль	Назначение
static_cast<T>(exp)	Используется для приведения такого типа, как int к типу double.
	В качестве T и exp могут использоваться указатели, ссылки, арифметические типы (такие, как int) или перечислимые типы enum. При этом нельзя смешивать разнородные типы, например приводить указатель к арифметическому типу.
dynamic_cast<T>(exp)	Используется для приведения типов вниз или вдоль иерархии наследования. Например, если класс X является наследником класса O, то указатель на класс O может быть приведен к типу указателя на класс X при условии, что такое преобразование корректно.
	В качестве T может использоваться указатель или ссылка на определенный тип класса или тип void*.
	В качестве exp может использоваться указатель или ссылка. Чтобы можно было преобразовать тип от базового класса к производному, базовый класс должен содержать по крайней мере одну виртуальную функцию. Иначе говоря, он должен быть полиморфным. Важной особенностью динамического приведения dynamic_cast является то, что если невозможно выполнить преобразование указателей, то возвращается NULL-указатель; а если невозможно выполнить преобразование ссылок, то возникает исключительная ситуация std::bad_cast (для этого необходимо включить заголовочный файл <typeinfo>). В результате может быть выполнена проверка успешности такого преобразования.
const_cast<T>(exp)	Это единственный стиль приведения, который может повлиять на выражения типа const или volatile. Причем он может быть либо наложен, либо снят. Это единственное назначение const_cast.
	Например, при передаче указателя на данные типа const для функции, которая принимает только указатель на непостоянные данные, причем известно, что эти данные не будут изменены, можно передать этот указатель после приведения const_casting.
	В качестве T и exp должны использоваться одинаковые типы, которые отличаются только фактором постоянства (const или volatile).

Окончание табл. 3.2

Стиль	Назначение
`reinterpret_cast<T>(exp)`	Используется для небезопасного или зависящего от реализации приведения типов. Этот стиль приведения типа следует использовать только в том случае, когда никакой другой не подходит. Дело в том, что он позволяет еще раз интерпретировать выражение как выражение совершенно другого типа, например привести тип `float*` к типу `int*`. Он часто используется для приведения типов указателей функций. Если вам необходимо использовать стиль `reinterpret_cast`, тщательно изучите приемлемость такого приведения и помните о необходимости четкого документирования этого намерения (и, возможно, мотивов использования такого подхода). В качестве `T` должен использоваться указатель, ссылка, арифметический тип, указатель на функцию или указатель на функцию-член. Указатель может быть приведен к целочисленному типу, и наоборот.

На практике наиболее часто используются стили приведения `static_cast` (для простейших преобразований типа, как, например, от `int` к `double`) и `dynamic_cast`.

Пример использования стиля приведения типов `static_cast` приводится в последней строке следующего кода.

```
int Sum = 0;
int* Numbers = new int[20];

for(int i=0; i<20; ++i)
{
   Numbers[i] = i*i;
   Sum += Numbers[i];
}

double Average = static_cast<double>(Sum)/20;
```

Внимательный читатель заметит, что этот код уже приводился выше в этой главе в листинге 3.4.

Динамическое приведение типа `dynamic_cast` наиболее часто используется в C++Builder для динамического приведения типов `TObject* Sender` или `TComponent* Owner`, чтобы гарантировать принадлежность объекта `Sender` или `Owner` к нужному классу, например `TForm`. Так, при размещении компонента в форме может возникнуть необходимость узнать способ его размещения (размещен ли он непосредственно в форме или на панели `Panel`). Для такой проверки необходимо применить следующий код:

```
TForm* OwnerForm = dynamic_cast<TForm*>(Owner);
if(OwnerForm)
{
   //Выполнить обработку, поскольку OwnerForm != NULL, т.е. 0
}
```

Сначала объявляется указатель на нужный тип, а затем он приравнивается результату динамического приведения типов `dynamic_cast`. Если приведение типов оказалось удачным, то указатель укажет на нужный тип и может быть использован для доступа к этому типу. А если приведение неудачно, то будет получен `NULL`-указатель, а значит, он может быть оценен соответствующим образом в последующем логическом выражении. Объект `Sender` может быть

использован аналогично. Помимо этой ситуации, есть много других случаев, когда может потребоваться такое приведение типов. В каждом из них важно четко представить себе цель, а затем определить средства ее достижения.

Каждый стиль приведения типов в языке C++ выполняет особую задачу, и его следует использовать только по назначению. Такие стили легко заметить в коде, что позволяет сделать его более читабельным.

Когда нужно использовать директивы препроцессора

Не стоит использовать директивы препроцессора для определения констант или макрофункции. Для констант следует использовать переменные типа `const` или `enum`, для макрофункции — подставляемые функции `inline` (или шаблоны подставляемых функций `inline`). Учтите также, что в любом случае применение макрофункции может оказаться неэффективным приемом программирования (и в таком случае не потребуются также подставляемые функции).

Например, число π можно определить следующим образом:

```
const double PI = 3.141592654;
```

Если его нужно поместить в определении класса, то следует использовать такой код:

```
class Circle
{
   public:
          static const double PI; // Это только объявление
} ;
```

В файле реализации (`*.cpp`) определение и инициализация константы может быть выполнена таким образом:

```
const double Circle::PI = 3.141592654;
// Это объявление и инициализация константы
```

Обратите внимание на то, что константа класса объявлена как статическая (`static`), т.е. для данного класса существует только одна копия этой константы. Обратите также внимание на то, что эта константа инициализируется в файле реализации (обычно после директивы о включении заголовочного файла с определением этого класса). Исключением из этого правила является инициализация целочисленных типов `char`, `short`, `long`, `unsigned` и `int`. Они могут быть инициализированы непосредственно в определении класса. При необходимости создания группы связанных по смыслу констант рекомендуется использовать перечислимый тип `enum`.

```
enum LanguagesSupported {  English, Chinese, Japanese, French } ;
```

Иногда перечислимый тип `enum` применяется для объявления отдельной целочисленной константы.

```
enum { LENGTH = 255 } ;
```

Такие объявления можно встретить в определениях класса. Однако более корректный подход основан на объявлении статичной константы (как в предыдущем примере с числом `PI`).

Замена макрофункции, например, такой, как показанная ниже,

```
#define cubeX(x) ( (x)*(x)*(x) )
```

на эквивалентную ей подставляемую функцию

```
inline double cubeX(double x) {  return x*x*x; }
```

также не вызывает никаких затруднений.

Обратите внимание на то, что эта функция принимает в качестве аргумента значение типа double. Для передачи этой функции в качестве параметра значения типа int необходимо выполнить приведение к типу double. Так как поведение встраиваемой функции должно быть аналогично поведению макрофункции, то следует избежать такой необходимости. Эта цель может быть достигнута одним из двух способов: перегрузить функцию либо создать шаблон функции. В данном случае перегрузка функции предпочтительней, т.к. шаблон функции предполагает, что эта функция может использоваться для классов, что в ряде случаев неприемлемо. Следовательно, int-версия встраиваемой функции может иметь такой вид:

```
inline int cubeX(int x) {  return x*x*x; }
```

Вообще, директиву #define рекомендуется использовать не для определения констант и макрофункций, а для создания защиты включаемых файлов. Напомним, что защита включаемых файлов применяется в заголовочных файлах, чтобы уже включенный ранее заголовочный файл не был включен снова. Например, типичный заголовочный файл в C++Builder обычно выглядит так:

```
#ifndef Unit1H  // Если модуль Unit1H не определен.
#define Unit1H  // Если нет, то он определяется в этой строке.

// Код заголовочного файла размещается здесь ...

#endif          // Конец условия "если Unit1H не определен".
```

Этот способ гарантирует, что код между директивами #ifndef и #endif будет включен только один раз. При выборе подходящих директив define для заголовочных файлов следует строго придерживаться принятых соглашений. В C++Builder принято использовать прописную букву H в конце имени заголовочного файла. При создании собственных единиц трансляции рекомендуется придерживаться этого соглашения. Конечно, для этого можно использовать другое соглашение об именах, например префикс INCLUDED_ для имени заголовочного файла, но тогда нужно последовательно придерживаться этого соглашения. Применение защиты для включаемых файлов позволяет предотвратить многократное включение заголовочного файла, но для этого она должна быть предварительно обработана.

При соблюдении соглашения об именах в IDE-среде для защиты включаемых файлов (добавление суффикса H в конце имени заголовочного файла), IDE-среда рассматривает единицу трансляции как набор, поэтому она будет именно так представлена в окне менеджера проектов Project Manager. Если вы не хотите, чтобы ваши файлы .cpp и .h интерпретировались таким образом, то не используйте принятое в этой IDE-среде соглашение об именах для защиты включаемых файлов.

Известно, что в очень больших проектах (или, точнее, в проектах с большой и разветвленной структурой включаемых файлов) это может вызывать существенное увеличение времени компиляции. (Более подробные сведения по этому поводу можно найти в книге *Large-Scale C++ Software Design*, Lakos, 1996, p. 82.) Поэтому для предотвращения нежелательного включения ранее определенного файла имеет смысл окружить защитой все выражения include о включении файлов. Например, если модуль Unit1 из предыдущего фрагмента кода также включает модули диалоговых форм ModalUnit1, ModalUnit2 и ModalUnit3, которые используются другими частями программы, то выражения include о включении этих файлов следует окружить защитой, как показано ниже.

```
#ifndef Unit1H  // Если модуль Unit1H не определен.
#define Unit1H  // Если – нет, то он определяется в этой строке.
```

```
#ifndef ModalUnit1H       // Не определен ли модуль ModalUnit1H?
#include "ModalUnit1.h"   // Если – нет, то его нужно включить.
#endif                    // Конец условия
                          // "если ModalUnit1H не определен"

#ifndef ModalUnit2H
#include "ModalUnit2.h"
#endif

#ifndef ModalUnit3H
#include "ModalUnit3.h"
#endif

// Здесь размещается код заголовочного файла ...

#endif          // Конец условия "если Unit1H не определен".
```

Этот способ нельзя назвать элегантным, но он очень эффективен. Напомним, что имена, используемые для защиты включаемых файлов не должны совпадать с именами, которые возникают в вашей программе. В таком случае заданное имя не будет ошибочно заменено другим. Именно поэтому настоятельно рекомендуется применять соглашение об именах и строго следовать ему.

Обратите внимание, что усовершенствованный менеджер проектов Project Manager в C++Builder 5 включает развертываемый список связанных заголовочных файлов для каждого файла с исходным кодом проекта. Для развертывания/свертывания этого списка достаточно щелкнуть на узле возле имени файла с исходным кодом. Обратите внимание, что эти списки основаны на объектных файлах (`.obj`) исходного кода, поэтому для использования этой возможности следует хотя бы раз скомпилировать файл с исходным кодом. Учтите также, что этот список может устареть, если в код вносились изменения без перекомпиляции.

Следует четко представлять себе, когда использование директив препроцессора приносит пользу программе, а когда — вред. Используйте их внимательно и только по назначению.

Стандартная библиотека C++

Стандартная библиотека C++, включая стандартную библиотеку шаблонов Standard Template Library (STL), является такой же составной частью стандарта ANSI/ISO C++, как и определение типа `bool`. Изучая и применяя компоненты этой библиотеки, можно сэкономить много времени, поскольку это избавит вас от ненужного повторного кодирования уже имеющихся компонентов. Стандартная библиотека обладает тем преимуществом над домороженным кодом, что ее компоненты тщательно протестированы, они достаточно быстро работают и являются стандартными, а потому их переносимость является большим плюсом. Ниже перечислены некоторые компоненты стандартной библиотеки.

- Исключительные ситуации, например `bad_alloc`, `bad_cast`, `bad_typeid` и `bad_exception`.
- Утилиты, например `min()`, `max()`, `auto_ptr<T>` и `numeric_limits<T>`.
- Потоки ввода и вывода, например `istream` и `ostream`.

- Контейнеры, например `vector<T>`.
- Алгоритмы, например `sort()`.
- Функции-объекты (функторы), например `equal_to<T>()`.
- Итераторы.
- Строки, например `string`.
- Комплексные числовые типы, например `complex<T>`.
- Специальные контейнеры, например `queue<T>` и `stack<T>`.
- Поддержка интернационализации.

Почти все компоненты стандартной библиотеки являются шаблонами, а большую часть ее составляет библиотека STL, а потому ее можно очень гибко использовать. Например, шаблон класса `vector` может использоваться для хранения однотипных данных. Поэтому его можно использовать вместо массивов языка C++, и это следует делать там, где только возможно. Более подробные сведения о библиотеке STL можно найти в разделе о стандартной библиотеке C++ и ее шаблонах в главе 4.

Список рекомендуемой литературы

В этом разделе перечислены учебники, в которых содержаться подробные сведения по теме этой главы. Они могут стать первоисточником многих сведений о программировании на языке C++. Книги перечислены в алфавитном порядке авторов и представлены вместе с краткой аннотацией для каждой из них.

- Cline, M., Lomow, G., and Girou, M. (1999). *C++ FAQs Second Edition.* Addison-Wesley Longman, Inc.

  Эта книга основана на интерактивном сборнике ответов на наиболее часто возникающие вопросы при программировании на языке C++, который расположен по адресу `http://www.cerfnet.com/~mpcline/c++-faq-lite/`, но книга содержит гораздо больше материала по этой теме. Она охватывает очень широкий круг вопросов — от самых основных до очень сложных, изложение ведется в очень удобной и понятной форме. Это очень хорошая книга, которая может использоваться как отличный справочник и лучше всего подходит в качестве второй книги по C++.

- Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Orientated Software.* Addison-Wesley Longman, Inc.

  Это одна из первых книг, посвященная шаблонам проектов. Приведенные в ней примеры на языке C++ (и Smalltalk) делают ее особенно полезной. Эта книга поможет вам увидеть методы объектно-ориентированного программирования с совершенно другой стороны.

- Horton, I. (1998). *Beginning C++: The Complete Language*. Wrox Press.

  В этой книге полностью описан стандарт ANSI/ISO C++. Она не привязана к какому-то отдельному компилятору и содержит самую последнюю информацию по этой теме. В ней излагаются очень точные и строгие сведения, которые нельзя найти в других книгах. Она может использоваться как основная книга по C++.

- Kalev, D. (1999). *ANSI/ISO C++ Professional Programmer's Handbook: The Complete Language*. Que Corporation.

  Эта относительно свежая книга со многими интересными размышлениями по поводу стандарта C++. Она является удобным практическим справочником по многим слож-

ным аспектам программирования на языке C++. Основное внимание в ней уделяется описанию мотивов включения некоторых современных компонентов, при этом предлагается оригинальная точка зрения, которую редко можно встретить во многочисленных книгах на эту тему.

- Lakos, J. (1996). *Large-Scale C++ Software Design*. Addison-Wesley Longman, Inc.

  Несмотря на такое название, эта книга будет полезна для всех программистов, за исключением тех, кто создает простейшие проекты на языке C++. Она разделена на три части: основные сведения, физическое проектирование и логическое проектирование. В ней предлагается множество советов и принципов (собранных вместе в конце для простоты повторного обращения к ним), что в значительной степени повышает эффект от чтения.

- McConnell, S. C. (1993). *Code Complete: A Practical Handbook of Software Construction* . Microsoft Press.

  В этой книге предлагается очень тщательное описание методологии кодирования. Это очень полезное руководство рекомендуется всем, кто хочет изучить и испытать преимущества разных стилей кодирования. В ней нет описания отдельных методов кодирования, а лишь приводятся комментарии к ним.

- Meyers, S. (1998). *Effective C++ Second Edition: 50 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley Longman, Inc.

  Это, вероятно, наиболее известная книга по программированию на языке C++, она написана в легкой для чтения манере, а ее размер является подтверждением того, что она содержит огромное количество полезной информации.

- Meyers, S. (1996). *More Effective C++: 35 New Ways to Improve Your Programs and Designs*. Addison-Wesley Longman, Inc.

  Это дополнение к предыдущей книге, *Effective C++*, содержащее множество крупиц мудрости, которыми должен овладеть каждый любознательный программист. Книги Мэйерса можно считать обязательными для каждого программиста.

- Sutter, H. (2000). *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*. Addison-Wesley Longman, Inc.

  Как следует из названия, эта книга предназначена для опытных программистов и входит в серию "C++ Guru of the Week", описание которой можно найти в списке рассылки новостей `comp.lang.c++.moderated`. В ней затронуто достаточно много тем и представлен большой объем информации. Она становится особенно полезной после прочтения нескольких других книг из этого списка.

Резюме

Рассмотренное в этой главе можно кратко подытожить так.

- При создании кода выработайте единый стиль кодирования и строго придерживайтесь его. Это позволит вам создать более ясный и легкий в сопровождении код.
- Убедитесь в том, что комментарии кода достаточно свежи и подробны.
- Попытайтесь понять, почему код создан именно таким, какой он есть, и к каким последствиям может привести использование других подходов.
- Старайтесь повысить уровень своих знаний языка C++ и интегрированной среды программирования C++Builder. Это слагаемые повышения производительности вашего труда.

Глава

4

Более сложные методы программирования в C++Builder

Стефан Махо (Stephane Mahaux)
Йото Йотов (Yoto Yotov)
Викаш Шах

В предыдущих главах уже приводились основные сведения о программировании с помощью C++Builder. Теперь наступило время для углубления знакомства с этим пакетом и изучения более сложных методов программирования. В этой главе рассматривается один из самых мощных наборов инструментов, которые имеются в распоряжении программиста на языке C++, стандартная библиотека шаблонов Standard Template Library. Описываются также способы использования интеллектуальных указателей и строгих контейнеров, а также реализация усовершенствованных обработчиков исключительных ситуаций. Кроме того, здесь представлены понятия многопоточной обработки и шаблонов проектирования.

Введение в стандартную библиотеку C++ и библиотеку шаблонов

Стандартную библиотеку Standard C++ Library (SCL) часто называют просто стандартной библиотекой Standard Library или стандартной библиотекой шаблонов Standard Template Library (STL). Она представляет собой стандартизованный набор шаблонных классов и функций, которые образуют мощную оболочку для программирования на языке C++.

На заметку

С исторической точки зрения библиотека Standard Template Library (STL) является всего лишь подмножеством библиотеки Standard C++ Library (SCL), в которой определены контейнеры и алгоритмы. С технической точки зрения стандартный строковый класс, потоки и тип `valarray` существуют отдельно от библиотеки STL. Однако на практике эти термины часто используются как синонимы.

Хотя на первый взгляд эта тема может показаться немного скучной, все же знание методов работы с библиотекой SCL следует рассматривать как обязательный компонент знаний программиста о языке C++. Правильное употребление этих знаний позволит создавать емкие, эффективные и гибкие решения для большинства задач программирования.

Шаблоны языка C++

Для реализации большинства возможностей в библиотеке SCL используются шаблоны C++. Шаблоны C++ являются относительно новым и важным дополнением спецификации ANSI/ISO C++. Обычно при изучении программирования эта тема рассматривается как совершенно отдельная и более сложная, поэтому более подробное ее описание выходит за рамки этой книги. Все же для осознания важности библиотеки SCL желательно ознакомиться с основными сведениями о ней. Вот почему этот раздел предназначен именно для тех читателей, которые совсем не знакомы с этим компонентом программирования на языке C++.

Образно шаблон можно представить себе как форму для выпечки печенья разной формы, только шаблоны используются для “выпечки” разнообразных функций и классов. При создании шаблона компилятору дается указание об использовании того семейства функций и классов, которые будут использоваться в программе. В очень общих чертах в нем описываются общие свойства этих функций или классов.

Рассмотрим простой пример. Предположим, что нам необходимо вызвать функцию `largest()`, которая принимает в качестве аргумента массив целых чисел и количество элементов этого массива, а возвращает наибольшее значение этого массива. Вот как может выглядеть эта функция:

```
int largest(int values[], int numvalues)
{
```

```
   int lrg = values[0];
   for(int i=1; i<numvalues; i++){
      if(values[i] > lrg)
         lrg = values[i];
   }
   return lrg;
}
```

Эта функция имеет одно ограничение: она может работать только с массивом целых чисел. Предположим, что эту функцию нужно также использовать для чисел с плавающей запятой. Для этого потребуется создать новую функцию, даже если ее работа практически ничем не отличается. В данном примере это не такая уж сложная задача. Достаточно скопировать приведенную версию кода, вставить код в новый файл, ключевое слово `int` заменить словом `float` и задача будет решена. Как можно догадаться, на практике не всегда встречаются такие простые примеры. Как же нужно поступить, чтобы организовать поддержку других типов в этой функции? Для решения именно этой задачи и предназначены шаблоны. В листинге 4.1 показано альтернативное решение данной задачи с помощью шаблона.

Листинг 4.1. Шаблон функции поиска максимального значения в массиве

```
template <typename T>
T largest(T values[], int numvalues)
{
   T lrg = values[0];
   for(int i=1; i<numvalues; i++) {
      if(values[i] > lrg)
         lrg = values[i];
   }
   return lrg;
}
```

Хотя эта функция выглядит практически так же, как и исходная функция, в ней появилось новое ключевое слово `template`. Это вовсе не функция, а шаблон, который сообщает, что `T` следует рассматривать как псевдоним для "любого типа". Допустим, что для компилятора используется предлагаемый по умолчанию параметр обработки шаблонов (т.е., снят флажок параметра External группы Templates во вкладке C++ диалогового окна Project Options). В таком случае компилятор сможет генерировать функцию (если это вообще возможно) для использования в вашей программе, если она не была определена явно и имеет характеристики, заданные в этом шаблоне. Для этого компилятор должен ознакомиться с полной спецификацией шаблона до вызова функции. Следовательно, в отличие от функций или классов, в которых объявления обычно располагаются в заголовочных файлах, а их реализации — в файлах с исходным кодом, реализация шаблона должна размещаться вместе с объявлением в заголовочном файле, который можно включить в файл, в котором потребуется использовать данный шаблон. Такой шаблон обычно называется "шаблоном функции" ("function template"), а генерируемая компилятором функция — "шаблонной функцией" ("templatized function") или "специализированной функцией" ("specialized function"). Для иллюстрации использования шаблона `largest`, предположим, что он находится в заголовочном файле `largest.h`, который включен в код консольной программы C++, показанный в листинге 4.2.

Листинг 4.2. Пример консольной программы с демонстрацией использования шаблона функции `largest`

```
#include "largest.h"
#include <string>
#include <iostream>

int main(void)
{
   int ivalues[3 ] == { 2, 6, 3 };
   float fvalues[5 ] == { 0.2, 0.5, 0.3, 0.6, 0.4 };
   std::string svalues[3 ] == {"Standard", "C++", "Library" };
   std::cout << largest(ivalues, 3) << std::endl
             << largest(fvalues, 5) << std::endl
             << largest(svalues, 3);

   return 0;
}
```

В этот код включены заголовочные файлы `<string>` и `<iostream>` библиотеки SCL, а затем объявлено использование пространства имен `std` для определения области действия в программе стандартных идентификаторов `string` (стандартный тип символьной строки) и `cout` (глобальный объект, определяющий способ вывода результатов программы в ее консольном окне). Затем объявлены три массива разных типов с выводом наибольшего значения из каждого массива (т.е., `6`, `0.6`, `Standard`) в отдельных строках с помощью трех разных версий функции `largest()`, основанных на типах `int`, `float` и `string`. Каждая из этих функций генерируется компилятором на основе описанного выше шаблона.

На заметку

Классы-контейнеры библиотеки SCL определены в пространстве имен `std`. Для ссылки на эти классы можно использовать два способа. Во-первых, это можно сделать на основе оператора определения области действия (`::`), как показано ниже:

```
std::cout <<23;
```

Во-вторых, это можно сделать с помощью директивы `using namespace`, которая переносит указанное пространство имен или имя в текущую область действия.

```
using namespace std;
cout <<23;
```

Более подробно пространство имен `std` будет рассмотрено ниже в этой главе.

В генерированных по шаблону функциях должны принимать параметр указанного в шаблоне типа, чтобы по преданным ей аргументам в момент вызова функции можно было определить конкретный вид функции. В приведенном в листинге 4.2 примере шаблона `largest` при вызове функции `largest()` и передаче ей массива типа `string` компилятор может заменить параметр `T` (см. листинг 4.1) типом `string`.

С другой стороны, на шаблоны классов не налагается такое ограничение. При создании отдельного экземпляра объекта класса по шаблону его тип должен быть явно указан компилятору. Например, в библиотеке SCL содержится шаблон класса `list`. Для инициализации объекта `mystrings` типа `list`, предназначенного для хранения строк, следует сделать такое объявление:

```
list<string> mystrings;
```

Шаблоны классов обладают бо́льшими возможностями, чем шаблоны функций. Действительно, шаблоны функций удобны при работе с шаблонными типами, которые сами по себе являются генерированными по шаблону классами. Подробное обсуждение этой темы выходит за рамки данного раздела, а потому ограничимся здесь только несколькими замечаниями.

- Следует уточнить приведенное выше мнение о том, что спецификатор типа шаблона может использоваться для "любого типа". В шаблонной функции `largest` в качестве параметра `T` может использоваться любой тип, который поддерживает приведенные в этом шаблоне операции. Например, тип, который не поддерживает работу с оператором присваивания (`=`) и оператором больше (`>`), будет отвергнут компилятором при попытке генерирования функции, соответствующей этому шаблону.
- Не существует никаких ограничений, связанных с количеством спецификаторов типов, которые могут быть объявлены в шаблоне. В нем можно указать несколько псевдонимов типов, заключая их в круглые скобки и разделяя запятыми, и даже назначая используемые по умолчанию типы с помощью оператора `=`, почти так же, как при указании списка параметров функции. Однако шаблон с более чем двумя спецификаторами без принимаемых по умолчанию значений вряд ли будет использоваться вследствие чрезмерной громоздкости.
- Чрезмерное использование шаблонов может привести к серьезному снижению производительности при создании приложения. В этом случае на компилятор ложится огромная нагрузка, и процесс компиляции может существенно замедлиться. Кроме того, код отдельной версии функции или класса, генерированных по шаблону, дублируется в объектных файлах всех модулей с исходным кодом, где используются данный класс или функция. Это приводит к разбуханию объектного кода, что может стать причиной замедления процесса компоновки. Более того, компоновщик вынужден решать задачу разрешения конфликтов при появлении повторяющейся информации. В результате насыщенный шаблонами код вовсе не может быть скомпонован.

Это введение не предназначалось для изложения основ проектирования шаблонов, а для осторожного погружения в мир наиболее сложных и часто неправильно воспринимаемых вопросов программирования на языке C++. Автор надеется, что ему удалось вызвать у читателей интерес к применению шаблонов, а также привить им циничное отношение к шаблонам, которое должно быть у всех квалифицированных программистов на языке C++. Шаблоны хороши с теоретической точки зрения, но их непросто использовать на практике. Как бы там ни было, но изложенный здесь материал окажется полезным стимулом для дальнейшего изучения методов работы с библиотекой SCL, откуда, как известно, уже нет пути назад.

Компоненты стандартной библиотеки Standard C++ Library

Библиотеку SCL можно разделить на несколько областей или понятий, которые описываются в следующих разделах.

Из-за ограниченного объема этой главы в ней рассматриваются только контейнеры, итераторы и алгоритмы, причем очень кратко. Особое внимание уделяется этим понятиям только из-за очень широкого распространения. Знакомясь с библиотекой, читатель обнаружит, что изученные методы можно сразу же применить на практике.

Помимо контейнеров, следует также познакомиться с адаптерами, но, поняв, что такое контейнер, читатель легко разберется с этим понятием с помощью справочной системы C++Builder Help.

В изучении этой темы большое значение имеет класс `string`. Хотя в C++Builder для работы со строками предусмотрен аналогичный компонент `AnsiString`, совместимый с библиотекой VCL, класс `string` все же удобно использовать для создания переносимого кода. Для поиска справочной информации о работе со строками в контекстной справке следует выбрать команду меню Help⇨C++Builder Help, затем открыть вкладку Index и ввести в текстовом поле `basic_string`.

Классы `valarray` и `bitset` используются только для очень узкого круга приложений.

Наконец, обсуждение потоков ввода/вывода здесь опущено. Для большинства задач обработки данных эта область имеет очень большое значение, но в этом разделе невозможно обсудить ее подробно.

Контейнеры

Контейнеры — это шаблоны классов, которые упрощают работу с наборами однотипных объектов. Они могут быть настроены для работы с любым типом элемента (при условии, что тип элемента удовлетворяет основным условиям). Контейнеры могут принимать самую разную форму: от связанных списков до индексных массивов и ассоциативных массивов, которые управляют парами ключ-значение.

Итераторы

Итераторы — это объекты, которые содержат указатели на отдельные элементы внутри контейнера и упорядочивают элементы внутри контейнеров. Каждый тип контейнера обладает способностью генерировать объекты-итераторы, соответствующие его собственной реализации.

Адаптеры

Адаптеры — это шаблоны классов, которые, по сравнению с контейнерами, имеют ограниченные возможности, но гораздо проще в употреблении. На самом деле они предоставляют только альтернативный интерфейс к одному из фактических типов контейнера (например, контейнер в виде описанного выше списка). Как пример адаптеров следует упомянуть стеки и очереди, в которых поддерживаются только основные операции `push()` (протолкнуть) и `pop()` (вытолкнуть).

Алгоритмы

Алгоритмами называются шаблоны функций, которые могут применяться по отношению к контейнерам для выполнения некой часто используемой операции, например поиска заданного значения, сортировки элементов или удаления дубликатов. В данной библиотеке имеется более 60 алгоритмов.

Класс `string`

Класс `string` предназначен для реализации символьных строк в библиотеке SCL. Он представляет собой частный случай более общего шаблона класса `basic_string`, который предназначен для работы с обычными 8-битовыми символьными значениями, например, типа `char`.

Класс `valarray`

Класс `valarray` представляет собой редко используемый шаблон класса, который можно применять для математических расчетов. Хотя его использование связано со множеством ограничений, в некоторых случаях именно этот контейнер наилучшим образом подходит для работы с числовыми типами.

Класс bitset

Класс bitset очень удобен для работы с двоичными значениями. В нем предусмотрены члены-функции для задания, проверки и обращения битов, а также для преобразования значения, заданного в виде последовательности символов 0 и 1.

Потоки ввода/вывода

Поток ввода/вывода описывает семейство классов и шаблонов классов, которые обрабатывают поток данных. В этих классах и шаблонах предусмотрены гибкие и безопасные способы управления обменом данными между несколькими средами. Под потоками обычно подразумеваются файловые потоки (запись данных на физический диск и считывание их с диска), строковые потоки (передача символьных данных между буферами устройств хранения), а также потоки других типов.

Принципы работы контейнеров и итераторов

При чтении часто приходится делать закладку, иногда даже две закладки на разных страницах, чтобы отметить нужный раздел. Эта ситуация очень хорошо подходит для моделирования принципа работы контейнера. При этом, как показано на рис. 4.1, книга является аналогом контейнера, ее страницы — элементами контейнера, а закладки — итераторами.

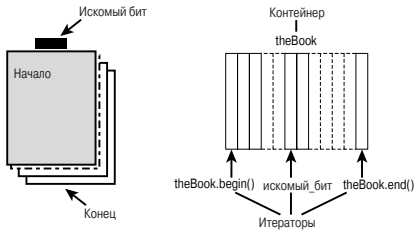


Рис. 4.1. Аналогия между контейнером и книгой с закладками

Теперь можно приступить к описанию наиболее распространенных типов контейнеров. Сначала рассмотрим контейнеры list (список), vector (вектор), deque (дек), set (множество) и map (отображение). (Контейнер map в литературе о языке C++ часто называется словарем, таблицей или ассоциативным массивом. — Прим. ред.) Подробные сведения о них (включая описание интерфейса шаблона класса) можно найти в справочной службе C++Builder 5 Help с помощью вкладки Index. Образно говоря, в этой главе автор лишь постарается вызвать у читателей аппетит к этой теме, а более подробные сведения заинтересованные читатели сами смогут найти в справочной службе C++Builder. При этом читателю придется самостоятельно решать: в каких случаях уместно использовать те или иные компоненты библиотеки SCL и как для этого применить знания, полученные при чтении документации C++Builder.

Во-первых, следует отметить, что доступ к каждому контейнеру можно получить, размещая его имя в директиве #include так, как показано ниже для контейнера list.

```
#include <list>
```

Как уже говорилось, все классы и функции библиотеки SCL должны быть объявлены в пространстве имен std. Следовательно, для объявления списка my_int_list переменных типа int потребуется использовать следующий синтаксис:

```
std::list<int> my_int_list;
```

Во вкладке Index в оперативной справочной службе C++Builder сказано, что шаблон класса list может иметь два аргумента.

```
template < class T, class Allocator = allocator<T> > class list;
```

По умолчанию для второго аргумента класса, Allocator, используется экземпляр типа allocator<T>, т.е. другой шаблон класса библиотеки SCL. Этот класс определяет для шаблона списка способ выделения и высвобождения памяти для элементов. Это тема не имеет большого значения и ее необязательно изучать, но следует иметь ввиду, что в библиотеке SCL есть несколько скрытых и обычно игнорируемых тонкостей, которые нужно учитывать, чтобы не запутаться.

На заметку

Пространство имен std (на самом деле, любое пространство имен) можно сделать текущим с помощью директивы using, например using std::list или using namespace std. Последний вариант позволяет сделать видимыми все идентификаторы библиотеки SCL. Однако во избежание конфликтов и предотвращения случайного изменения области действия при использовании цепочки директив #include рекомендуется явным образом квалифицировать идентификаторы. Для повышения читабельности приведенного кода в нем будет опущена часть std::.

Как уже говорилось, в разных ситуациях следует использовать разные типы контейнеров. Аналогичное замечание относится и к разным типам итераторов, потому что каждый тип контейнера имеет собственный тип итератора. В любом контейнере предусмотрено использование следующих вложенных типов:

- iterator — класс итератора, предназначенный для обхода и модификации элементов контейнера;
- const_iterator — этот тип аналогичен итератору, за исключением того, что в нем не допускается изменение элементов, на которые указывает итератор.

На заметку

Библиотека SCL не соответствует обычному объектно-ориентированному подходу, которым не предусмотрено существование общего базового класса, где определяются сходные черты интерфейса разных контейнеров. В библиотеке SCL предлагается формальная спецификация тех правил, которым контейнеры должны удовлетворять для полной совместимости с библиотекой SCL. В основном библиотека SCL не является объектно-ориентированной библиотекой. В ней используются компоненты языка C++, чтобы в максимальной степени расширить сферу применения и повысить эффективность ее использования.

Все типы итераторов ведут себя как традиционные указатели языка C, в том смысле, что можно перегрузить такие функции-члены: оператор доступа ->, оператор разыменования *, а также оператор инкремента ++. Если итератор i указывает на некоторый элемент контейнера, то *i ссылается на него, i->x осуществляет доступ к члену x этого элемента, а i++ указывает на следующий элемент контейнера.

В каждом контейнере предусмотрены функции-члены begin и end. Функция begin возвращает объект iterator (или объект const_iterator, контейнер имеет тип const), что позволяет получить доступ к первому элементу контейнера. Функция end возвращает объект iterator, что позволяет получить доступ не к последнему элементу контейнера, а к некоторому мнимому *идущему вслед за ним элементу* (*past-the-end value*). В данном случае нужно только знать, что он находится вслед за последним элементом контейнера.

Для демонстрации способа итеративного доступа к элементам контейнера рассмотрим в качестве примера описанный выше контейнер std::list<int>. Он выбран достаточно про-

извольно, поскольку во всех типах контейнеров используется такой распространенный компонент, как итерация. Напомним, что для использования шаблона класса `std::list` в код нужно прежде вставить директиву `#include <list>`. Для доступа ко всем элементам контейнера можно применить показанный ниже цикл `for`.

```
for(std::list<int>::iterator i = lst.begin();
    i != lst.end(); i++)
{
   std::cout << *i << std::endl;
}
```

Контейнеры `list`, `vector` и `deque`

Эти три типа контейнеров удобно рассмотреть вместе, поскольку они имеют целый ряд важных сходств.

Контейнер `list` (список) используется для представления дважды связанного списка. Он наиболее эффективен для вставки или удаления элементов. Список также прекрасно подходит для организации контейнера, в котором регулярно нужно проводить сортировку (хотя, как будет показано ниже, для вставки новых элементов в соответствии с заданным порядком сортировки лучше подходит контейнер `set`). В шаблоне списка поддерживаются двунаправленные итераторы. Двунаправленным итератором называется тип, который позволяет выполнять обход соседних элементов в прямом и обратном направлениях. Для этого в классе `list<T>::iterator` предусмотрена перегрузка оператора `++` и оператора `--`.

Тип `vector` (вектор) является контейнером с произвольным доступом, что означает возможность доступа к его элементам по индексу с помощью оператора [ ], т.е. точно так же, как к элементам обычного массива. В этом контейнере предусмотрена поддержка итераторов с произвольным доступом. В нем также имеются операторы инкремента и декремента, которые используются в качестве двунаправленных итераторов в контейнерах-списках. Эти итераторы также поддерживают арифметические операции с указателями. Для элементов вектора можно выполнять операции вставки и удаления, но быстродействие не является сильной стороной вектора. Вектор имеет смысл использовать в тех случаях, когда количество элементов контейнера изменяется не очень часто.

Что получится, если “скрестить” список с вектором? Мы получим двунаправленную очередь, или дек (`deque`). Возможности дека можно получить практически за счет простого объединения перечисленных выше возможностей списка и вектора. Так же, как и в списке, в деке очень эффективно выполняются операции вставки и удаления, особенно с начала или с конца (в середине дека они выполняются не так эффективно). Однако так же, как при работе с векторами, в деке применяются итераторы с произвольным доступом к элементам контейнера; таким образом, возможен доступ к элементам по индексу.

Во всех перечисленных типах контейнеров поддерживаются описанные ниже члены-функции, предназначенные для добавления новых элементов в контейнер или удаления имеющихся там элементов.

В приводимых примерах с помощью директивы `typedef` объявлен тип `Container`, который будет контейнером переменных типа `int`. Этот пример кода с таким объявлением типа будет успешно скомпилирован, потому что вызываемые функции предусмотрены во всех трех типах контейнеров. Эти примеры предельно просты, поэтому трудно отдать предпочтение какому-либо из них, и задача выбора оптимального контейнера решается очень просто!

Итак, вот как выглядят эти объявления:

```
typedef std::list<int> Container;
typedef std::vector<int> Container;
typedef std::deque<int> Container;
```

Теперь перейдем к объявлению функций для вставки элементов.

```
void push_back(const T&x);
void push_front(const T&x);
```

Функции `push_back()` и `push_front()` используются для вставки элементов с начала или с конца контейнера. Вставленное значение является копией `x`. (Помните, что полное описание назначения и синтаксиса этих функций можно найти в файлах оперативной справочной службы C++Builder.) В приведенном ниже примере кода показано, как инициализировать контейнер целочисленных переменных `Container` и вставлять в него значения. В комментариях указан порядок элементов контейнера `Container` после выполнения каждой операции.

```
Container ints; // Конструирование пустого объекта Container
ints.push_back(1);  // { 1 }
ints.push_back(2);  // { 1, 2 }
ints.push_front(3); // { 3, 1, 2 }
```

Функция `insert()` может использоваться для вставки элемента в любом месте списка.

```
iterator insert(iterator pos, const T&x);
```

Функция `insert()` принимает аргумент `pos` типа `iterator`, указывающий на элемент, перед которым будет вставлен новый элемент. При этом функция возвращает итератор, который указывает на вновь добавленный элемент.

В приведенном ниже коде приведено продолжение предыдущего примера с применением оператора `insert()` для вставки элементов в середине контейнера.

```
// Получение итератора, который указывает на первый элемент.
Container::iterator pos = ints.begin();
advance(pos, 1); // перемещение итератора к следующему элементу
//Вставка двух новых элементов.
pos = ints.insert(pos, 4); // { 3, 4, 1, 2 }
ints.insert(pos, 5);       // { 3, 5, 4, 1, 2 }
```

На заметку

Обратите внимание: если с помощью ключевого слова `typedef` объявить тип контейнера, например `typedef std::list<int>Container`, как это было сделано в примерах членов-функций контейнера, то можно ссылаться на вложенный в контейнере тип, например итератор, с помощью этой квалификации типа. Т.е. вместо ссылки на класс итератора `std::list<int>::iterator`, можно использовать ссылку `Container::iterator`. Такая ссылка не только имеет более простой вид, но и упрощает сопровождение кода, если вместо контейнера `list` выбираются другие типы контейнеров.

Для удаления элементов с начала и с конца списка могут использоваться, соответственно, функции `pop_back()` и `pop_front()`.

В нашем примере элементы списка могут быть удалены следующим образом:

```
ints.pop_front(); // { 5, 4, 1, 2 }
ints.pop_back();  // { 5, 4, 1 }
```

Кроме того, для удаления элементов из середины контейнера `Container` можно использовать функцию `erase()`.

```
iterator erase(iterator pos);
```

Она удаляет элемент в соответствии с указанной позицией `pos`, а итератор возвращает указатель на следующий элемент. В следующем коде показано, как удалить второй элемент контейнера `Container` и изменить значение следующего элемента.

```
pos = ints.begin();
advance(pos, 1);
pos = ints.erase(pos); // { 5, 1 }
*pos = 6;              // { 5, 6 }
```

Здесь рассмотрены лишь некоторые из наиболее часто используемых функций, а также продемонстрировано, как с их помощью выполняются основные операции с данными.

Контейнеры set и map

Как уже говорилось, шаблоны классов list, vector и deque несколько отличаются от set и map. Конечно, между всеми типами контейнеров есть сходства и различия, но основной критерий при выборе контейнера формулируется следующим образом: что характерно для элементов контейнера — уникальность значений и быстрота доступа к ним или поддержание заданного упорядочения?

Если ответом будет: "Важны все эти параметры", то следует использовать контейнеры set или map. Если же ответ: "Важна только быстрота доступа, а другие качества необязательны", то следует отдать предпочтение контейнерам list, vector или deque.

Контейнер set следует рассматривать как список, элементы которого уже отсортированы. При вставке новых элементов в контейнер set они всегда размещаются в соответствии с заданным порядком сортировки. А иначе их вставить просто не удастся. Для ускорения операции вставки значения можно использовать итератор, но в любом случае оно будет вставлено в соответствии с порядком сортировки. Вследствие строгого упорядочения элементов контейнера проверка наличия элемента в контейнере выполняется очень быстро.

Контейнер map является ассоциативным контейнером, в котором хранятся пары ключ-значение. Напомним, что для использования любого шаблона класса, например map, необходимо сначала инициализировать его. Иначе говоря, необходимо сообщить компилятору о тех типах объектов, с которыми будет работать конкретный экземпляр класса. В случае контейнера map необходимо указать два типа: тип ключа, используемого для индексации элементов контейнера, и типы хранимых в контейнере значений.

Еще один шаблон класса библиотеки SCL — контейнер pair, содержащий два открытых члена-элемента данных, first и second, типы которых необходимо указать при специализации шаблона. Элементы контейнера map являются экземплярами типа pair, для которых заданы те же типы, что и для членов контейнера map, где член first является ключом, а член second — значением. Ключи контейнера map всегда содержат уникальные значения. Более того, так же, как и в контейнере set, элементы контейнера pair всегда остаются в порядке сортировки (заданном по значениям ключа). Контейнер map использует двунаправленные итераторы, которые указывают на расположение пар элементов. По известному значению итератора i доступ к ключу или значению элемента можно получить, используя конструкции i->first или i->second.

Способ упорядочения элементов контейнеров set и map определяется одним из параметров шаблона. Например, объявление шаблона для контейнера set выглядит так:

```
template <class Key, class Compare = less<Key>,
class Allocator = allocator<Key> >
class set ;
```

Второй параметр, класс Compare, используется для сравнения двух значений. По умолчанию для него используется шаблон класса less библиотеки SCL, специализированный типом первого параметра (с псевдонимом Key). Шаблон класса less использует

оператор меньше (<) и является специальным типом шаблона, который создает *функциональный объект* (*function object*), или *функционал* (*functional*), или *функтор* (*functor*). Вот его объявление:

```
template <class T>
struct less : binary_function<T, T, bool> {
   bool operator () (const T& a, const T& b) const;
};
template <class T>
bool less<T>::operator () (const T& a, const T& b) const
{
   return (a<b);
}
```

Функциональные объекты используют перегруженный оператор () таким образом, что его экземпляры могут применяться синтаксически так же, как если бы они были функциями. Они являются наиболее мощным заменителем старомодных указателей функций. Далее при рассмотрении алгоритмов читатель сможет убедиться в том, что функциональные объекты используются довольно часто.

Для объявления множества set переменных типа int, которые отсортированы в порядке убывания, а не возрастания, можно использовать шаблон класса greater библиотеки SCL:

```
std::set<int, std::greater<int> > intset;
```

Для вставки значений в контейнер set или map используется функция-член insert(), а для поиска некоторого значения по ключу — итератор find.

```
pair<iterator,bool> insert(const value_type&x);
iterator find(const key_type&x);
```

key_type — это тип ключа, определенный с помощью ключевого слова typedef. В контейнере set типы аргументов value_type и key_type идентичны. В контейнере map аргумент value_type имеет тип pair, содержащей значение и его ключ. Простейший способ создания объекта типа pair основан на использовании шаблона функции make_pair, которая принимает в качестве аргумента два значения и возвращает объект pair, содержащий эти значения. Функция insert() возвращает объект pair типа pair<iterator,bool>. Если вставленный объект pair уже имеется в контейнере set или map, то возвращаемый объект pair будет содержать итератор со значением указателя "вслед за последним элементом" и логическое значение false. А если такого элемента нет и элемент вставлен, то возвращаемый объект pair будет содержать итератор с указателем на вставленную пару и логическое значение true. Другой более читабельный метод добавления значения в контейнер map основан на перегрузке оператора []. Оба эти метода показаны в примере кода в листинге 4.3. Теперь смысл функции find() достаточно просто уяснить: она возвращает итератор с указателем на найденный элемент с ключом x или итератор с указателем "вслед за последним элементом", если этот элемент не найден.

Листинг 4.3. Использование контейнера map

```
typedef std::map<std::string, std::string> ssmap;
ssmap days;
days.insert(make_pair(std::string("Mon"), std::string("Monday")));
days["Tues"] == "Wednesday"; //Что?!! Это, не верно!
days["Tues"] == "Tuesday";   //Правильно, исправим!
```

```
//Добавлен ли элемент со значением Thursday?
ssmap::iterator i = days.find("Thurs");
if(i == days.end())
{
   // Добавить элемент со значением Thursday, если его нет.
   // (На самом деле нам известно, что его нет!)
days["Thurs"] == "Thursday";
}
std::cout << "Введите сокращенное название дня недели:";
char input [32];
std::cin >> input;
if(days.find(input) != days.end())
{
   std::cout << "Полное название дня недели " << days[input ] << std::endl;
}
else
{
   std::cout << "Извините, нет такого названия дня недели." << std::endl;
}
```

Здесь представлены только основные сведения о контейнерах, хотя о них можно рассказать гораздо больше.

Стандартные алгоритмы

Выше уже рассматривались преимущества и недостатки различных типов контейнеров. Могут оказаться полезными также описания интерфейсов классов, которые приводятся в файлах оперативной справочной службы. Например, шаблон класса `list` содержит функцию-член для сортировки элементов, а в шаблоне класса `vector` ее нет. Это объясняется тем, что класс-список `list` предназначен для сортировки и имеет собственную подпрограмму сортировки. С другой стороны, в классе-векторе `vector` сортировка элементов не предусмотрена. Но если такая необходимость возникнет, то можно использовать способ, который предусмотрен в библиотеке SCL.

Алгоритмами называются шаблоны функций, которые не являются членами-функциями и используются для выполнения стандартных операций с контейнером. Один из таких алгоритмов `sort` применяется для сортировки диапазона (или всех) элементов контейнера. Его было бы неэффективно применять для сортировки всего списка, потому что в списке имеется собственная функция, которая гораздо быстрее справляется с этой задачей. Пустой тратой времени была бы также сортировка уже отсортированного контейнера, например `set` или `map`. Но во многих случаях применение алгоритма `sort` является простейшим способом сортировки элементов.

Во многих контейнерах используются два итератора, определяющие диапазон элементов контейнера, к которому применяется алгоритм. По умолчанию диапазон включает первый итератор и каждый последующий элемент, за исключением второго итератора. Предполагается (но не проверяется), что второй итератор располагается за первым. Соответственно, члены-функции `begin()` и `end()` контейнера определяют диапазон, состоящий из всех элементов контейнера.

В предыдущем разделе было представлено понятие функционального объекта, который используется как аргумент во многих алгоритмах.

В качестве примера рассмотрим следующий боле сложный алгоритм сортировки с тремя параметрами: двумя итераторами и функциональным объектом. Такой алгоритм сортировки выглядит следующим образом:

```
template <class RandomAccessIterator, class Compare>
void sort (RandomAccessIterator first,
RandomAccessIterator last, Compare comp);
```

Параметры `first` и `last` указывают диапазон сортируемых элементов. Параметр `comp` соответствует функциональному объекту, который сравнивает два элемента контейнера и возвращает `true`, если элемент `first` расположен перед элементом `second` в порядке сортировки, либо `false`, в противном случае. Рассмотрим приведенный ниже контейнер-дек со строками.

```
std::deque<string> strings;
```

Алгоритм сортировки может быть основан на длине строк; например, чем длиннее строка, тем ближе к началу дека она располагается. Для этого следует использовать следующий функциональный объект:

```
struct length_compare {
   bool operator () (const string& a, const string& b) const
   {
      return a.length() > b.length();
   }
};
```

С помощью этой структуры определяется функциональный объект, который принимает две строки и возвращает `true`, если первая строка длиннее второй. Для сортировки всего контейнера со строками можно вызвать функцию `std::sort`, создав ее с помощью конструктора и одновременно передав в качестве параметра функциональный объект `length_compare`:

```
std::sort(strings.begin(), strings.end(), length_compare());
```

Такой контейнер теперь будет отсортирован. Для проверки этого соберите проект `SCLProg.bpr`, который находится в каталоге `SCLProg` на прилагаемом к этой книге компакт-диске. Полученное приложение предложит пользователю протолкнуть строки в дек, а затем отсортирует дек с помощью функционального объекта `length_compare`. Эта программа также демонстрирует способы использования алгоритма `for_each` для передачи отсортированного списка обратно в форму или записи его в физическом файле.

Для выполнения различных задач предусмотрено большое количество разнообразных алгоритмов. Полный список таких алгоритмов, имеющихся в стандартной библиотеке Standard C++ Library, можно найти в оперативной справочной службе (ищите по слову algorithms).

Заключительные замечания по поводу библиотеки SCL

Автору удалось описать всего лишь некоторые особенности работы с библиотекой SCL. Это значит, что предусмотренные в ней возможности охватывают гораздо больший диапазон возможных приложений.

Оперативная справочная служба может служить достаточно полным справочным руководством по работе с библиотекой SCL. В ней есть несколько разделов с простыми и понятным изложением материала и комментариями к листингам и другим справочным материалам. После знакомства с этой документацией читатель, несомненно, узнает много нового.

Интеллектуальные указатели и строгие контейнеры

В этом разделе рассматривается метод выделения памяти в куче и ее автоматического освобождения, т.е. метод, который может существенно сократить вероятность утечки памяти и упростить создание и сопровождение кода. Он не является особенностью только среды C++Builder и может применяться при работе с любым компилятором, в котором предусмотрена поддержка шаблонов. Эта тема имеет очень большое значение, поскольку каждый программист на языке C++ должен уметь пользоваться таким методом.

Куча и стек

Говоря упрощенно, *стеком* (*stack*) называется память, управляемая компилятором как частью окружения программы. Как программист, автор предпочитает использовать стек, поскольку при работе с ним не нужно заботиться о его освобождении, даже в случае возникновения исключительной ситуации. Дело в том, что он автоматически освобождается при каждом выходе программы за пределы текущей области действия. То есть, компилятор всегда автоматически освобождает память, выделенную в стеке.

Кучей (*heap*) называется память, непосредственно управляемая программой. Она обеспечивает большую гибкость, но отслеживание всех операций с памятью может оказаться очень трудной задачей, потому что, в отличие от стека, за освобождение всей выделенной в куче памяти отвечает программист. Если какая-то часть выделенной памяти не освобождена, то в таком случае говорят, что в программе имеет место утечка памяти (leak memory).

Указатели

В языке C++ для ссылки на выделенную в куче память и работы с ней используются указатели. К сожалению, указатель не обеспечивает средств для автоматического освобождения памяти, на которую она ссылается. Таким образом задача освобождения каждого байта целиком ложится на плечи программиста. При работе с современными сложными программами эту задачу гораздо легче сформулировать, чем выполнить.

Интересно, что память для указателей обычно выделяется в стеке, а потому они удаляются автоматически. Как будет показано ниже, из этой особенности тоже можно извлечь определенную пользу.

Строгие указатели

Существует очень простой метод связывания кучи со стеком. Это значит, что при установлении такой связи выделенная в куче память будет освобождаться автоматически при очистке стека, причем гарантированно! Вот так работает этот метод. Вместо использования регулярного указателя для ссылки на выделенную память следует использовать *строгий указатель* (*strong pointer*). Он состоит из простого шаблона класса, деструктор которого освобождает память в куче. Поэтому при автоматическом удалении строгого указателя при очистке стека также автоматически будет освобождена память в куче, на которую ссылается этот указатель. Кроме того, следует переписать оператор разыменования для того, чтобы класс вел себя так же, как регулярный неформатированный указатель. Поскольку строгий указатель основан на шаблоне, то может указывать на объект любого типа, а потому охватывает практически все способы выделения памяти. В листинге 4.4 показана сердцевина такого шаблона класса `auto_ptr`.

Листинг 4.4. Шаблон класса для строгого указателя `auto_ptr`

```
auto_ptr<X>::auto_ptr( X*p )
{
    the_p =p ;
}
auto_ptr<X>::~auto_ptr()
{
    delete the_p ;
}
X*auto_ptr<X>::operator ->()
{
    return the_p ;
}
X&auto_ptr<X>::operator *()
{
    return *the_p ;
}
```

Это действительно очень простой пример, но, как говорится, мал золотник, да дорог. Вот пример его использования.

```
auto_ptr<TLabel> myLabel(new TLabel(this)) ; //Строгий указатель
myLabel->Parent = this ;
myLabel->Caption = "This is a label" ;
```

На заметку

Небольшое напоминание: `auto_ptr`, как и любой другой класс библиотеки SCL, является частью пространства имен `std`. Для компиляции предыдущего кода необходимо либо вставить директиву `using namespace std;` в самом начале кода или использовать префикс `std::`. Кроме того, не забудьте включить в код файл `memory.h`, который находится в подкаталоге `Include` каталога C++Builder.

Самое важное заключается в том, что память для надписи теперь необязательно освобождать вручную. Во-вторых, строгий указатель используется так же, как и обычный указатель. В-третьих, строгие указатели могут содержать только указатели, а потому тип шаблона относится только к типу указуемого объекта.

В таких случаях говорят, что строгий указатель *владеет* памятью или объектом, на который он указывает. Важно понимать, что созданный объект не может существовать дольше строгого указателя. Строгий указатель должен иметь достаточно большую область действия, необходимую для выполнения задач, связанных с объектом.

Класс `auto_ptr` проще всего используется, когда объект (или выделенная в куче область памяти) необходим только на время жизни строгого указателя, который на него указывает. Иначе говоря, если строгий указатель и память на которую он указывает, могут быть удалены автоматически в конце области действия. Конечно, его можно использовать, даже если память имеет меньшую продолжительность жизни, чем область действия. В этом случае нужно просто сообщить строгому указателю об освобождении памяти с помощью функции `reset()`. Это используется для обеспечения безопасности кода при исключительных ситуациях.

Только один строгий указатель может владеть памятью, на которую он указывает. Это имеет смысл, поскольку память может быть освобождена только один раз. В результате при-

своения одного строгого указателя другому передается ему право владения указанной памятью. Эта особенность может показаться очень странной, потому что программисты привыкли ассоциировать операцию присвоения с операцией копирования, но в данном случае после присвоения строгий указатель изменяется, как показано в приведенном ниже примере.

```
auto_ptr<TLabel> label1( new TLabel(this) );
auto_ptr<TLabel> label2 ;
label2 = label1 ; // Права владения переданы.
```

После выполнения этого кода указатель `label2` указывает на объект `TLabel` и владеет им, тогда как указатель `label1` ничем не владеет и указывает на неопределенное значение (`NULL`).

Рассмотрим первый пример в контексте C++Builder. Создадим новое приложение, а в нем — кнопку и присоединим к ней следующий обработчик события `OnClick`.

```
void TForm1::Button1Click( TObject*Sender )
{
    static int counter;

    auto_ptr<TLabel> myLabel( new TLabel(this) ) ; // Строгий указатель.
    myLabel->Parent = this ;
    myLabel->Name = "Label" ++String(++counter) ;
    myLabel->Top = 10 ;
    myLabel->Left = 10 ;
    myLabel->Caption = "This is a label named " ++myLabel->Name ;
}
```

Приводит в изумление отсутствие надписи в форме после выполнения этого кода. Дело в том, что она была удалена строгим указателем после очистки стека в конце метода.

Для правильной работы строгий указатель должен иметь область действия класса, как это организовано при работе с обычным указателем. Кроме того, для обеспечения безопасности при возникновении исключительных ситуаций следует использовать только строгие указатели. Вот еще один вариант обработчика события `OnClick`:

```
// В заголовочном файле:
class TForm1 :public TForm
{
//...
private:
    auto_ptr<TLabel>classLabel ;
//...
};

// В файле с исходным кодом:
void TForm1::Button1Click(TObject*Sender )
{
    static int counter =0 ;

    // Строгий указатель
    auto_ptr<TLabel> myLabel( new TLabel(this) ) ;

    myLabel->Parent = this ;
    myLabel->Name = "Label" ++String(++counter) ;
    myLabel->Top = 10 ;
```

```
    myLabel->Left = 10 ;
    myLabel->Caption = "This is a label named " +
                       myLabel->Name ;

    classLabel = myLabel ; // Переход в область действия класса
}
```

Теперь после каждого щелчка на кнопке будет создан новый объект `TLabel`. После присвоения его указателю `classLabel` строгий указатель сначала освободит объект, которым он ранее владел, а затем вступит во владение новым объектом. И никакой утечки памяти!

Читатель, возможно, удивится, узнав, что этот шаблон класса уже является частью стандарта C++. Так как пакет C++Builder полностью совместим с ним, то он содержит этот класс в заголовочном файле `memory.h` в каталоге `Include` пакета C++Builder. Познакомьтесь с ним, чтобы убедится в его простоте и узнать о его методах.

На этом хорошие новости завершаются, а плохие — заключаются в том, что комитет по стандартизации на этом этапе прекратил свою работу, не предоставив полного решения этой задачи. Действительно, класс `auto_ptr` едва ли можно назвать стандартным. Однако наряду с ним имеется еще два класса, которые могут удовлетворить запросы разработчиков.

Класс `auto_ptr` содержит метод `get()`, который следует использовать с чрезвычайной осторожностью, так как он возвращает прямую ссылку на область памяти, которой владеет указатель. Как уже говорилось выше, только один строгий указатель может владеть выделенной в куче памятью. А данный метод позволяет создать другой строгий указатель, который указывает на эту память и владеет ею. Он также может использоваться для высвобождения памяти, которой владеет строгий указатель.

Вероятно, именно по этой причине класс `auto_ptr` был признан противоречивым и включен в стандарт только в последний момент. Как видите, этого нельзя избежать, поскольку должен существовать способ передачи неформатированного указателя вызовам API-функций, в которых он используется. При использовании метода `get()` убедитесь в том, что он вызывается только для тех функций, которые не освобождают эту область памяти. Если все же функция освобождает выделенную область памяти, то программисту придется позаботиться о том, чтобы вызвать метод `release()` еще до использования строгого указателя или до выхода из области действия и попытки снова освободить эту память.

Интеллектуальные указатели

Интеллектуальные указатели созданы на основе строгих указателей и предназначены для того, чтобы несколько указателей могли ссылаться на один и тот же объект. Это достигается за счет совместного использования интеллектуальными указателями контейнера `set` библиотеки SCL, в котором они ссылаются друг на друга, причем память будет освобождена только при удалении последнего умного указателя. Оставляя в стороне подробности реализации этой конструкции, достаточно сказать, что работает она просто замечательно.

Важным следствием этого является то, что выделенная память уже не связана с областью действия первого умного указателя, который владеет ею. Ее время жизни может варьироваться при каждом запуске программы, в зависимости от того, когда удаляется интеллектуальный указатель — владелец этой области памяти.

Этот шаблон класса используется аналогично классу `auto_ptr`. Однако его определение имеет одно важное отличие: если значение одного умного указателя передается другому, то права владения используются ими совместно.

Предыдущее предупреждение об использовании метода `get()` в строгих указателях в равной степени относится и к интеллектуальным указателям. Неаккуратное применение может привести к опустошительным результатам, а потому его следует использовать с чрезвычайной осторожностью.

Более того, класс `SmartPointer`, который находится на прилагаемом к этой книге компакт-диске, надежен и безопасен при работе с потоками (файлы `SmartPointer.cpp` и `SmartPointer.hpp`). Благодаря этому он чрезвычайно эффективен при создании многопоточного кода. Представьте себе всю сложность освобождения памяти, совместно используемой несколькими потоками, если программисту неизвестно, какой поток завершает работу последним. Этого кошмара можно избежать, использовав класс `SmartPointer`.

Помимо упомянутых различий, этот класс используется аналогично классу `auto_ptr`. Рассмотрим следующий простой пример:

```
SmartPointer<TLabel> label1( new TLabel(this) );
SmartPointer<TLabel> label2 ;
label2 = label1 ; // Права владения используются совместно
```

После выполнения этого фрагмента кода, интеллектуальные указатели `label1` и `label2` совместно владеют одним объектом и ссылаются на него. Любой из этих интеллектуальных указателей может быть удален первым.

Строгий контейнер

Другой наиболее распространенной задачей является управление динамическими объектами как одной группой. Опытный программист на языке C++ знает, насколько трудно выполнить эту задачу, не применяя стандартную библиотеку Standard C++ Library (SCL). Создание строгого контейнера без утечки памяти, аналогичного вектору библиотеки SCL, представляет собой более сложную задачу, чем создание класса строгого указателя. Дело в том, что функциональность вектора должна при этом дублироваться. На прилагаемом к этой книге компакт-диске содержится пример такого контейнера — `StrongVector` (см. файлы `StrongVector.cpp` и `StrongVector.hpp`).

Более того, класс `Vector` библиотеки SCL и класс `StrongVector` работают практически одинаково, а потому на изучение последнего не потребуется тратить дополнительное время. Библиотека SCL кратко рассматривается в начале этой главы.

В листингах 4.5 и 4.6 показан простой пример применения контейнера-вектора библиотеки SCL и класса `StrongVector`.

Листинг 4.5. Применение контейнера `Vector` из библиотеки SCL

```
public:
   typedef vector<TLabel*> LABELS ;
   LABELS labels ;

TForm1::TForm1( TComponent* Owner )
{
    labels.push_back(new TLabel(this));
}

TForm1::~TForm1()
{
```

```
    LABELS::iterator i ;
    i = labels.begin() ;
    while( i++ < labels.end() )
        delete *i ;
}
```

Листинг 4.6. Применение контейнера `StrongVector`

```
public:
   typedef StrongVector<TLabel> LABELS ;
   LABELS labels ;

TForm1::TForm1( TComponent* Owner )
{
    auto_ptr<TLabel> label( new TLabel(this) );
    labels.push_back( label );
}
```

Это сравнение позволяет обнаружить различия между способами применения этих контейнеров. Во-первых, обратите внимание на то, что при объявлении строгого контейнера и строгого указателя, указатель не является типом шаблона. Дело в том, что эти шаблоны владеют указателями, поэтому приведение их типа к указателю происходит автоматически. Во-вторых, строгий указатель используется для передачи надписи строгому вектору. Это необходимо, чтобы гарантировать цельность цепочки строгих указателей. Наконец, при использовании контейнера `StrongVector` не потребуется освобождать память явным образом.

Библиотека SCL содержит не только контейнеры-векторы, но и многие другие конструкции. Вряд ли стоит заниматься их дублированием. К счастью, это и не нужно. Мощный потенциал библиотеки SCL можно эффективно использовать, если предварительно присвоить новым объектам тип `StrongVector` для обеспечения большей безопасности. После этого работу с обычными (нетипизированными) указателями можно продолжить, используя любые контейнеры библиотеки SCL. В листинге 4.7 предлагается простой пример такого способа применения контейнера `StrongVector` библиотеки SCL.

Листинг 4.7. Применение контейнеров библиотеки SCL на основе контейнера `StrongVector`

```
public:
    typedef StrongVector<TLabel> CONTAINER ;
    CONTAINER container ;

    typedef set<TLabel> LABELS ;
    LABELS labels ;

TForm1::TForm1( TComponent*Owner )
{
    CONTAINER::iterator i ;
    for( int ctr=0; ctr <10; ++ctr )
    {
        // Присвоение надписей строгому контейнеру
        auto_ptr<TLabel> label( new TLabel(this) );
```

```
        container.push_back( label );

        // Теперь любой контейнер библиотеки SCL
        // может безопасно использоваться
        i =container.end();
        labels.push_back( *i );
    }
}
```

Возможные ловушки

Несмотря на перечисленные достоинства строгих и интеллектуальных указателей, всегда следует помнить о том, что они не являются обычными указателями. Поэтому, работая с ними, следует остерегаться, чтобы попасть в потенциальные ловушки.

- Никогда не применяйте строгие/интеллектуальные указатели для ссылки на область памяти в стеке. Представьте себе, какая путаница произойдет при попытке освободить эту область памяти.
- Никогда не инициализируйте строгие/интеллектуальные указатели с помощью указателя `this`. Это закончится автоматическим удалением вашего объекта и некоторых других.
- Никогда не передавайте строгий указатель в качестве параметра, поскольку это связано с двумя побочными эффектами. Во-первых, строгий указатель как параметр передаст права владения экземпляру в функции или методе. Во-вторых, освобождение памяти произойдет после завершения этой функции или метода. Это не относится к интеллектуальным указателям.
- Никогда не передавайте разыменованный интеллектуальный указатель другому потоку. Это приведет к разрыву цепочки. Например, поток, завершивший свою работу первым, освободит память, при этом ссылка на нее во втором потоке будет недействительной.
- Использование строгих/интеллектуальных указателей означает утрату наследуемого полиморфизма. Другими словами, основной класс `auto_ptr<Base>` нельзя использовать вместо производного класса `auto_ptr<Derived>`.

Заключительные замечания об интеллектуальных указателях и строгих контейнерах

С помощью представленных здесь классов можно без излишних затрат времени создавать более надежные и стабильные приложения. Благодаря строгим указателям/векторам можно покончить с практикой насыщения кода многочисленными конструкциями `try...catch`. А при использовании класса `SmartPointer` в потоках вам не придется заботится о том, чтобы все потоки были осведомлены о других потоках, которые совместно с ними используют одну область памяти.

Парадигма строгого управления памятью позволяет экономить время при создании безопасного в отношении исключительных ситуаций кода, а также при поиске причин утечки памяти. Сопровождение кода является важной составляющей всех расходов, поэтому читабельность кода имеет очень большое значение. Таково побочное влияние уменьшения количества строк, необходимых для обеспечения устойчивой работы приложения при возникновении исключительной ситуации. Конструкций `try ... catch` избежать нельзя, но их не следует использовать для управления кучей.

Несмотря на удобство и мощь классов `auto_ptr`, `StrongPointer` и `StrongVector`, важно понимать, что они подходят не для всех ситуаций. Внимательно изучите описанные выше ловушки, чтобы не попасть в них. Например, несмотря на то что интеллектуальный указатель можно передавать как параметр, следует учесть, что его частое использование сопровождается существенными накладными расходами.

На заметку

Более подробную информацию по этому поводу можно найти в следующих двух статьях.

- Roubal, E. "Templatized Managed/Smart Pointers," *C++Report*, February 1998 Vol10/No2:23–28.
- Milewski, B. "Strong Pointers and Resource Management in C++," *C++Report*, September 1998 Vol10/No8:23–27.

Статья Рубала открыла для меня множество новых фактов об интеллектуальных указателях! Надеюсь, что не только автор, но и читатели смогут извлечь для себя пользу, применяя этот компонент объектно-ориентированного подхода.

Усовершенствованные обработчики исключительных ситуаций

Обработка исключительных ситуаций (exception handling) — это механизм перехвата, передачи и обработки исключительных ситуаций в приложении. Иногда использование этого механизма может иметь обратный эффект, поэтому программистам необходимо внимательно изучить средства обработки исключительных ситуаций. Эти темы являются фундаментальными при описании языка и подробно рассматриваются в главе 3, а также в книгах об основах языка C++. А в этом разделе предполагается, что читатель уже знаком с основными концепциями обработки исключительных ситуаций.

Обработчик исключительных ситуаций (exception handler) — это класс или модуль приложения, предназначенный для обработки исключительной ситуации. В этом разделе представлен класс, который может использоваться программистами C++Builder для получения отчета о возникновении исключительной ситуации. Кроме того, этот класс автоматически перехватывает и сообщает о возникновении исключительных ситуаций, обработка которых не предусмотрена непосредственно в программе.

Обзор стратегии

Начнем с краткого обзора стратегии обработки исключительных ситуаций, которая применяется в работе представленного здесь обработчика исключительных ситуаций. Во-первых, необходимо обезопасить код программы от возникновения таких ситуаций. Это значит, что исключительная ситуация не должна дестабилизировать работу программы или приводить к утечке ресурсов. Особенно внимательно следует отнестись к конструкторам, циклам и выделению памяти.

Во-вторых, следует организовать обработку исключительных ситуаций в функции `WinMain()`. Программа C++Builder может инициировать любое событие только во время обработки сообщений. Обработка сообщений начинается с вызова `Application.Run()`, поэтому обработчик исключительных ситуаций не будет вызван для исключительных ситуаций, возникающих в `WinMain()`. Этот код должен находиться в блоке `try` или `catch`. Все каркасы приложений C++Builder выполняют это автоматически. Для просмотра каркаса вашего приложения выберите команду меню Project⇨View Source.

В-третьих, в особых случаях нужно организовать перехват исключительных ситуаций, например когда не требуется отображать сообщение для пользователя или нужно разрешить эту проблему иначе или в каком-то другом месте. О других ситуациях беспокоиться не следует, и именно в этом заключается преимущество модели обработки исключительных ситуаций.

Наконец, нужно заменить предлагаемый по умолчанию (со стороны компилятором) обработчик исключительных ситуаций тем, который описывается ниже (т.е. классом `ErrorHandler`).

На заметку

В предыдущей главе представлены способы использования ключевых слов `try`, `catch` и `__finally`. Если вы не знакомы с основными принципами обработки исключительных ситуаций, то вернитесь назад и еще раз тщательно прочтите эту главу. Дело в том, что при изучении способов замены предлагаемого по умолчанию обработчика исключительных ситуаций нельзя обойтись без базовых знаний об исключительных ситуациях и их обработке.

Обзор преимуществ

Предлагаемый способ обработки исключительных ситуаций обладает следующими преимуществами по сравнению с предусмотренным по умолчанию.

- Вся доступная информация записывается в файл для последующего просмотра сотрудниками, выполняющими поддержку кода.
- Создается снимок состояния системы в целом. Он включает информацию об исключительной ситуации, объекте, который вызвал ее появление, об активной форме, приложении и об операционной системе Windows.
- Учитывается информация о формах. Это очень важно, поскольку обеспечивается контекст исключительной ситуации. Просмотр предпринимаемых пользователями действий в момент возникновения исключительной ситуации облегчает поиск возникшей в программе ошибки.
- Появляется возможность удовлетворить специфические для приложения требования.
- Повышается интеграция с приложением. Например, сообщение об ошибке может как отображаться в специально подготовленной форме, так и фиксироваться без отображения на экране.
- Имеется возможность регистрировать сообщения, которые не являются результатом исключительных ситуаций, и создавать отчеты о них.

Замена предлагаемого компилятором обработчика

Класс `ErrorHandler` можно разместить в блоке `catch` всех конструкций `try ... catch` приложения. Но на самом деле нам требуется такой вездесущий обработчик исключительных ситуаций, который мог бы обрабатывать даже неперехватываемые прежде ситуации. В конце концов, большинство исключительных ситуаций возникает неожиданно. То есть, нужно указать C++Builder на использование класса `ErrorHandler` при возникновении любой исключительной ситуации.

Основные принципы

В C++Builder предусмотрен предлагаемый по умолчанию обработчик исключительных ситуаций. Однако он выполняет только основные функции, т.е. только выводит сообщение об

ошибке в модальном окне. К счастью, создатели библиотеки VCL предусмотрели простой и легкий способ его замены: нужно лишь назначить в качестве обработчика `TApplication::OnException`. В этом случае предложенный обработчик сможет вызвать класс `ErrorHandler` для обработки всех неперехваченных исключительных ситуаций. Ниже в этой главе будет рассмотрен пример использования этого класса.

Дочерние формы многодокументного интерфейса

Для дочерних форм многодокументного интерфейса (у которых свойство стиля `FormStyle` имеет значение `fsMDIChild`) не требуются никакие специальные действия. Дело в том, что обработчик "знает", где получить нужную информацию, включая активную дочернюю форму многодокументного интерфейса.

Формы однодокументного интерфейса или модальные формы

Для форм любого стиля, за исключением форм многодокументного интерфейса (у которых свойство стиля `FormStyle` имеет значение `fsMDIChild`), следует создать отдельный обработчик исключительных ситуаций. В случае необходимости обработчик может получить очень полезную информацию о форме, которая недоступна при использовании других способов. Этого можно добиться такими способами.

1. Создайте обработчик исключительных ситуаций для формы и организуйте в нем вызов функции `ErrorHandler::LogEntry()`. Используйте указатель `this` в качестве последнего параметра, чтобы этот метод включал информацию о форме. Пример использования этого метода `ErrorHandler::DefaultExceptionHandler()` представлен в листинге 4.12.
2. В обработчике события `OnActivate` формы используйте описанный ранее способ для создания нового обработчика на основе метода, описанного в пункте 1. В результате библиотека VCL будет использовать его для обработки неперехваченных исключительных ситуаций во время активности формы. Пример конструктора `ErrorHandler` приведен в листинге 4.9.
3. В обработчике события формы `OnDeactivate` восстановите предыдущий обработчик событий или задайте значение `NULL` для `TApplication::OnException`. Если обработчик не задан, то C++Builder будет использовать обработчик, предлагаемый по умолчанию.

При выводе формы на основе базовой формы этот процесс потребуется выполнить только один раз в базовой форме. Рассмотрим создание базовой формы, на основе которой создаются другие формы проекта.

Опустим подробное описание мотивов использования этих мер. `ErrorHandler::DefaultExceptionHandler()` заменяет применяемый по умолчанию обработчик исключительных ситуаций. Одним из параметров этой функции является форма, которая передается методу `ErrorHandler::LogEntry()`. Переменная формы используется для отслеживания специфической для формы информации. Как вы увидите позднее в этой главе, по умолчанию форма задается как активная дочерняя форма многодокументного интерфейса. Вот почему информация об однодокументных или модальных формах не отображается в журнале регистрации. Эту проблему можно разрешить с помощью трех перечисленных выше способов.

Другие классы

Для классов, которые не являются формами, например `TDataModule` или созданные вами собственные классы, ничего делать не требуется. Предполагается, что эти классы используются в контексте активной формы, в которой применяется ее собственный обработчик.

Добавление в класс специфической для проекта информации

Так как для каждого отдельного приложения существует только один экземпляр класса, то его модификацию можно выполнить непосредственно. Однако создание производных классов позволит повторно применить его в других проектах и эффективно использовать объектно-ориентированную модель. При создании собственного класса на основании существующего важно помнить, что инициализировать нужно именно производный, а не базовый класс. Более подробно этот вопрос описывается в следующем разделе.

Код обработчика событий

Теперь мы готовы приступить к рассмотрению класса `ErrorHandler`. В этом разделе после листинга каждого метода приводится его краткое описание. Для экономии места здесь не приводятся заголовочный файл и некоторые менее важные методы обработки исключительных ситуаций в стиле языка C. В полном виде этот код содержится в файлах `EHandler.cpp` и `EHandler.h` на прилагаемом к книге компакт-диске.

Внимательно изучите этот код. Вызовы API-функций и используемые в нем приемы могут быть применены в других частях приложения.

Например, метод `LogObjectState()` показывает, как получить значение свойства `DataSource` без приведения типа элемента управления.

Исходный код в листинге 4.8 покажется читателю знакомым, поскольку он почти такой же, как код в только что созданной форме C++Builder. Хотя этот класс не является формой, он выполняет много ее функций с помощью библиотеки VCL. Однако нетрудно заметить, что этот код отличается последней строкой, которая заслуживает особого внимания. Во-первых, `ErrorHandler` — это имя определяемого класса. Во-вторых, переменная `errorHandler_G` не является указателем, а фактической реализацией класса — обработчика исключительных ситуаций. Так как переменная объявляется в заголовке как внешняя, то обработчик исключительной ситуации будет доступен в глобальном масштабе сразу же после запуска программы. Напомним, что глобальные объекты реализуются еще до вызова функций или методов, даже функции `WinMain()`.

Листинг 4.8. Реализация класса

```
#include <vcl.h>
#pragma hdrstop
#include "EHandler.h"
//---------------------------------------------------------------
#pragma package(smart_init)
//---------------------------------------------------------------
ErrorHandler errorHandler_G ;
//---------------------------------------------------------------
```

В конструкторе, показанном в листинге 4.9, следует отметить следующие важные моменты. Во-первых, он вызывается после автоматической реализации переменной `errorHandler_G` во время запуска программы. Для построчного выполнения кода программы выберите команду меню Run⇨Trace to Next Source Line или нажмите комбинацию клавиш <Shift+F7>. Во-

вторых, с помощью функции OpenLogFile() обработчик откроет журнал регистрации, куда будут записаны все возникающие ошибки. Наконец, он заменяет предлагаемый по умолчанию обработчик исключительных ситуаций. Другими словами, добавление этого модуля в проект автоматически приведет в действие обработчик исключительных ситуаций.

Листинг 4.9. Замена предлагаемого по умолчанию обработчика исключительных ситуаций с помощью конструктора

```
__fastcall ErrorHandler::ErrorHandler(void )
  : ERROR_TITLE  ( "<<< ERROR INFORMATION >>>" ),
    MESSAGE_TITLE( "<<< MESSAGE >>>" ),
    SYSTEM_TITLE ( "< SYSTEM STATE >" ),
    PROGRAM_TITLE( "< APPLICATION STATE >" ),
    OBJECT_TITLE ( "< OBJECT THROWING EXCEPTION >" ),
    FORM_TITLE   ( "< FORM >" ),
    DETAIL_TITLE ( "< ERROR DETAILS >" ),
    DATA_TITLE   ( "< DATA >" ),
    log          ( NULL ),
    log_External ( NULL )
/*************************************************************\
* ACCESS:Public                                               *
\*************************************************************/
{
    // открыть файл регистрации
    OpenLogFile();

    // Заменить предлагаемый по умолчанию обработчик
    // исключительных ситуаций собственным обработчиком.
    previousExceptionHandler = Application->OnException ;
    Application->OnException = DefaultExceptionHandler ;
}// Конструктор
//-------------------------------------------------------------
```

В листинге 4.10 показан код деструктора. Сначала он проверяет отсутствие подмены его другим применяемым по умолчанию обработчиком исключительных ситуаций. Дело в том, что нет никакого смысла в применении двух обработчиков исключительных ситуаций. Далее, если не достигнут конец программы, снова используется обращение к обработчику исключительных ситуаций компилятора. Наконец, в конце деструктора освобождаются ресурсы класса.

Листинг 4.10. Деструктор

```
__fastcall ErrorHandler::~ErrorHandler()
{
    // Предупредить в случае ВОЗМОЖНОЙ ошибки при работе
    // обработчика ErrorHandler::DefaultExceptionHandler()
    // или попытке внешнего вмешательства и его подмены.
    if( Application->OnException == NULL )
       WriteToLog("***Class ErrorHandler was disabled. An error"
                  " may have occured in it.***");
//     WriteToLog("***Класс ErrorHandler не активен. Возможно,"
//                "в нем возникла исключительная ситуация.***");
```

```
    else if(Application->OnException != DefaultExceptionHandler)
        WriteToLog("***The OnException event points to another "
                   "handler. There may be a conflict.***" );
//      WriteToLog("***Событие OnException указывает на другой"
//                 "обработчик. Возможен конфликт.***" );

    //Восстановить предыдущий обработчик исключительных ситуаций
    Application->OnException = previousExceptionHandler ;

    // Освободить используемые ресурсы
    delete log ;
    delete log_External ;
}// Деструктор
//---------------------------------------------------------------
```

В листинге 4.11 описаны методы открытия журнала регистрации. Интересно, что в методе `OpenLogFile()` буфер журнала (объявленный в заголовочном файле как закрытый) открывается как для ввода, так и для вывода данных. Однако в этом классе только добавляется информация в журнал регистрации. Его единственным назначением является открытие файла для чтения, а также отображение информации из журнала регистрации. Для этого класс выполняет следующие действия:

- получает право управления журналом регистрации;
- предоставляет доступ только для чтения;
- сокращает до минимума объем используемых ресурсов за счет совместного использования двумя потоками одного буфера файла.

При организации совместного использования буфера следует позаботиться о том, чтобы он имел большую продолжительность жизни, чем зависящие от него потоки. Дело в том, что в данном случае, потоки освобождаются деструктором, тогда как буфер файла освобождается при очистке стека класса в конце работы деструктора.

Листинг 4.11. Открытие журнала регистрации

```
void __fastcall ErrorHandler::OpenLogFile( void )
/*************************************************************\
* ACCESS: Protected *
\*************************************************************/
{
    // Открытие файла регистрации
    // Замечание: Файл журнала регистрации будет закрыт
    //            автоматически после удаления объекта буфера.
    //            Буфер гарантировано переживет потоки,
    //            поскольку он создается в стеке, тогда как
    //            потоки - в куче.
    sharedFileBuffer.open( GetLogFileName().c_str(),
                           ios_base::in |
                           ios_base::out | ios_base::app ) ;

    log          = new ostream( &sharedFileBuffer ) ;
    log_External = new istream( &sharedFileBuffer ) ;
```

```
}// OpenLogFile()
//---------------------------------------------------------

String __fastcall ErrorHandler::GetLogFileName( void )
/***********************************************************\
* ACCESS: Public *
\***********************************************************/
{
    String exeName( Application->ExeName ) ;
    int extStart = exeName.Pos( "." ) ;
    String logFileName( exeName.SubString( 0, extStart -1 ) );
    logFileName += ".log" ;

    return logFileName ;
}//GetLogFileName()
//---------------------------------------------------------

istream*__fastcall ErrorHandler::GetLogFileStream(void )
/***********************************************************\
* ACCESS: Public *
\***********************************************************/
{
    return log_External ;
}//GetLogFileStream()
//---------------------------------------------------------
```

В листинге 4.12 показан код нового обработчика исключительных ситуаций, который используется в коде конструктора. Его задача заключается только в вызове функции `LogEntry()`, которая выполняет всю остальную работу. Код этой функции приведен в листинге 4.13. Как видно, методу известна дочерняя форма многодокументного интерфейса, являющаяся активной. Если программа не использует многодокументный интерфейс, то информация о форме не будет зарегистрирована до тех пор, пока исключительная ситуация не будет перехвачена непосредственно или используемый по умолчанию обработчик не будет заменен для каждой формы так, как описано выше.

Этот метод также временно обращается к используемому по умолчанию обработчику исключительных ситуаций, если применяемый обработчик вызывает появление исключительной ситуации.

Листинг 4.12. Новый обработчик исключительных ситуаций `DefaultExceptionHandler`

```
void __fastcall ErrorHandler::DefaultExceptionHandler(
    TObject *sender,
    Exception*exception )
/***********************************************************\
* ACCESS: Public *
\***********************************************************/
{
    // Восстановление обработчика исключительных ситуаций,
    // используемого по умолчанию компилятором
    Application->OnException = NULL ;
```

```
    // Обработка исключительной ситуации
    LogEntry(sender, exception, EXCEPTION_UNHANDLED,
        Application->MainForm->ActiveMDIChild );

    // Опять переход к использованию нашего обработчика.
    Application->OnException = DefaultExceptionHandler ;
}//DefaultExceptionHandler()
//---------------------------------------------------------
```

Именно метод `LogEntry()` фактически выполняет всю работу и на основании нескольких параметров способен вывести более обширную информацию. Так как он является открытым методом, его можно вызвать непосредственно, хотя обычно это делается в блоке `catch`.

Изучая код метода `LogEntry()`, который приведен в листинге 4.13, можно сразу заметить, что он прежде всего информирует пользователя о возникшей проблеме, отображая сообщение об исключительной ситуации. Всякая другая информация совершенно излишня для пользователя. Обратите внимание, что параметр `doDisplay` по умолчанию имеет значение `true`. После этого метод проверяет, не является ли ситуация непоправимой.

Затем в файл журнала регистрации таких ситуаций добавляется еще одна запись. Впоследствии для получения дополнительной информации о состоянии системы и регистрации ее в журнале вызываются другие закрытые и защищенные методы.

Конечно, этот класс должен быть особенно надежно защищен от возникновения исключительных ситуаций, а потому его следует заключить в конструкцию `try/finally`. Следует иметь в виду, что защита от исключительных ситуаций не означает их отсутствия. Поэтому `DefaultExceptionHandler()` возвращает управление обработчику исключительных ситуаций, который по умолчанию используется компилятором.

Листинг 4.13. Регистрация исключительной ситуации

```
void __fastcall ErrorHandler::LogEntry( TObject*const sender,
    Exception* exception, Severity severity, TForm*const form,
    const bool doDisplay )
/*************************************************************\
* ACCESS: Public *
\*************************************************************/
{
    if( doDisplay )
        Show( exception ) ;

    if( severity == EXCEPTION_FATAL )
        WriteToLog( "***FATAL EXCEPTION - Attempting to " +
                    "log system 's state***" ) ;
//      WriteToLog( "***НЕПОПРАВИМАЯ ОШИБКА - При попытке " +
//                  "регистрации состояния системы***" ) ;

    //***Запись в файл журнала регистрации***
    if( form )
        form->Cursor = crHourGlass ;

    try
```

```
    {
        WriteToLog( " " ) ;
        if( severity != MESSAGE_ONLY )
            WriteToLog( ERROR_TITLE ) ;
        else
            WriteToLog( MESSAGE_TITLE ) ;

        WriteToLog( "Handler", "VCL Exception" ) ;
        LogExceptionState( exception ) ;
        LogObjectState( sender ) ;
        LogFormState( form ) ;
        LogProgramState( severity ) ;
        LogWindowsState() ;
    }//try
    __finally
    {
        // Всегда следует вернуть
        // используемый по умолчанию указатель мыши
        if( form )
            form->Cursor = crDefault ;
    }//finally

    if( severity == EXCEPTION_FATAL )
        Application->Terminate() ;
}//LogEntry()
//---------------------------------------------------------------
```

Для добавления записи в файл журнала регистрации транзакций без вызова исключительной ситуации можно использовать перегруженный вариант функции `LogEntry()`. Это может понадобиться для регистрации ошибки, обнаруженной с помощью другого механизма (возврата кода, например), или некой ситуации, возникновение которой необходимо зафиксировать. Перегруженный вариант этой функции приведен в листинге 4.14.

Листинг 4.14. Регистрация сообщения

```
void __fastcall ErrorHandler::LogEntry( String message,
                                        Severity severity,
                                        TForm*form,
                                        const bool doDisplay )
/*************************************************************\
* ACCESS: Public *
\*************************************************************/
{
    if( doDisplay )
        Show(message );

    //*** Запись в файл журнала регистрации ***
    if( form )
        form->Cursor = crHourGlass ;
```

```
    try
    {
        WriteToLog( " " ) ;
        if( severity != MESSAGE_ONLY )
            WriteToLog( ERROR_TITLE ) ;
        else
            WriteToLog( MESSAGE_TITLE ) ;

        WriteToLog( message ) ;

        LogFormState( form ) ;
        LogProgramState( severity ) ;
        LogWindowsState() ;
    }//try
    __finally
    {
        // Всегда, когда это возможно, следует вернуть
        // используемый по умолчанию указатель мыши
        if( form )
            form->Cursor = crDefault ;
    }//finally
}//LogEntry()
//---------------------------------------------------------------
```

В листинге 4.15 показан способ извлечения информации, содержащейся в объекте исключительной ситуации. Это очень простой процесс, основанный на приведении типа к наиболее близкому производному типу и получении интересующей информации. Обратите внимание на то, что исключительная ситуация типа `EDBEngineError` может содержать вложенные сообщения.

Листинг 4.15. Извлечение подробной информации об исключительной ситуации

```
void __fastcall ErrorHandler::LogExceptionState(
                                  Exception* exception )
/*************************************************************\
* ACCESS: Private *
\*************************************************************/
{
    if( ! exception )
        return ;

    WriteToLog( "ERROR", exception->Message ) ;
    WriteToLog( "Exception type", exception->ClassName() ) ;

    //EDBEngineError
    EOleException*  OleException ;
    EOleSysError*   OleSysError ;
    EDBEngineError* BDEException = dynamic_cast<EDBEngineError*>
        (exception) ;
    if(BDEException )
```

```
{
    TDBError*  error_ptr ;
    AnsiString errorCount_String(BDEException->ErrorCount) ;
    AnsiString subscript_String, category_String,
        errorCode_String, errorSubCode_String,
        nativeErrorCode_String ;

    WriteToLog(DETAIL_TITLE );

    // Получение вложенных сообщений
    for( int subscript = 0;
         subscript < BDEException->ErrorCount;
         subscript++)
    {
        error_ptr = BDEException->Errors[subscript] ;

        if( error_ptr )
        {
            subscript_String       = subscript +1 ;
            category_String        = error_ptr->Category ;
            errorCode_String       = error_ptr->ErrorCode ;
            errorSubCode_String    = error_ptr->SubCode ;
            nativeErrorCode_String =
                              (int)error_ptr->NativeError ;
            WriteToLog( "DBE Error ",
                        subscript_String + " of " +
                        errorCount_String ) ;
            WriteToLog( "->Message", error_ptr->Message ) ;
            WriteToLog( "->Category", category_String ) ;
            WriteToLog( "->Error Code", errorCode_String ) ;
            WriteToLog( "->Sub Code", errorSubCode_String ) ;
            if( error_ptr->NativeError )
                WriteToLog( "->Native Code",
                            nativeErrorCode_String ) ;
        }//if
        else
            WriteToLog( "DBE Error", "*Unavailable *" ) ;
    }//for
}//if EDBEngineError

//EOleException
else if( ( OleException =
           dynamic_cast<EOleException*>(exception)
         )
         != 0 )
{
    WriteToLog( DETAIL_TITLE ) ;
    WriteToLog( "OLE Error Code", OleException->ErrorCode) ;
    WriteToLog( "OLE Source App.", OleException->Source ) ;
    WriteToLog("Refer To File", OleException->HelpFile ) ;
```

```
    }//if EOleException

    //EOleSysError
    else if( ( OleSysError =
               dynamic_cast<EOleSysError*>(exception) )
             != 0 )
    {
        WriteToLog( DETAIL_TITLE ) ;
        WriteToLog( "OLE API Error", OleSysError->ErrorCode ) ;
    }//if EOleSysError

}//LogExceptionState()
//---------------------------------------------------------------
```

В листинге 4.16 показано, как извлечь информацию о состоянии операционной системы Windows. Информация очень подробна, что позволяет не упустить в приложении ни одного факта, важного для оценки состояния системы. По сути это комбинация прямых API-вызовов Windows и вызовов функций библиотеки VCL.

Листинг 4.16. Извлечение подробных сведений об операционной системе

```
void __fastcall ErrorHandler::LogWindowsState( void )
/*************************************************************\
* ACCESS: Private *
\*************************************************************/
{
    // Заголовок
    WriteToLog( SYSTEM_TITLE ) ;

    // Последний код ошибки MS-Windows и
    // сообщение для данного потока
    AnsiString currentThreadID( (int)GetCurrentThreadId() ) ;
    WriteToLog( "Thread ID", currentThreadID ) ;
    int lastWin32Error_int = (int)GetLastError() ;
    AnsiString lastWin32Error_String(lastWin32Error_int ) ;
    if( lastWin32Error_int )
    {
        char*buffer ; // Буфер LPVOID ;
        if( FormatMessage( FORMAT_MESSAGE_ALLOCATE_BUFFER |
        FORMAT_MESSAGE_FROM_SYSTEM, NULL, lastWin32Error_int,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
        (LPTSTR) &buffer, 0, NULL ) )
       {
           lastWin32Error_String += " - " ;
           lastWin32Error_String += buffer ;
       }//if message
       LocalFree( buffer ) ;

       SetLastError( 0 ) ; // Сброс кода ошибки для потока Win32
     }//if Win32 error
```

```
WriteToLog( "Win32 Error", lastWin32Error_String ) ;

// Текущее время
WriteToLog( "Date &Time", Now() ) ;

// Имя компьютера
char   buffer1[MAX_COMPUTERNAME_LENGTH + 1] ;
DWORD  bufferSize = sizeof( buffer1 ) ;
BOOL   success    = GetComputerName(buffer1, &bufferSize) ;
String buffer     = success ? buffer1 : "N/A" ;
WriteToLog( "Computer Name", buffer ) ;

// Имя пользователя Windows
bufferSize = sizeof( buffer1 ) ;
success    = GetUserName( buffer1, &bufferSize ) ;
buffer     = success ? buffer1 : "N/A" ;
WriteToLog( "Windows User", buffer ) ;

// Текущий путь
String path( ExpandFileName( "." ) ) ;
WriteToLog( "Current Path", path ) ;

// Информация о системе
SYSTEM_INFO systemInfo ;
GetSystemInfo( &systemInfo ) ;
switch( systemInfo.dwProcessorType )
{
    case PROCESSOR_INTEL_386:
    WriteToLog( "Processor", "Intel 386" ) ;
    break ;

    case PROCESSOR_INTEL_486:
    WriteToLog( "Processor", "Intel 486" ) ;
    break ;

    case PROCESSOR_INTEL_PENTIUM:
    WriteToLog( "Processor", "Intel Pentium" ) ;
    break ;

    case PROCESSOR_MIPS_R4000:
    WriteToLog( "Processor", "MIPS" ) ;
    break ;

    case PROCESSOR_ALPHA_21064:
    WriteToLog( "Processor", "Alpha" ) ;
    break ;

    default:
    WriteToLog( "Processor", "Unknown" ) ;
    break ;
```

```
    }//switch
    char addressBuffer [35 ] ; // ultoa() возвращает до 33 байт
    AnsiString addressString(ultoa( (unsigned long)systemInfo.
        lpMinimumApplicationAddress, addressBuffer, 16 ) ) ;
    WriteToLog( "Min Address", addressString ) ;
    addressString = ultoa( (unsigned long)systemInfo.
        lpMaximumApplicationAddress, addressBuffer, 16 ) ;
    WriteToLog( "Max Address", addressString ) ;

    // Состояние памяти
    MEMORYSTATUS memoryStatus ;
    const unsigned long OneK = 1024 ;
    const unsigned long OneMeg = OneK * OneK ;
    char valueBuffer [20 ] ; // itoa() возвращает до 17 байт
    memoryStatus.dwLength = sizeof( memoryStatus ) ;
    GlobalMemoryStatus( &memoryStatus ) ;

    AnsiString valueString(itoa( (int)memoryStatus.dwMemoryLoad,
        valueBuffer, 10 ) ) ;
    WriteToLog( "Mem Load", valueString + "%" ) ;
    valueString = itoa( (int)(memoryStatus.dwTotalPhys / OneK),
                        valueBuffer, 10 ) ;
    WriteToLog( "Total Phys Mem", valueString + " KB" ) ;
    valueString = itoa( (int)(memoryStatus.dwAvailPhys / OneK),
                       valueBuffer, 10 ) ;
    WriteToLog( "Avail Phys Mem", valueString + " KB" ) ;
    valueString = itoa( (int)(memoryStatus.dwTotalVirtual /
                            OneMeg),
                       valueBuffer, 10 ) ;
    WriteToLog( "Total Virt Mem", valueString + " MB" ) ;
    valueString = itoa( (int)(memoryStatus.dwAvailVirtual /
                            OneMeg),
                       valueBuffer, 10 ) ;
    WriteToLog( "Avail Virt Mem", valueString + " MB" ) ;
    valueString = itoa( (int)(memoryStatus.dwTotalPageFile /
                            OneMeg),
                       valueBuffer, 10 ) ;
    WriteToLog( "Total Page Mem", valueString + " MB" ) ;
    valueString = itoa( (int)(memoryStatus.dwAvailPageFile /
                            OneMeg),
                       valueBuffer, 10 ) ;
    WriteToLog( "Avail Page Mem", valueString + " MB" ) ;
}//LogWindowsState()
//---------------------------------------------------------------
```

Код записи информации о состоянии программы в журнал регистрации показан в листинге 4.17. Так как это метод общего назначения, то в нем содержится только самая основная информация. Обычно в журнал регистрации включаются значения всех глобальных переменных. Более подробная информация по этому поводу излагается выше в разделе о добавлении в класс специфической для проекта информации.

Листинг 4.17. Запись полученной информации о состоянии программы в журнал регистрации

```
void __fastcall ErrorHandler::LogProgramState(Severity severity)
/*************************************************************\
* ACCESS: Protected *
\*************************************************************/
{
    WriteToLog( PROGRAM_TITLE ) ;

    //Severity
    switch( severity )
    {
        case MESSAGE_ONLY:
            WriteToLog( "Severity", "System message only" ) ;
            break ;

        case EXCEPTION_HANDLED:
            WriteToLog( "Severity", "Exception handled" ) ;
            break ;

        case EXCEPTION_UNHANDLED:
            WriteToLog( "Severity", "Unhandled exception" ) ;
            break ;

        case EXCEPTION_FATAL:
            WriteToLog( "Severity ", "*FATAL *" ) ;
            break ;

        default:
            WriteToLog( "Severity", "*Unknown *" ) ;
            break ;
    }//switch

    // Получение информации о версии проекта
    // из выполняемого файла
    bool rc ;
    DWORD handle ;
    DWORD size = GetFileVersionInfoSize(
        Application->ExeName.c_str(),
        &handle ) ;
    if( size )
    {
       void*buffer = new char[size] ;
       try
       {
           rc = GetFileVersionInfo(Application->ExeName.c_str(),
                                   handle, size, buffer ) ;
           if( rc )
           {
              char*valuePointer ;
              unsigned int valueSize ;
```

```
            rc = VerQueryValue(buffer,
                               TEXT("\\StringFileInfo\\" +
                                    "040904E4\\FileVersion"),
                               (void**)&valuePointer,
                               &valueSize ) ;
            if( rc )
            {
               AnsiString value( valuePointer,valueSize ) ;
               WriteToLog( "Version", value ) ;
            }//if VerQueryValue()
         }//if GetFileVersionInfo()
      }//try
      __finally
      {
          delete buffer ;
      }//finally
   }//if GetFileVersionInfoSize()

   // Используемый компилятор
   switch( __BORLANDC__ )
   {
       case 0x520:
           WriteToLog( "Compiler", "C++Builder 1" ) ;
           break ;

       case 0x530:
           WriteToLog( "Compiler", "C++Builder 3" ) ;
           break ;

       case 0x540:
           WriteToLog( "Compiler", "C++Builder 4" ) ;
           break ;

       case 0x550:
           WriteToLog( "Compiler", "C++Builder 5" ) ;
           break ;

       default:
           char compiler[50 ] ;
           sprintf( compiler, "Borland C++%x", __BORLANDC__ ) ;
           WriteToLog( "Compiler", compiler ) ;
   }//switch
}//LogProgramState()
//---------------------------------------------------------------
```

Кроме того, часто очень важно иметь информацию об объекте, который вызвал исключительную ситуацию, поскольку это позволяет выявить недопустимые действия пользователя. В листинге 4.18 показано, как это делается с помощью метода `LogObjectState()`. В первой части кода извлекается информация о наиболее важных свойствах базовых классов библиотеки VCL. Во второй части представлена подробная информация об объекте, если такие данные

имеются (включая значения всех полей присоединенного набора данных). Это выполняется за счет поиска свойства DataSource с помощью того же метода, который используется инспектором объектов Object Inspector в IDE-среде. Этот метод позволяет избежать приведения типов, т.е. для его применения необязательно знать тип объекта.

Причиной возникновения исключительной ситуации может быть не текущий объект, а связанный с ним. Например, если исключительная ситуация была вызвана во время манипуляций с другим объектом в коде обработчика события OnClick кнопки, то метод DefaultExceptionHandler() укажет на эту кнопку как на инициатора исключительной ситуации. При чтении записей в журнале регистрации следует иметь в виду этот факт.

Для настройки этого метода следует вспомнить раздел о добавлении в класс специфической информации.

Обратите внимание на то, что функция typeid() возвращает только RTTI-информацию об объектах C++Builder. Причем для объектов Delphi возвращается только информация о статическом типе, который обычно равен TObject.

Листинг 4.18. Извлечение подробной информации об объекте, вызвавшем исключительную ситуацию

```
void __fastcall ErrorHandler::LogObjectState( TObject* sender )
/************************************************************\
* ACCESS: Protected *
\************************************************************/
{
    if( ! sender )
        return ;

    TDataSet*dataSetReferenced = NULL ;

    WriteToLog( OBJECT_TITLE );

    WriteToLog( "Calling Obj", sender->ClassName() ) ;
    WriteToLog( "->BCB Class", typeid(sender).name() ) ;

    char addressBuffer[35] ; // ultoa() возвращает до 33 байт
    AnsiString addressString ;
    addressString = ultoa( (unsigned long)sender,
                           addressBuffer, 16 ) ;
    WriteToLog( "->Address", addressString ) ;

    // Является ли классом TComponent?
    TComponent* component = dynamic_cast<TComponent*>(sender) ;
    if( component )
    {
        WriteToLog( "Component",
                    "(" + component->ClassName() + ")" ) ;
        WriteToLog( "->Class", typeid(component).name() ) ;
        TControl*owner =
            dynamic_cast<TControl*>( component->Owner ) ;
        if( owner )
            WriteToLog( "->Owner", owner->Name ) ;
```

```
    // Является ли классом TControl?
    control = dynamic_cast<TControl*>( sender ) ;
    if( control )
    {
        WriteToLog( "->Name", control->Name ) ;

        // Управляется ли этот элемент данными?
        // Если да, получить сведения об этом наборе данных.
        PPropInfo propInfo = GetPropInfo(
                         PTypeInfo( control->ClassInfo() ),
                         "DataSource" ) ;
        if( propInfo )
        {
            TDataSource* dataSourceProperty =
               (TDataSource*)GetOrdProp(control, propInfo) ;

            if( dataSourceProperty )
                dataSetReferenced =
                    dataSourceProperty->DataSet ;
        }// Для элемента, управляемого данными.

        // Является ли он классом TWinControl?
        TWinControl* winControl =
            dynamic_cast<TWinControl*>( sender ) ;
        if( winControl )
        {
            String showing =
                winControl->Showing ? "True" : "False" ;
            WriteToLog("->Showing", showing ) ;

            // Является ли он классом TPageControl?
            TPageControl* pageControl =
                dynamic_cast<TPageControl*>( sender ) ;
            if( pageControl )
            {
               WriteToLog("->Tab",
                         pageControl->ActivePage->Caption);
            }//if для TPageControl
        }//if для TWinControl
    }//if для TControl
}//if для TComponent

// Получение сведений об [указанном] наборе данных.
TDataSet*dataSet = dataSetReferenced ? dataSetReferenced :
    dynamic_cast<TDataSet*>( sender ) ;
if( dataSet )
{
    WriteToLog( DATA_TITLE ) ;
    WriteToLog( "DataSet", dataSet->Name ) ;
```

```
        WriteToLog( "->Owner", dataSet->Owner->Name ) ;
        WriteToLog( "->Active",
                    dataSet->Active ? "True" : "False" ) ;

        String state;
        switch( dataSet->State )
        {
            case dsInactive:
                state = "Inactive" ;
                break ;

            case dsBrowse:
                state = "Browsing" ;
                break ;

            case dsEdit:
                state = "Editing" ;
                break ;

            case dsInsert:
                state = "Inserting" ;
                break ;

            case dsCalcFields:
                state = "Updating calc.fields" ;
                break ;

            case dsFilter:
                state = "Filtering" ;
                break ;

            default:
                state = "Internal processing" ;
                break ;
        }//switch
        WriteToLog( "->State", state ) ;

       // Сведения о содержимом полей
       TFields* fields = dataSet->Fields ;
       int       count = fields->Count ;
       WriteToLog( String( "->FIELDS:(" + String(count) +
                           ")" ) ;
       for( int ctr=0; fields &&(ctr < count); ++ctr )
       {
           WriteToLog( fields->Fields[ctr]->FieldName,
                       fields->Fields[ctr]->AsString ) ;
       }//for для полей
    }//if для dataSet
}//LogObjectState()
//---------------------------------------------------------------
```

В больших проектах информация, собираемая методом LogObjectState(), может быть слишком специальной, что затруднит определение места, где следует искать ошибку. Показанный в листинге 4.19 метод LogFormState() позволяет извлечь очень общую, но чрезвычайно полезную информацию о контексте исключительной ситуации.

Листинг 4.19. Извлечение информации о контексте исключительной ситуации

```
void __fastcall ErrorHandler::LogFormState( TForm* form )
/*************************************************************\
* ACCESS: Protected *
\*************************************************************/
{
    if( ! form )
    return ;

    WriteToLog( FORM_TITLE ) ;

    WriteToLog( "Form", form->Name ) ;

    if( form->ActiveControl )

    WriteToLog( "->Active Cntrl", form->ActiveControl->Name ) ;

    if( form->Owner && (form->Owner != Application) )
        WriteToLog( "->Owner", form->Owner->Name ) ;
    else
        WriteToLog( "->Owner", "Application" ) ;

    if( form->Parent )
        WriteToLog("->Parent", form->Parent->Name ) ;
}//LogFormState()
//---------------------------------------------------------
```

Этот класс позволяет получить огромное количество информации и записать ее в журнал регистрации. А какую информацию можно отобразить для пользователя? Дело в том, что пользователь обязательно должен быть проинформирован о возникшей проблеме, но в достаточно кратком и исчерпывающем виде. Для отображения такой информации на экране используется модальное окно.

В листинге 4.20 приведены две перегруженные версии метода Show(), по одной для каждой версии метода LogEntry(). Одна из версий отображает сообщения об исключительных ситуациях, а другая используется для отображения обычных строковых сообщений.

Эти методы легко переопределить для повышения интеграции с приложением. Например, для сообщения можно использовать другое окно или строку состояния.

Листинг 4.20. Организация обратной связи с пользователем

```
void __fastcall ErrorHandler::Show( Exception*exception )
/*************************************************************\
* ACCESS: Protected *
\*************************************************************/
{
```

```
    MessageBeep( MB_ICONEXCLAMATION );
    Application->ShowException( exception );
}//Show()
//---------------------------------------------------------------

void __fastcall ErrorHandler::Show( String message )
/*************************************************************\
* ACCESS: Protected *
\*************************************************************/
{
    MessageBeep( MB_ICONEXCLAMATION );
    MessageBox( NULL,message.c_str(),
                "Attention", MB_OK +MB_ICONERROR ) ;
}//Show()
//---------------------------------------------------------------
```

Для записи полученных сведений в журнал регистрации во всех методах `Get...State()` следует использовать одну из перегруженных версий метода `WriteToLog()`. В первой — полученная информация сразу передается в журнал регистрации, а во второй — еще до записи в журнал выполняется проверка и изменение длины записываемой в журнал строки.

Как показано в листинге 4.21, эти версии метода имеют очень простую структуру. Поскольку используемый поток позволяет только добавлять информацию в файл, то даже не придется искать место вставки.

Листинг 4.21. Добавление данных в журнал регистрации

```
void __fastcall ErrorHandler::WriteToLog( String text )
/*************************************************************\
* ACCESS: Protected *
\*************************************************************/
{
    *log << text.c_str() << endl ;
}//WriteToLog()
//---------------------------------------------------------------

void __fastcall ErrorHandler::WriteToLog( String name,
                                          String value )
/*************************************************************\
* ACCESS: Protected *
\*************************************************************/
{
    String text;
    const int MAX_NAME =15 ;

    name.Trim() ;
    if( name.Length() > MAX_NAME )
        text = AnsiString::StringOfChar(
                          ' ', MAX_NAME - name.Length() ) ;
    else
        text = SetLength( MAX_NAME ) ; // Только для локальной // копии.
```

```
    text += name ;
    text += ":" ;
    text += value.Trim() ;

    WriteToLog( text ) ;
}//WriteToLog()
//---------------------------------------------------------------
```

Для просмотра класса `ErrorHandler` в действии запустите новое приложение с многодокументным интерфейсом (выбирая команду меню File⇨New и пиктограмму MDI Application во вкладке Projects). Выберите каталог для нового проекта. Затем выберите команду меню Project⇨Add to Project и найдите файл `EHandler.cpp` на прилагаемом к этой книге компакт-диске. Теперь приложение готово к регистрации сообщений о возникающих ошибках.

Заключительные замечания об обработке исключительных ситуаций

Хотя обработчик исключительных ситуаций, используемый по умолчанию компилятором, прекрасно подходит для простых и небольших программ, его явно недостаточно для большинства проектов. Способность собирать информацию о состоянии системы в заданный момент времени и сохранять ее для последующего просмотра и изучения является неотъемлемым условием успешного создания программного продукта.

Представленный здесь обработчик подходит не для всех типов приложений. Тем не менее используемые в нем методы могут быть легко перенесены для обслуживания других ситуаций.

Работа над созданием обработчика исключительных ситуаций никогда не заканчивается. Поэтому, вполне возможно, что представленный здесь исходный код будет развиваться для обработки новых исключительных ситуаций и включения в него фрагментов, созданных вашими коллегами.

Создание многопотоковых приложений

Для программистов термины “многозадачность” (“multitasking”) и “многопотоковость” (“multithreading”) часто являются источником путаницы и излишней головной боли. Достаточно один раз глубоко разобраться в этих понятиях и они станут частью обязательных знаний программиста.

Многозадачность

Многозадачность — это способность операционной системы одновременно выполнять несколько программ. (Читателям наверняка уже приходилось применять многозадачность при переключении из документа Microsoft Word в документ Windows Explorer.) Многозадачность может показаться характерной особенностью только таких графических операционных систем как Windows или Linux, однако даже в самых первых компьютерах в некоторой степени использовалась концепция многозадачности. Например, в операционной системе UNIX допускается фоновое выполнение сразу нескольких программ.

В оболочках Windows 3.x приложения используют так называемую *кооперативную многозадачность* (*cooperative multitasking*). Кооперативность означает, что программа может управлять центральным процессором, а потому перед переключением для продолжения работы с другим приложением она должна закончить обработку данных. Этот тип многозадачности имеет серьезный недостаток: если приложение перестает отвечать на запросы пользователя, это приводит к зависанию операционной системы. В 32-разрядных версиях Windows эта проблема была разрешена на основе *вытесняющей многозадачности* (*preemptive multitasking*). Согласно словарному определению, "вытесняемый" ("preemptive") означает [процесс], "выполняемый до тех пор, пока не будет активизирован какой-либо другой" [процесс]. Иначе говоря, для переключения между задачами в 32-разрядных версиях Windows выполнение текущего приложения приостанавливалось, независимо от того, готово ли оно передать управление или нет.

На заметку

По очевидным причинам кооперативная многозадачность также называется "невытесняющей многозадачностью" ("nonpreemptive multitasking"). В отличие от вытесняющей многозадачности, операционная система с кооперативной многозадачностью не может приостановить выполнение приложения, которое перестало отвечать на запросы.

Многопотоковость

Многопотоковость — это способность программы одновременно выполнять несколько задач (потоков). В большинстве приложений для Windows используется только один *первичный поток* (*primary thread*). Первичный поток отвечает за создание дочерних окон и обработку сообщений. Все вторичные потоки используются для выполнения фоновых операций: загрузки больших файлов, просмотра информации, выполнения математических вычислений. Далее мы рассмотрим различные аспекты создания многопотоковых приложений.

Маленькие дети иногда бесконечно повторяют подслушанные слова — просто для того, чтобы продемонстрировать свои знания или "взрослость". Аналогичную ситуацию можно наблюдать и среди программистов. Некоторые разработчики стремятся чрезмерно часто использовать только что освоенные ими методы программирования. Не используйте потоки в приложении при наличии продолжительных фоновых операций. Иногда достаточно внести небольшие улучшения в код, чтобы избежать использования потоков. Зачем же усложнять себе жизнь? Как будет показано ниже в этой главе, многопотоковые приложения обладают многими преимуществами.

Создание потока с помощью вызовов API-функций

Новый поток можно создать, вызывая API-функцию `CreateThread()`. Помимо прочего, параметры функции `CreateThread()` содержат атрибуты безопасности, флаги создания и т.д.

```
HANDLE CreateThread(
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    DWORD dwStackSize,
    LPTHREAD_START_ROUTINE lpStartAddress,
    LPVOID lpParameter,
    DWORD dwCreationFlags,
    LPDWORD lpThreadId
);
```

Структура атрибутов безопасности LPSECURITY_ATTRIBUTES определяет, могут ли другие процессы изменить объект, а также может ли дочерний процесс наследовать его дескриптор. Если значение lpThreadAttributes равно NULL, то поток получает принимаемый по умолчанию дескриптор безопасности.

Параметр dwCreationFlags указывает флаги создания потока. Если его значение равно CREATE_SUSPENDED, то поток не запустится до вызова функции ResumeThread(). Задайте параметру значение 0, чтобы поток запускался сразу после его создания.

Параметр lpThreadId указывает на пустое значение типа DWORD, которое получит идентификатор потока. При работе с Windows NT/2000 идентификатор потока не возвращается, если значение этого параметра NULL. В Windows 9x этот параметр обязательно должен иметь не пустое значение. Для обеспечения полной совместимости с текущей операционной системой не используйте для этого параметра значение NULL.

Наиболее важное значение имеет запускающая функция, которая также называется *потоковой функцией*, или *функцией потоков* (*thread function*). lpStartAddress — адрес этой функции, которая принимает один параметр и возвращает значение типа DWORD.

```
DWORD WINAPI ThreadFunc(LPVOID);
```

Наконец, параметры dwStackSize и lpParameter указывают размер стека (в байтах) и передаваемый потоку параметр.

В некотором смысле запускающую программу можно сравнить с функцией main() или WinMain() в программе на языке C++. Функция ThreadFunc() является главной точкой входа потока.

API-функция CreateThread(), как и многие другие API-функции, содержит большой список простых и сложных аргументов. Однако при первом знакомстве с ними изучение сразу всех аспектов использования этой функции может вас запутать.

Для преодоления этой проблемы следует прежде всего обратить внимание на аргументы, которые могут иметь нулевые значения. Например, значение 0 можно задать практически для всех параметров CreateThread(), за исключением lpStartAddress и lpThreadId. Изучив эти аргументы, всегда можно вернуться назад и продолжить изучение других параметров функции CreateThread().

На основании приведенных выше сведений и справочного файла Win32 Programmer's Reference, теперь можно приступить к созданию простого многопотокового приложения. В рассматриваемом ниже примере используются две кнопки: Start и Stop. После щелчка на кнопке Start пользователь запустит вновь созданный поток, который в произвольном порядке начнет рисовать эллипсы и прямоугольники. После щелчка на кнопке Stop выполнение потока будет приостановлено. Взгляните на листинг 4.22, полный текст которого можно найти в каталоге ThreadAPI на прилагаемом к этой книге компакт-диске.

Листинг 4.22. Фрагмент кода из файла ThreadFormUnit.cpp

```
#include <vcl.h>
#pragma hdrstop

#include "ThreadFormUnit.h"
#pragma package(smart_init)
```

```
#pragma resource "*.dfm"

TThreadForm *ThreadForm;
HANDLE Thread;

DWORD WINAPI ThreadFunc(LPVOID Param)
{
   HANDLE MainWnd(Param);

   RECT R;
   GetClientRect(MainWnd,&R);

   const MaxWidth = R.right - R.left;
   const MaxHeight = R.bottom - R.top;
   int X1, Y1, X2, Y2, R1, G1, B1;
   bool IsEllipse;

   while(true)
   {
      HDC DC = GetDC(MainWnd);

      X1 = rand() % MaxWidth;
      Y1 = rand() % MaxHeight;
      X2 = rand() % MaxWidth;
      Y2 = rand() % MaxHeight;

      R1 = rand() & 255;
      G1 = rand() & 255;
      B1 = rand() & 255;

      IsEllipse = rand() & 1;

      HBRUSH Brush = CreateSolidBrush( RGB(R1, G1, B1) );
      SelectObject(DC, Brush);

      if(IsEllipse)
         Ellipse(DC, X1, Y1, X2, Y2);
      else
         Rectangle(DC, X1, Y1, X2, Y2);

      ReleaseDC(MainWnd, DC);
      DeleteObject(Brush);
   }
}

__fastcall
   TThreadForm::TThreadForm(TComponent*Owner): TForm(Owner)
{
   randomize();
   DWORD Id;
   Thread = CreateThread(0, 0, ThreadFunc,
```

```
      ThreadForm->Handle, CREATE_SUSPENDED, &Id);

   if(!Thread)
   {
      ShowMessage("Error! Cannot create thread.");
      Application->Terminate();
   }
}

void __fastcall TThreadForm::StartClick(TObject *)
{
   ResumeThread(Thread);
   Start->Enabled = false;
   Stop->Enabled = true;
}

void __fastcall TThreadForm::StopClick(TObject *)
{
   SuspendThread(Thread);
   Stop->Enabled = false;
   Start->Enabled = true;
}
```

На заметку Как видно в листинге 4.22, в этом коде почти исключительно используются API-функции, поскольку необходимо избежать доступа к свойствам и методам библиотеки VCL из вторичных потоков. В следующем разделе будут приведены причины этого и описаны способы решения этой проблемы.

Хотя этот код имеет достаточно большую длину, он прост и легок для понимания. В конструкторе формы с помощью функции `CreateThread()` создается приостановленный поток, а затем проверяется его корректность. Для изменения состояния потока с помощью кнопок Start и Stop используются API-функции `ResumeThread()` и `SuspendThread()`, соответственно. Наконец, потоковая функция рисует геометрические фигуры в произвольном порядке и с произвольными размерами на канве формы. (Обратите внимание, как дескриптор окна передается функции `ThreadFunc()`.) На рис. 4.2 показан внешний вид этого приложения.

На заметку Еще один эффективный способ запуска нового потока основан на использовании функции `_beginthread()`, которая определена в файле `process.h` (он содержится в подкаталоге `Include` каталога C++Builder):

```
unsigned long _beginthread(void (_USERENTRY *__start)(void *),
   unsigned __stksize,void *__arg);
```

Эта функция наиболее часто применяется в многопотоковых приложениях, т.к. содержит меньшее количество параметров.

Объект TThread

В C++Builder объекты потоков Windows инкапсулируются в объекте `TThread`. Создание нового потока в основном заключается в создании нового экземпляра наследника класса `TThread`. В листинге 4.23 приведено определение абстрактного класса `TThread`.

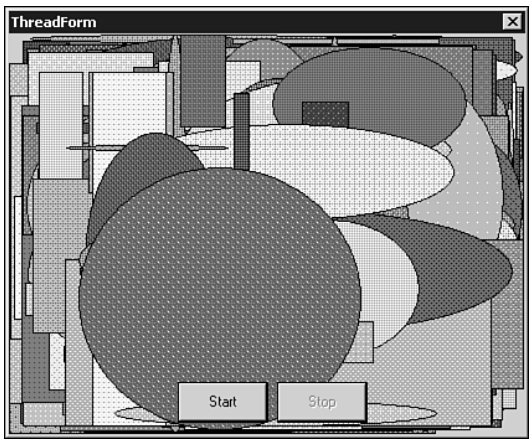


Рис. 4.2. Приложение ThreadAPI

Листинг 4.23. Класс TThread

```
class DELPHICLASS TThread;
class PASCALIMPLEMENTATION TThread : public System::TObject
{
    typedef System::TObject inherited;

private:
    unsigned FHandle;
    unsigned FThreadID;
    bool FTerminated;
    bool FSuspended;
    bool FFreeOnTerminate;
    bool FFinished;
    int FReturnValue;
    TNotifyEvent FOnTerminate;
    TThreadMethod FMethod;
    System::TObject* FSynchronizeException;
    void __fastcall CallOnTerminate(void);
    TThreadPriority __fastcall GetPriority(void);
    void __fastcall SetPriority(TThreadPriority Value);
    void __fastcall SetSuspended(bool Value);

protected:
    virtual void __fastcall DoTerminate(void);
    virtual void __fastcall Execute(void)=0 ;
    void __fastcall Synchronize(TThreadMethod Method);
    __property int ReturnValue = {read=FReturnValue,
                                  write=FReturnValue,
                                  nodefault};
    __property bool Terminated = {read=FTerminated,nodefault};

public:
```

```
    __fastcall TThread(bool CreateSuspended);
    __fastcall virtual ~TThread(void);
    void __fastcall Resume(void);
    void __fastcall Suspend(void);
    void __fastcall Terminate(void);
    unsigned __fastcall WaitFor(void);
    __property bool FreeOnTerminate = {read=FFreeOnTerminate, write=
         FFreeOnTerminate, nodefault};
    __property unsigned Handle = {read=FHandle, nodefault};
    __property TThreadPriority Priority =
        { read=GetPriority,
          write = SetPriority,
          nodefault};
    __property bool Suspended = {read=FSuspended,
                                 write=SetSuspended,
                                 nodefault};
    __property unsigned ThreadID = {read=FThreadID, nodefault};
    __property TNotifyEvent OnTerminate = {read=FOnTerminate,
                                           write=FOnTerminate};
};
```

Создать наследника класса `TThread` очень просто. Откройте с помощью команды меню File⇨New диалоговое окно репозитария объектов и выберите в нем пиктограмму Thread Object. При этом C++Builder предложит ввести имя класса для нового наследника. Введите **`TRandomThread`** и щелкните на кнопке OK.

После этого C++Builder автоматически создаст новый файл с исходным кодом, содержащий объект `TRandomThread`, который показан ниже.

```
#include <vcl.h>
#pragma hdrstop

#include "Unit2.h"
#pragma package(smart_init)

__fastcall TRandomThread::TRandomThread(bool CreateSuspended)
   : TThread(CreateSuspended)
{
}

void __fastcall TRandomThread::Execute()
{
}
```

Метод `Execute()` содержит код, который будет выполнен после запуска потока. То есть, метод `Execute()` заменяет потоковую функцию. Обратите также внимание на то, что конструктор объекта содержит параметр `CreateSuspended`. Аналогично флагу `CREATE_SUSPENDED`, если параметр `CreateSuspended` имеет значение `true`, то сначала следует вызвать метод `Resume()`; в противном случае метод `Execute()` не будет вызван.

В табл. 4.1 и 4.2 кратко описаны наиболее распространенные свойства и методы класса `TThread`.

Таблица 4.1. Свойства класса TThread

Свойство	Описание
FreeOnTerminate	Определяет, будет ли объект потока автоматически удален при завершении работы потока
Handle	Предоставляет доступ к дескриптору потока. Его следует использовать для вызова API-функций
Priority	Задает приоритет потока, который в случае необходимости можно регулировать
ReturnValue	Определяет значение, возвращаемое другим потокам после завершения работы текущего потока
Suspended	Указывает, приостановлено ли выполнение потока
Terminated	Определяет, может ли прекращаться выполнение потока
ThreadID	Определяет идентификатор потока

Таблица 4.2. Методы класса TThread

Метод	Описание
DoTerminate()	Вызывает обработчик события OnTerminate без прекращения работы потока
Execute()	Содержит код, который будет выполнен при запуске потока
Resume()	Возобновляет работу приостановленного потока
Suspend()	Приостанавливает работу потока
Synchronize()	Выполняет вызов в первичном потоке библиотеки VCL
Terminate()	Сигнализирует о прекращении выполнения потока
WaitFor()	Ожидает прекращения работы потока

Вернемся к нашему примеру с рисованием произвольных геометрических фигур (листинг 4.22). Но теперь попробуем использовать в нем только объекты библиотеки VCL.

Объект TRandomThread уже создан, поэтому мы используем его как вторичный поток приложения. Первый шаг состоит в добавлении включаемого файла основного модуля в модуль нового потока. Для этого выберите команду меню File⇨Include Unit Hdr, а затем — ThreadFormUnit.

При этом в конструктор объекта TRandomThread достаточно поместить только генератор случайных чисел.

```
__fastcall TRandomThread::TRandomThread(bool
   CreateSuspended) : TThread(CreateSuspended)
{
   randomize();
}
```

Рассмотрим теперь основную часть потока, т.е. метод Execute(). Теперь нам не нужно определять размер формы с помощью API-функции GetClientRect(). Для этого достаточно считать значения свойств ClientWidth и ClientHeight.

```
const MaxWidth = ThreadForm->ClientWidth;
const MaxHeight = ThreadForm->ClientHeight;
```

```
int X1, Y1, X2, Y2, R1, G1, B1;
bool IsEllipse;
```

Объект TCanvas, с которым мы уже знакомы, может значительно упростить процесс рисования. Однако в библиотеке VCL невозможен одновременный доступ к одному графическому объекту со стороны нескольких потоков. Следовательно, для решения этой проблемы необходимо применять методы Lock() и Unlock(), чтобы запретить доступ других потоков к объекту TCanvas при рисовании в нем фигур.

```
while(true)
{
   ThreadForm->Canvas->Lock();

   X1 = rand() % MaxWidth;
   Y1 = rand() % MaxHeight;
   X2 = rand() % MaxWidth;
   Y2 = rand() % MaxHeight;

   R1 = rand() & 255;
   G1 = rand() & 255;
   B1 = rand() & 255;

   IsEllipse = rand() & 1;

   ThreadForm->Canvas->Brush->Color = TColor(RGB(R1, G1, B1));

   if(IsEllipse)
      ThreadForm->Canvas->Ellipse(X1, Y1, X2, Y2);
   else
      ThreadForm->Canvas->Rectangle(X1, Y1, X2, Y2);

   ThreadForm->Canvas->Unlock();
}
```

На этом создание кода объекта потока заканчивается. Рассмотрим теперь основной модуль, в котором с помощью конструктора формы создадим новый экземпляр объекта TRandomThread, так как показано ниже.

```
TRandomThread* Thread;

__fastcall TThreadForm::TThreadForm(TComponent*) : TForm(Owner)
{
   Thread = new TRandomThread(true);
   if(!Thread)
   {
      ShowMessage("Error!Cannot create thread.");
      Application->Terminate();
   }
}
```

После щелчка на кнопке Start будет вызван представленный ниже метод Resume().

```
void __fastcall TThreadForm::StartClick(TObject *)
{
```

```
   Thread->Resume();
   Start->Enabled = false;
   Stop->Enabled = true;
}
```

После щелчка на кнопке Stop будет вызван представленный ниже метод `Suspend()`.

```
void __fastcall TThreadForm::StopClick(TObject *)
{
   Thread->Suspend();
   Stop->Enabled = false;
   Start->Enabled = true;
}
```

В IDE-среде C++Builder предусмотрено окно отладки потоков, которое содержит список всех имеющихся потоков с указанием их идентификаторов, состояния, расположения и статуса. Для отображения этого окна на экране следует выбрать команду меню View⇨Debug Windows⇨Threads или нажать комбинацию клавиш <Ctrl+Alt+T>.

Выполнение потока автоматически прекращается после завершения выполнения функции `Execute()` или закрытия приложения. Чтобы занятая потоком память по окончании его работы освобождалась, используйте следующую строку в методе `Execute()`:

```
FreeOnTerminate = true;
```

Однако иногда требуется прекратить выполнение потока с помощью кода. Для этого можно использовать метод `Terminate()`, который задает значение `true` для свойства `Terminated`.

При этом очень важно понимать, что метод `Terminate()` не покидает тело потока. Необходимо периодически проверять в методе `Execute()` наличие значения `true` для свойства `Terminated`. Например, чтобы иметь возможность прекратить работу потока `TRandomThread`, следует в блоке `while` добавить приведенную ниже строку кода.

```
while(true)
{
   if(Terminated) break;
```

Преимущество метода `Terminate()` заключается в том, что очистку можно выполнить самостоятельно — это позволяет более надежно управлять прекращением работы потока. К сожалению, вызов метода `Terminate()` становится бесполезным, если поток не отвечает на запросы.

API-функция `TerminateThread()` является более радикальным способом выхода из потока. Функция `TerminateThread()` немедленно закрывает текущий поток без освобождения памяти, занятой объектом потока. Эту функцию следует использовать только в экстремальных случаях, когда нет других вариантов. Синтаксис функции `TerminateThread()` достаточно прост и выглядит так:

```
TerminateThread((HANDLE)Thread->Handle, false);
```

Основной поток библиотеки VCL

Свойства и методы объектов библиотеки VCL не всегда защищены от вмешательства других потоков. Это значит, что при доступе к свойствам и методам может использоваться память, которая не защищена от вторжения других потоков. Следовательно, основной поток библиотеки VCL должен быть единственным потоком, управляющим библиотекой VCL.

На заметку Основной поток библиотеки VCL — это первичный поток приложения. Он управляет и обрабатывает сообщения Windows, которые получены элементами управления библиотеки VCL.

На заметку Графические объекты являются исключением из этого правила. Как уже говорилось выше, доступ к канве других потоков для рисования на ней геометрических фигур можно предотвратить с помощью методов `Lock()` и `Unlock()`.

Для того чтобы позволить потокам доступ к объектам библиотеки VCL, в объекте `TThread` предусмотрен метод `Synchronize()`. Он выполняет действия, которые содержатся в подпрограмме, так, как если бы они выполнялись из основного потока библиотеки VCL.

```
void __fastcall Synchronize(TThreadMethod &Method);
```

Рассмотрим пример потока, который отображает увеличивающиеся значения в компоненте `Label`. Очевидно, для этого в методе `Execute()` будет использован цикл `for`. Но как изменить текст надписи `Label`? Это можно сделать с помощью механизма синхронизации с библиотекой VCL. В листинге 4.24 содержится исходный код объекта `TLabelThread`, а на рис. **4.3** показаны результаты работы этого приложения.

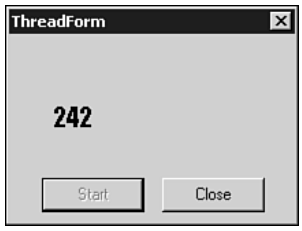


Рис. 4.3. Приложение `LabelThread`.

Листинг 4.24. Объект потока `TLabelThread`

```
#include <vcl.h>
#pragma hdrstop

#include "ThreadFormUnit.h"
#pragma package(smart_init)

#include <Classes.hpp>

class TLabelThread : public TThread
{
private:
protected:
   int Num;
   void __fastcall Execute();
   void __fastcall DisplayLabel();
public:
   __fastcall TLabelThread(bool CreateSuspended);
};
//---------------------------------------------
__fastcall TLabelThread::TLabelThread(bool
   CreateSuspended) : TThread(CreateSuspended)
{
}
```

```
void __fastcall TLabelThread::DisplayLabel()
{
   ThreadForm->Label->Caption = Num;
}

void __fastcall TLabelThread::Execute()
{
   FreeOnTerminate = true;
   for(Num = 0; Num <= 1000; Num++)
   {
      if(Terminated)break;
      Synchronize (DisplayLabel);
   }
}
```

В отличие от примера TRandomThread, где используется бесконечный цикл, в этом проекте работа потока прекращается после достижения значения 1000. С помощью события OnTerminate объекта TLabelThread можно определить этот момент и, соответственно, момент прекращения работы потока.

```
void __fastcall TThreadForm::bStartClick(TObject *)
{
   Thread = new TLabelThread(false);
   Thread->OnTerminate = OnTerminate;
   Start->Enabled = false;
}
void __fastcall TThreadForm::OnTerminate(TObject *)
{
   bStart->Enabled = true;
}
```

Действительно, событие OnTerminate может использоваться в качестве замены метода Synchronize(). Если поток перед закрытием должен выполнить некоторые действия, то событие OnTerminate позволит осуществить доступ к свойствам и методам библиотеки VCL из основного модуля.

Рассмотрим предыдущий пример с кнопкой bStart в обработчике события OnTerminate. Для выполнения этих же действий непосредственно из объекта потока нужно создать более сложный код:

```
//void __fastcall EnableButton();

void __fastcall TLabelThread::EnableButton()
{
   ThreadForm->bStart->Enabled = true;
}

void __fastcall TLabelThread::Execute()
{
   // ...
```

```
        if(Terminated)
        {

           Synchronize(EnableButton);

    // ...
    }
```

Указание приоритета

В приложении с несколькими потоками важно знать, какие потоки обладают более высоким приоритетом и будут запущены первыми. В табл. 4.3 описываются все возможные уровни приоритета.

Таблица 4.3. Приоритеты потоков

Уровень приоритета	Описание
THREAD_PRIORITY_TIME_CRITICAL	На 15 пунктов выше нормального
THREAD_PRIORITY_HIGHEST	На 2 пункта выше нормального
THREAD_PRIORITY_ABOVE_NORMAL	На 1 пункт выше нормального
THREAD_PRIORITY_NORMAL	Нормальный приоритет
THREAD_PRIORITY_BELOW_NORMAL	На 1 пункт ниже нормального
THREAD_PRIORITY_LOWEST	На 2 пункта ниже нормального
THREAD_PRIORITY_IDLE	На 15 пунктов ниже нормального

Все потоки создаются с приоритетом THREAD_PRIORITY_NORMAL. Сразу после создания потока его уровень приоритета можно повысить или понизить с помощью функции SetThreadPriority(). В данном случае следует руководствоваться таким общим правилом: поток, работающий с пользовательским интерфейсом, должен иметь наивысший приоритет, что позволяет приложению активно реагировать на действия пользователя. Фоновые потоки обычно имеют приоритеты THREAD_PRIORITY_BELOW_NORMAL или THREAD_PRIORITY_LOWEST, чтобы они легко могли быть прекращены в любой момент.

На заметку

Уровни приоритета обычно называются *относительными приоритетами* (*relative scheduling priorities*), потому что они выражены в относительных величинах по отношению к уровням других потоков одного процесса.

Объект TThread содержит свойство Priority, которое определяет уровень приоритета потока. Оно может иметь следующие значения:

tpTimeCritical
tpHighest
tpHigher
tpNormal
tpLower
tpLowest
tpIdle

Как видите, эти значения очень близко соответствуют описанным выше уровням приоритета.

Рассмотрим пример, чтобы вы осознали важность уровней приоритета потоков. Запустите новое приложение и создайте в нем два индикатора выполнения (для свойства `Max` задайте значение `5000`) и кнопку `Start`. Попробуем увеличить положение индикатора с помощью потоков с разными приоритетами. В листинге 4.25 содержится исходный код объекта потока `TPriorityThread`.

Листинг 4.25. Объект потока `TPriorityThread`

```
#include <vcl.h>
#pragma hdrstop

#include "PriorityThreadUnit.h"
#include "ThreadFormUnit.h"
#pragma package(smart_init)

__fastcall TPriorityThread::TPriorityThread(bool
   Temp) : TThread(false)
{
   First = Temp;
}

void __fastcall TPriorityThread::DisplayProgress()
{
   if(First)
      ThreadForm->ProgressBar1->Position++;
   else
      ThreadForm->ProgressBar2->Position++;
}

void __fastcall TPriorityThread::Execute()
{
   FreeOnTerminate = true;
   for( Num = 0; Num <= 5000; Num++)
   {
      if(Terminated) break;
      Synchronize (DisplayProgress);
   }
}
```

Обратите внимание, что конструктор `TPriorityThread` слегка изменен. Логическая переменная `Temp` (которая заменяет `CreateSuspended`) указывает на тот индикатор выполнения, к которому следует осуществить доступ.

Основной модуль содержит только код обработчика события щелчка `OnClick` на кнопке `Start`:

```
void __fastcall TThreadForm::bStartClick(TObject *)
{
   TPriorityThread *First;
   First = new TPriorityThread (true);
   First->Priority = tpLowest;
```

```
   TPriorityThread *Second;
   Second = new TPriorityThread(false);
   Second->Priority = tpLowest;

   bStart->Enabled = false;
}
```

Запустите программу и щелкните на кнопке `Start`. При этом оба индикатора выполнения достигнут конца приблизительно в одно и то же время, как показано на рис. 4.4. Теперь установите приоритет `tpLower` для первого потока. К чему это приведет? Результат этого изменения кода показан на рис. 4.5.

Рис. 4.4. Приложение с потоками одинакового приоритета

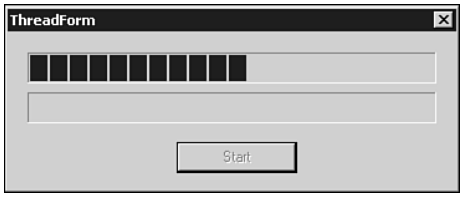


Рис. 4.5. Приложение с потоками разного приоритета

Контроль времени работы потоков

Иногда при создании кода нужно организовать контроль времени работы определенных фрагментов кода. Основной принцип заключается в том, чтобы записать показания системного времени до и после выполнения кода, а потом вычесть начальное время из конечного. В обычных приложениях это можно сделать с помощью Win32 API-функции `GetTickCount()`, которая показана в листинге 4.26.

Листинг 4.26. Контроль времени с помощью функции `GetTickCount()`

```
int Start = GetTickCount();

// ...
Form1->Canvas->Lock();
for(int x = 0; x <= 100000; x++)
    Form1->Canvas->TextOut(10, 10, x);
Form1->Canvas->Unlock();
//...

int Total = GetTickCount() - Start;
ShowMessage(FloatToStr(Total / 1000.0) + "sec");
```

Аналогичный пример можно найти в разделе о ручном контроле времени в главе 6 при описании компиляции и оптимизации приложения с помощью функции `clock()` вместо `GetTickCount()`. Для контроля за временем выполнения кода можно использовать и другие функции.

К сожалению, из-за принципа вытесняющей многозадачности в Windows выполнение потоков часто прерывается. Поэтому нельзя использовать функцию `GetTickCount()` для извлечения информации о времени работы потока. Однако в Windows предусмотрена API-функция `GetThreadTimes()`, которая помогает контролировать чистое время работы потока.

```
BOOL GetThreadTimes(
HANDLE hThread,
LPFILETIME lpCreationTime,
LPFILETIME lpExitTime,
LPFILETIME lpKernelTime,
LPFILETIME lpUserTime
);
```

Функция `GetThreadTimes()` доступна только в Windows NT/2000.

Как видите, в функции `GetThreadTimes()` использована структура `FILETIME`. До выполнения арифметических операций необходимо сохранить пользовательскую информацию о времени в структуре `LARGE_INTEGER`. Затем, вычитая 64-разрядные члены `QuadPart` структуры `LARGE_INTEGER`, можно получить значение времени работы кода в единицах, кратных 100 наносекундам, как показано в листинге 4.27.

Листинг 4.27. Пример использования функции `GetThreadTimes()`

```
FILETIME CreationTime, ExitTime, KernelTime;
union {
   LARGE_INTEGER iUT;
   FILETIME fUT;
} UserTimeS, UserTimeE;

GetThreadTimes((HANDLE)Handle, &CreationTime,
   &ExitTime, &KernelTime, &UserTimeS.fUT);
// ...
Form1->Canvas->Lock();
for(int x = 0; x <= 100000; x++)
   Form1->Canvas->TextOut(10, 10, x);
   Form1->Canvas->Unlock();
// ...

GetThreadTimes((HANDLE)Handle, &CreationTime,
   &ExitTime, &KernelTime, &UserTimeE.fUT);

float Total = UserTimeE.iUT.QuadPart - UserTimeS.iUT.QuadPart;
Total /= 10 * 1000 * 1000; // Преобразование в секунды

OutputDebugString(FloatToStr(Total).c_str());
```

`OutputDebugString()` — очень полезная API-функция, с помощью которой можно послать строку в окно отладки событий Event Log. В обычных условиях автор предпочитает использовать диалоговые окна с сообщениями или изменять заголовок окна, но в многопоточных приложениях эти действия иногда могут быть чрезвычайно опасными без достаточно исчерпывающей обработки в коде программы. Следовательно, функция `OutputDebugString()` является отличным альтернативным вариантом вывода отладочной информации. Более подробно отладка приложения описывается в главе 7.

Синхронизация потоков

Вероятно, самым большим недостатком использования потоков является трудность координации их совместной работы. Допустим, что в приложении одновременно выполняются два потока, которые изменяют глобальные данные. Что случится, если они попытаются одновременно осуществить доступ к одними тем же данным? Или что произойдет, если второй поток вынужден ожидать завершения обработки данных первым потоком? Для координации потоков в Windows предусмотрено несколько способов синхронизации их работы.

Критические разделы

В качестве примера рассмотрим приложение с двумя потоками на основе объекта `TCriticalThread`, которые пытаются одновременно осуществить доступ к одними тем же глобальным данным, как показано в листинге 4.28.

Листинг 4.28. Использование объекта потока `TCriticalThread` из файла `CriticalThreadUnit.cpp`

```
#include <vcl.h>
#pragma hdrstop

#include "CriticalThreadUnit.h"
#include "ThreadFormUnit.h"

#pragma package(smart_init)

__fastcall TCriticalThread::TCriticalThread(bool CreateSuspended)
   :TThread(CreateSuspended)
{
}

void __fastcall TCriticalThread::DisplayList()
{
   ThreadForm->ListBox->Items->Add(Text);
}

void __fastcall TCriticalThread::Execute()
{
   FreeOnTerminate = true;
   for(int x = 0; x <= 50; x++)
   {
      if(Terminated)break;
      //EnterCriticalSection(&ThreadForm->CS);
      Sleep(50);
      ThreadForm->ListText.Insert("=====", 1);
      Text = ThreadForm->ListText;
      Synchronize(DisplayList);
      ThreadForm->ListText.SetLength(ThreadForm->ListText.Length()-5);
      //LeaveCriticalSection(&ThreadForm->CS);
   }
}
```

А в основном модуле создадим два экземпляра этого объекта, как показано в листинге 4.29.

Листинг 4.29. Основной модуль `ThreadFormUnit.cpp`

```
#include <vcl.h>
#pragma hdrstop

#include "ThreadFormUnit.h"
#include "CriticalThreadUnit.h"

#pragma package(smart_init)
#pragma resource "*.dfm"

TThreadForm *ThreadForm;

__fastcall TThreadForm::TThreadForm(TComponent*Owner)
   : TForm(Owner)
{
   ListText = "=====";
   //InitializeCriticalSection(&CS);
}

void __fastcall TThreadForm::StartClick(TObject *Sender)
{
   TCriticalThread *FirstThread;
   FirstThread = new TCriticalThread(false);
   TCriticalThread *SecondThread;
   SecondThread = new TCriticalThread(false);
}

void __fastcall TThreadForm::FormClose(TObject *,
                                       TCloseAction &Action)
{
   //DeleteCriticalSection(&CS);
}
```

Этот простой (и бесполезный) код демонстрирует важность синхронизации потоков. Сначала объект `TCriticalThread` добавляет в глобальный тестовый список `ListText` пять символов равенства (=). А затем отображает значения `ListText` в диалоговом окне списка `ListBox`. Наконец, объект `TCriticalThread()` обрезает пять символов, возвращая текстовый список `ListText` в исходное состояние. Логично было бы предположить, что все элементы в диалоговом окне списка `ListBox` должны иметь вид `==========`, но как показано на рис. 4.6, это не всегда верно. Почему? Дело в том, что второй поток также обладает доступом к той же глобальной переменной.

```
VOID InitializeCriticalSection(
    LPCRITICAL_SECTION lpCriticalSection
);

VOID EnterCriticalSection(
    LPCRITICAL_SECTION lpCriticalSection
);
```

```
VOID LeaveCriticalSection(
    LPCRITICAL_SECTION lpCriticalSection
);

VOID DeleteCriticalSection(
    LPCRITICAL_SECTION lpCriticalSection
);
```

Нетрудно догадаться, как используются эти функции. Сначала следует объявить переменную типа `CRITICAL_SECTION`. Эта переменная инициализируется во время запуска программы (функция `InitializeCriticalSection()`) и удаляется при закрытии программы (`DeleteCriticalSection()`). Когда поток начинает обработку данных, доступ к ним блокируется с помощью функции `EnterCriticalSection()`; после завершения обработки критический раздел закрывается.

Вернемся к листингам 4.28 и 4.29 и закомментируем четыре строки, в которых вызываются описанные выше функции критического раздела. Затем откроем заголовочный файл основного модуля и добавим в него следующую строку.

```
CRITICAL_SECTION CS;
```

Как показано на рис. 4.7, теперь все элементы диалогового окна списка `ListBox` содержат одинаковые строки.

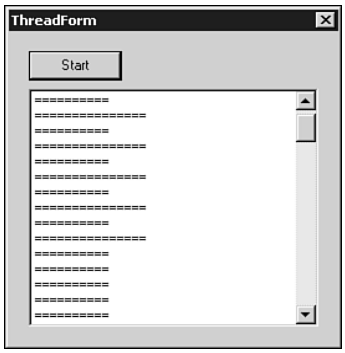


Рис. 4.6. Вид приложения `CriticalThread` без использования критических разделов

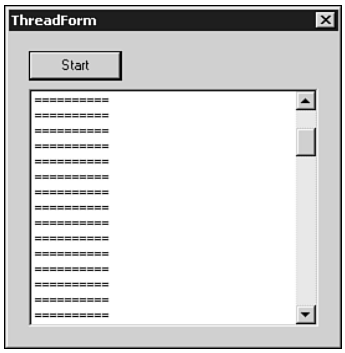


Рис. 4.7. Проект `CriticalThread` с критическими разделами

Мьютексы

Мьютексы (mutex) выполняют те же функции, что и критические разделы, однако имеют несколько дополнительных компонентов.

Хотя критические разделы не обладают такими богатыми возможностями, они выполняются быстрее мьютексов и семафоров. Если время является важным фактором работы приложения, то следует рассмотреть возможность использования критических разделов.

Объекты-мьютексы создаются с помощью API-функции `CreateMutex()`.

```
HANDLE CreateMutex(
    LPSECURITY_ATTRIBUTES lpMutexAttributes,
    BOOL bInitialOwner,
    LPCTSTR lpName
);
```

Создав объект-мьютекс, необходимо использовать API-функцию `WaitForSingleObject()`; она позволяет получить права владения этим объектом-мьютексом, перейти в режим ожидания открытия доступа к нему и использовать мьютекс до тех пор, пока не будет вызвана функция `ReleaseMutex()`.

```
HANDLE Mutex;
Mutex = CreateMutex(NULL, false, NULL);
if(Mutex == NULL)
{
   ShowMessage("Cannot create mutex!");
// ShowMessage("Мьютекс создать нельзя!");
   return;
}

// ...

if(WaitForSingleObject(Mutex, INFINITE) == WAIT_OBJECT_0)
{
   // ...
}

ReleaseMutex(Mutex);
```

На заметку В отличие от критических разделов, один и тот же мьютекс может использоваться двумя или более процессами.

На заметку Если поток не передает права владения мьютекс-объектом, то этот мьютекс рассматривается как "брошенный". В этом случае функция `WaitForSingleObject()` возвратит значение `WAIT_ABANDONED`. Хотя это не совсем безопасно, всегда можно организовать получение прав владения брошенным мьютексом.

Другие вопросы

Для синхронизации также можно использовать такие объекты, как, например, семафоры и таймеры. Знание принципов работы с критическими разделами и мьютексами, позволяет продвинуться на шаг вперед в области синхронизации потоков.

Шаблоны

Использование шаблонов для проектирования понятных и легких в сопровождении архитектур программного обеспечения стало особенно популярным в течение последних нескольких лет. В этом разделе рассматриваются преимущества использования шаблонов.

Рекуррентная природа шаблонов

Все, с чем нам приходиться иметь дело, так или иначе связано с шаблонами. Вообще говоря, шаблоном (pattern) называется нечто, что помогает идентифицировать нашу деятельность и окружающие сущности. Шаблон также является отправной точкой для повторного использования явлений и сущностей.

Иногда, при изложении какой-либо идеи, достаточно обсудить только часть проблемы, а другую — собеседник воссоздает самостоятельно в силу того, что она хорошо известна ему и вам, а потому принимается как должное.

Шаблоны можно использовать повторно. Иногда они определяются произвольно, причем могут сильно отличаться друг от друга.

Рекуррентные шаблоны для создания программного обеспечения

По определению языки программирования не могут обладать такими же степенями свободы. Они требуют конкретного указания всей информации о способе работы программы. Программа, которая могла бы самостоятельно определять способ своей работы, была бы крайне непредсказуемой. Однако, туманное представление о задачах проектирования и их решениях очень характерно для многих разработчиков Такие высокоуровневые объектно-ориентированные языки программирования, как C++, имеют гораздо более выразительные средства, чем другие языки, но не предоставляют эффективных инструментов для обмена идеями среди разработчиков программного обеспечения.

В 1995 году группа ученых в области информатики, Эрик Гамма (Erich Gamma), Ричард Хелм (Richard Helm), Ральф Джонсон (Ralph Johnson) и Джон Влисидис (John Vlissides), широко известных как “банда четырех” (“The Gang Of Four”), опубликовали историческую книгу *Design Patterns: Elements of Reusable Object-Oriented Software*. В этой книге они предложили совершенно новый способ сотрудничества программистов, включающий обдумывание, обсуждение и документирование проектов. Его философия заключается в том, что наиболее сложные архитектуры программного обеспечения можно разбить на несколько более простых элементов. Эти элементы вряд ли будут уникальными, наоборот, вероятно, они уже существуют в той или иной форме во многих других проектах. Шаблоны можно представить себе как очень общие очертания повторяющейся конкретной проблемы. Подгоняя и комбинируя отдельные шаблоны, сложное программное обеспечение можно спроектировать наиболее соответствующим способом. Вот цитата из этой книги.

> С помощью шаблона именуется и объясняется общая структура, что призвано решить повторяющуюся проблему проектирования в объектно-ориентированных системах. С его помощью описывается проблема, применяемое решение, а также последствия его применения. В нем также указывается возможная реализация и приводятся ее примеры. Решение — это в данном случае общий набор объектов и классов, которые необходимы для решения проблемы. Это решение настроено и реализовано с учетом конкретного контекста.

Книга содержит подробное введение в шаблоны, включая учебный пример, в котором показано, как эта методика может быть воплощена на практике в рабочем проекте. Она также содержит каталог из 23 шаблонов, которые могут быть включены в разрабатываемые вами проекты.

Шаблоны в качестве словаря

Достаточно опытный программист, вероятно, найдет в этом каталоге шаблоны, которые уже использовались им. Например, если проект содержит некую часть, которую можно описать как “класс, имеющий один экземпляр с глобальным доступом”, то эту часть можно назвать одноэлементной. Для описания именно такого класса Гамма и др. предложили термин *одноэлементный шаблон*, или *Singleton*. В результате для описания проекта можно предло-

жить более краткий словарь. Как и при использовании обычного словаря, любые сущности становятся более понятными, если их удается кратко и точно описать. Шаблоны позволяют достаточно подробно и в то же время кратко описывать такие сущности.

Формат шаблона

Все шаблоны имеют одинаковый формат, позволяющий легко найти их сходства и различия. Этот формат хорошо документирован в книге Гаммы и др. и спроектирован так, чтобы можно было быстро найти подходящий шаблон (или несколько шаблонов) для любой задачи проектирования. Документация о шаблоне разделена на несколько разделов.

- **Intent** (Назначение). В этом разделе резюмируются проблемы, для решения которых предназначен шаблон. Например, шаблон Observer имеет следующее назначение: “Определить зависимость один-ко-многим между объектами так, чтобы при смене состояния одного объекта автоматически уведомлялись и обновлялись все зависящие от него объекты”.
- **Also Known As** (Псевдонимы). Это список других имен, под которыми известен шаблон.
- **Motivation** (Мотивация). Обычно это несколько абзацев и иллюстраций, описывающих ситуации, в которых может применяться данный шаблон. В качестве примера обычно приводится учебный пример, иногда с диаграммой классов, на которой показаны результаты применения и настройки шаблона для конкретного случая. В этом разделе объясняются способы воплощения шаблона на практике в реальном проекте.
- **Applicability** (Применимость). Ряд замечаний, которые помогают оценить применимость шаблона в конкретной ситуации. Этот раздел обычно начинается со слов: “Используйте шаблон X, в случаях, когда ...” (“Use pattern X when….”).
- **Structure** (Структура). Диаграмма классов, которая выражает суть шаблона. Если раздел **Motivation** содержит описание практического применения шаблона с несколькими реальными примерами, то в разделе **Structure** приведена очень общая диаграмма.
- **Participants** (Участники). Краткое описание классов-участников, которые перечислены в разделе **Structure**. В нем приводятся перекрестные ссылки на имена классов и соответствующие термины-примеры из раздела **Motivation**.
- **Collaborations** (Взаимодействие). Краткое описание способов взаимодействия участников (например, какие функции вызывает каждый класс из другого класса, а также как и почему это делается).
- **Consequences** (Последствия). Список преимуществ или недостатков, связанных с использованием этого шаблона. Этот раздел поможет определить наиболее оптимальный метод применения шаблона в проекте, а также степень его влияния на дальнейшее создание программного обеспечения. При активном использовании шаблонов всегда следует иметь в виду возможность принятия компромиссных решений.
- **Implementation** (Реализация). Способ воплощения шаблона в программном обеспечении. В этом месте обычно впервые указываются элементы языков программирования.
- **Sample Code** (Образец кода). Он обычно приводится на языке C++, хотя иногда примеры кода даются на других языках, например Smalltalk.
- **Known Uses** (Известные способы применения). Описание способов применения шаблона в существующих коммерческих проектах.
- **Related Patterns** (Родственные шаблоны). Шаблон обычно лучше всего применять вместе с другими шаблонами, перечисленными в этом разделе документации.

Классификация шаблонов

Типы повторяющихся проблем при создании объектно-ориентированного программного обеспечения можно классифицировать по нескольким видам. Естественно, при этом нужно следовать некоторой объектно-ориентированной парадигме, согласно которой в программе могут создаваться объекты разных классов, каждый из которых обладает определенными структурными свойствами и тенденциями поведения по отношению к другим.

Созидающие шаблоны

Эти шаблоны предназначены для создания объекта. Например, шаблон Singleton может применяться в тех случаях. когда в приложении требуется создать только один экземпляр класса.

Кроме того, шаблон Factory Method является еще одним созидающим шаблоном, который часто используется для создания объекта, класс которого неизвестен до запуска программы, а шаблон Prototype используется в тех случаях, когда нужно создать новые объекты на основе уже существующих экземпляров.

Структурные шаблоны

Структура сложных объектов в объектно-ориентированной схеме может быть реализована двумя способами. Во-первых, она может стать результатом наследования после добавления структурных свойств в класс с образованием его подкласса. Во-вторых, она может возникнуть как результат композиции после определения структуры объекта на основе других дочерних объектов.

Пользователям библиотеки C++Builder VCL наверняка знаком шаблон Adapter (адаптер), который иногда называют шаблоном Wrapper (оболочка). Он позволяет создать альтернативный интерфейс для объекта, когда существующий интерфейс несовместим с другими частями системы. В библиотеке VCL класс `TWinControl` является адаптером, который Windows API-интерфейс на основе дескрипторов и сообщений заменяет интерфейсом на основе модели свойств, событий и методов. Он применяется в качестве базового класса для повторно используемых компонентов, которые представляют элементы управления оконной среды.

Другой фундаментальной целью проектирования является разделение интерфейса объекта и его реализации, чтобы их можно было изменять независимо. Например, это позволяет изменять реализацию без разрушения существующего кода, который работает с объектом. С точки зрения широкой перспективы, с учетом эволюции и роста системы рекомендуется разбить ее на независимые подсистемы, чтобы упростить ее сопровождение и сократить время обучения методам работы с ней. Для достижения этих целей рекомендуется использовать такие шаблоны, как Bridge и Facade.

Поведенческие шаблоны

Поведенческие шаблоны связаны с представлением алгоритмов в программе и потока управления в системе. Такие фундаментальные поведенческие концепции в объектно-ориентированном проектировании, как полиморфизм, обеспечивают уровень контроля над операциями, которые выполняются в приложении, а также условиями их выполнения. Например, полиморфизм позволяет во время выполнения приложения производить различные операции над объектом, в зависимости от его типа. Хотя часто нужно организовать более сложное поведение, которое непосредственно не поддерживается объектно-ориентированными принципами.

Одно частное ограничение объекта заключается в том, что его класс фиксирован в течение всего периода жизни, а это означает фиксированную реализацию операций, которые его

поддерживают. Это может привести к возникновению проблем, когда эти операции потребуется изменить в соответствии с состоянием объекта. Очевидным решением было бы переписать эти операции таким образом, чтобы они сначала запрашивали информацию о состоянии и соответственно выполняли разные действия. Однако это приведет к созданию громоздких и трудных для понимания функций. Более удачное решение можно было бы создать на основе инкапсуляции семейств операций в отдельных классах. В таком случае при любых изменениях состояния объекта потребуется лишь переконфигурировать его с помощью нового экземпляра одного из этих классов. Операции объекта могут затем прозрачно передавать запросы к соответствующей операции текущей реализации объекта, что создает иллюзию изменения его класса. Именно для этого используется шаблон *State* (состояние).

Другой поведенческий шаблон *Observer* (наблюдатель) имеет очень большое значение в приложениях, основанных на документах и их графических представлениях. Первый пример такого шаблона можно узнать в IDE-среде C++Builder. При добавлении или удалении компонентов из формы в режиме конструктора форм объявления этих компонентов добавляются в объявление класса формы в виде необходимых директив о включении заголовочных файлов и компоновки пакетов. При этом визуальные формы и код на языке C++ тесно связаны друг с другом, что позволяет синхронизировать их. Можно с 99-процентной гарантией утверждать, что это возможно только благодаря применению шаблона Observer. Этот шаблон позволяет организовать работу двух или более различных объектов, которые могут независимо друг от друга работать с совместно используемым объектом. Причем последний уведомляет всех своих пользователей-обозревателей (“observers”) о происходящих в нем изменениях. В случае IDE-среды совместно используемым объектом является внутреннее представление самой формы (т.е. объекта, который хранится в DFM-файле), а обозревателями — объекты, которые управляют работой конструктора форм Form Designer и редактора кода Code Editor.

Заключительные замечания о шаблонах

Это краткое введение в шаблоны должно напомнить о важности серьезного подхода к проектированию программного обеспечения. C++Builder считается инструментом быстрой разработки программного обеспечения, а потому у разработчиков иногда создается ложное впечатление, что этап планирования является излишней тратой ресурсов, потому что для многих задач можно с помощью IDE-среды достаточно легко создать рабочий прототип. Однако при создании более совершенных программ очень важно избежать создания программного обеспечения на основе компонентов, которые вскоре могут быть несовместимы с новыми изменениями условий их работы.

Серьезный подход к проектированию может оказаться жизненно важным, а шаблоны — источником вдохновения для поиска свежих идей и способом слежения за долговременной реализацией проекта с учетом текущих нужд. Шаблоны не диктуют методологию программирования приложений, поэтому для их использования вовсе не нужно пересматривать привычные приемы программирования. Шаблон скорее похож на набор прекрасных идей, которые помогают создавать более удачное и легко сопровождаемое программное обеспечение.

Резюме

В этой главе описываются сложные методы программирования с помощью C++Builder. Помимо прочего, в ней описаны способы использования стандартной библиотеки шаблонов Standard Template Library, интеллектуальных и строгих указателей, способы реализации усовершенствованных обработчиков исключительных ситуаций, методы создания многопотоковых приложений, а также применение шаблонов.

Библиотека SCL содержит классы-контейнеры, которые упрощают манипулирование группами элементов. При работе с проектами разработчикам следует всегда учитывать это, чтобы не приходилось заново изобретать колесо.

Парадигма строгого управления памятью позволяет экономить время при создании кода, безопасного по отношению к исключительным ситуациям, а также *исключает* поиск причин утечки памяти. Благодаря этому, так никогда и не придется узнать, сколько ужасных дней и ночей вам удалось избежать.

Несмотря на то что обработчик исключительных ситуаций, используемый компилятором по умолчанию, может вполне адекватно подходить для простых и небольших программ, он все же может оказаться очень грубым для других проектов. Способность собирать информацию о состоянии системы в заданный момент времени и сохранять ее для последующего изучения является основным условием успешного сопровождения программного продукта.

Шаблоны являются отправными точками для обработки часто встречающихся и похожих ситуаций. При разработке программного обеспечения шаблоны позволяют разработчику не только понять и решить конкретные проблемы, но и воспользоваться предлагаемыми советами и примерами, а также обрести источник вдохновения.

Разработчики часто путают термины *многозадачность* и *многопотоковость*. Многозадачность — это способность операционной системы одновременно запускать несколько программ, а многопотоковость — способность программы одновременно запускать несколько задач (потоков). Следует заметить, что в большинстве приложений Windows используется только один поток.

Глава

5

Принципы и методы создания интерфейса пользователя

Джэйми Оллсоп
Зе Ксианг Ву

Одним из самых значительных преимуществ C++Builder является способность визуального создания интерфейса пользователя приложения. Для того чтобы созданный интерфейс удовлетворял требованиям и ожиданиям пользователей, необходимо придерживаться определенных принципов. Обладая необходимыми знаниями о том, как следует создавать интерфейс пользователя в соответствии с этими требованиями, вы сможете создавать интуитивно понятные и простые в использовании приложения.

В первом разделе этой главы рассматриваются принципы проектирования интерфейса пользователя и предлагаются рекомендации для удовлетворения предъявляемых к нему требований. Остальная часть этой главы содержит советы по поводу реализации рекомендаций, представленных в первом разделе главы. В ней используется простой пример приложения для демонстрации разнообразных методов, связанных с проектированием и созданием интерфейса пользователя. В тех случаях, когда излагаемый материал не имеет отношения к данному простому примеру приложения, предлагаются другие альтернативные примеры.

Принципы создания интерфейса пользователя

Как уже говорилось выше, при проектировании интерфейса пользователя следует придерживаться определенных принципов и рекомендаций. Хотя большая их часть основана на соображениях здравого смысла, среди них есть и не такие очевидные. В приведенном ниже списке представлены основные рекомендации, которые следует иметь в виду при работе над приложением. (Порядок их перечисления не имеет никакого значения.)

- Соответствие ожиданиям пользователя. Приложение, позволяющее пользователю выполнять определенную задачу, должно функционировать именно так, как предполагается пользователем. Если интерфейс отвечает ожиданиям пользователя, ему будет очень удобно работать с ним.
- Простота и ясность интерфейса. Внешний вид интерфейса должен очевидным образом выражать функциональность приложения и позволять перемещаться от одной части интерфейса к другой. Простой и понятный интерфейс не отвлекает внимание пользователя от выполнения ключевых задач. Для этого следует группировать элементы управления, в то же время не создавая областей с чрезвычайно высокой их плотностью. Кроме того, следует убедиться в том, что контролируемые мышью элементы управления имеют достаточно большой размер, что позволит легко и безошибочно выполнять перемещения и щелчки мышью. Не следует создавать очень большие промежутки между часто используемыми пользовательскими элементами управления. В противном случае пользователю будет очень сложно работать: в первом случае из-за излишне высокой концентрации, а во втором — из-за дополнительных перемещений мыши.
- Интуитивно понятный и знакомый интерфейс. Попытайтесь сделать интерфейс таким, чтобы пользователи смогли догадаться о выполнении какой-либо задачи без необходимости специального обучения. Вряд ли пользователи будут знать все особенности вашего интерфейса, поэтому нельзя избежать того, что в какой-то момент они встретят нечто незнакомое в нем. Если такой момент возникнет в самом начале знакомства с вашей программой и пользователь не сможет разобраться в ситуации, то высока вероятность того, что он откажется от дальнейшего использования вашего приложения. Если же предоставить пользователям интуитивно понятный интерфейс, они смогут гораздо быстрее научиться пользоваться даже самыми сложными компонентами приложения. Это особенно важно в тех случаях, когда пользователь пытается определить

потенциал вашего приложения и его пригодность для решения своих задач. Дополнительное условие состоит в том, чтобы выполняемые с интерфейсом пользователя действия были логически связаны. Иначе говоря, интерфейс должен функционировать именно в том порядке, который понятен пользователю. Это поможет ему почувствовать себя более уверенно и комфортно при работе с вашим приложением.

- Дружественный к пользователю интерфейс. Если вам удастся создать дружественный к пользователю интерфейс, он не откажется потратить некоторое время на обучение методам работы с ним. Желание потратить время на обучение методам работы с интуитивно понятным и знакомым интерфейсом может в значительной степени повлиять на успех обучения приемам работы с вашим приложением, что позволит максимально быстро использовать его для практической работы.
- Обратная связь с пользователем. Обеспечение обратной связи с пользователем укрепляет его доверие и позволяет удостовериться в правильности выполняемых им действий. Обратная связь помогает сохранить интерес пользователя при выполнении рутинных или длительных задач. Это очень важный компонент любого интерфейса пользователя, а потому всегда нужно стараться обеспечить полезную и конкретную обратную связь с пользователем. Обратная связь может принимать разные формы. Некоторые из них очевидны, например нажатая кнопка после щелчка на ней. Такие мелочи не следует рассматривать как необязательные. С фундаментальной точки зрения интерфейс пользователя в целом обеспечивает обратную связь с пользователем. Дополняя обратную связь другими компонентами, можно усилить восприятие пользователя. В некоторых случаях дополнительная обратная связь может оказаться очень существенной, например, когда пользователь не может выполнить какие-либо действия (этот вопрос рассматривается в следующем пункте).
- Максимально доступный интерфейс пользователя. Это означает организацию разных типов обратной связи и форм ввода данных, удовлетворяющих потребностям пользователей, которые могут испытывать трудности при использовании обычных подходов. Например, для пользователей со слабым зрением можно предложить дополнительную звуковую обратную связь, а для пользователей, которые с трудом работают со стандартными устройствами ввода, можно создать устройства ввода на основе распознавания речи.
- Справочная служба. Иногда пользователь попадает в затруднительную ситуацию. Для того чтобы выйти из нее можно использовать следующие два способа: предложить достаточно полную документацию и организовать службу поддержки. Часть этой документации всегда следует предоставить пользователю в виде интерактивной справки или в распечатанном виде. Более подробно вопросы создания документации приложения рассматриваются в главе 27. По возможности следует также организовать службу поддержки. Служба поддержки может варьироваться от совсем простой, на основе электронной почты, до круглосуточной поддержки со стороны специалистов, оказывающих консультативную помощь по телефону или непосредственно, с выездом на рабочее место. Консультативную помощь по электронной почте следует рассматривать как обязательный вид службы поддержки.
- Возможность настройки. Следует предоставить пользователю возможность настраивать интерфейс по своему усмотрению. Такие простые действия, как выбор цвета, могут оказать существенное влияние на восприятие интерфейса пользователем. Использование системных цветов, например `clMenu` (для команд меню), позволяет придать интерфейсу приятный для пользователя внешний вид. Но эта настройка позволяет не только перемещать компоненты интерфейса для придания ему более привлекательного внешнего вида, но и для организации удобного доступа к тем элементам, которые ис-

пользуются наиболее часто. Вряд ли все пользователи будут использовать ваше приложение одинаковым образом. Возможность изменять интерфейс так, чтобы пользователь мог работать в привычном для себя стиле, может оказать существенное влияние на выбор им вашего приложения в качестве предпочтительного инструмента.

- Предоставление пользователю возможности отказаться от выполненных действий. Даже очень опытный пользователь может допустить ошибку. Поэтому в приложении следует предусмотреть возможность отказа от ошибочно выполненной операции. В наиболее распространенном виде это означает предоставление функции отмены выполненного действия, которая стала практически стандартным компонентом всех интерфейсов с элементами редактирования.
- Ясное и четкое информирование пользователя об ошибке. Независимо от степени защищенности приложения от некорректных действий пользователя, он всегда может сделать что-то, что приведет к возникновению ошибки. Обычной реакцией на возникновение ошибки является отображение диалогового окна с сообщением для пользователя о том, что произошла ошибка, и как ее устранить. Иногда пользователю можно даже предоставить возможность завершить операцию, которая привела к возникновению ошибки. Одна из наиболее неприятных сторон этой ситуации заключается в том, что обычно диалоговое окно с таким сообщением содержит зашифрованное сообщение, которое совершенно непонятно даже разработчику приложения. Следовательно, нужно предоставить пользователю конструктивную информацию о любых возникающих ошибках. По возможности попытайтесь полностью предотвратить влияние ошибки на работу пользователя за счет ее перехвата. Проблема в данном случае состоит в том, что очень трудно идентифицировать все возможные причины возникновения ошибок и либо изолировать пользователя от ошибки, либо предоставить конструктивную обратную связь с информацией об ошибке. Профессиональная обработка возникающих ошибок в интерфейсе пользователя часто является областью постоянных усовершенствований.
- Использование символов, изображений и цвета для создания более привлекательного и простого в использовании интерфейса пользователя. Символы позволяют ускорить работу с интерфейсом, потому что распознавание символа происходит быстрее, чем чтение текста. Обычно наряду с символом используется поясняющий его текст. Сначала для выполнения тех или иных действий пользователи полагаются на текстовое описание, а затем, по мере привыкания к этим символам, могут использовать их вместо текстового описания. Символы должны последовательно использоваться во всем интерфейсе для достижения максимальной эффективности работы. Причем они должны быть тщательно продуманы, чтобы при взгляде на них был очевиден скрытый за ними смысл. Для улучшения восприятия интерфейса пользователя также могут использоваться изображения, либо как графические элементы, обладающие функциональными возможностями, либо просто для создания более привлекательного внешнего вида. При редактировании изображений следует проявлять осторожность. Например, цвет можно использовать для группирования связанных элементов управления, их визуального отделения или привнесения дополнительной информации, например, выделения синтаксических элементов в IDE-среде.
- Использование всех устройств ввода. Разные пользователи по-разному выполняют одни и те же действия. Для некоторых определенные устройства ввода данных могут оказаться более удобными, чем другие. В интерфейсе пользователя должно быть предусмотрено использование указательного (например, мыши) и клавиатурного устройства. При этом следует соблюдать общепринятые соглашения. Например, если в интерфейсе предусмотрена функция копирования, то пользователь должен быть уверен в том, что для копирования может быть использована клавиатура, а именно комбинация клавиш <Ctrl+C>.

Как уже говорилось, приведенные выше рекомендации основаны на здравом смысле, но все же не помешает напомнить о них еще раз. При интенсивной работе по созданию и отладке сложного кода можно легко упустить то, что некоторые пользователи говорят на другом языке или находятся совсем в другой стране. Поэтому им следует четко и ясно сообщить о том, что происходит, и как следует поступить в такой ситуации.

Часто имеет смысл тщательно проверить созданные интерфейсы, чтобы убедиться в их соответствии перечисленным выше рекомендациям, или, что еще лучше, поручить их тестирование и рецензирование сторонней организации. Также полезно еще раз изучить другие известные вам интерфейсы, чтобы учесть в своей работе их преимущества и недостатки. Иногда знание недостатков интерфейса может быть так же полезно, как и знание его преимуществ. Поэтому изучение слабых сторон интерфейса поможет избежать повторения таких ошибок в будущем.

Проекты, используемые в этой главе

Остальная часть этой главы разбита на несколько разделов, посвященных специальным вопросам, которые возникают при реализации пользовательского интерфейса в приложениях. Каждый раздел посвящен рассмотрению отдельной темы. В большинстве этих разделов рассматривается приложение MiniCalculator, которое было создано для демонстрации рассматриваемых в этой главе тем. Это приложение вместе с полными версиями исходного кода и всеми изображениями можно найти на прилагаемом к книге компакт-диске. На нем также находится выполняемый файл приложения (`MiniCalculator.exe`), который используется для запуска и тестирования работы программы. В других разделах, не связанных непосредственно с проектом MiniCalculator, используются другие проекты, которые перечислены и кратко описаны в табл. 5.1. Обратите внимание, что некоторые примеры проектов рассматриваются в нескольких разделах, причем именно в том порядке, в котором они указаны в этой таблице.

Таблица 5.1. Проекты, рассматриваемые в этой главе

Проект	Разделы
`Focus.bpr`	“Перемещение фокуса ввода”
`MDIProject.bpr`	“Настройка клиентской области в родительской форме многодокументного интерфейса”, “Активные списки”
`Panels.bpr`	“Выравнивание”, “Закрепление”, “Ограничения”
`ProgressCursor.bpr`	“Элементы управления `TProgressBar` и `TCGauge`”, “Указатель мыши”
`ScreenInfo.bpr`	“Решение проблемы использования разных типов экрана”

Из всех перечисленных в табл. 5.1 проектов особо следует отметить проект `MDIProject`, содержащий приложение с многодокументным интерфейсом или MDI-интерфейсом (multiple-document interface — MDI). Все остальные приложение имеют однодокументный интерфейс или SDI-интерфейс (single-document interface — SDI). SDI-приложения обычно имеют одну основную форму, а другие формы не являются ее дочерними формами, независимо от того, модальные они (как диалоговые окна) или нет. В приложении с MDI-интерфейсом некоторые формы являются дочерними основной (или родительской) формы. Такие формы привязаны к клиентской области родительской формы. Важно то, что родительская форма служит визуальным контейнером для дочерних форм. Дочерние формы не являются модальными, поэто-

му при работе с родительской формой можно переключаться между ее дочерними формами. MDI-приложения обычно содержат некоторые общие функциональные черты, например возможность упорядочения дочерних окон, слияние меню, открытие разных типов дочерних окон. Для демонстрации этих возможностей рассмотрим проект `MDIProject.bpr`.

Проект MiniCalculator

Программа MiniCalculator имеет функциональные возможности калькулятора. Ее интерфейс имеет внешний вид обычного калькулятора с некоторыми дополнительными компонентами, присущими приложению Windows. Это очень важно! Имитирование внешнего вида обычного калькулятора позволяет создать дружественный для пользователя интерфейс, но также налагает некоторые ограничения, которыми обладает калькулятор. Нам бы хотелось создать знакомый интерфейс благодаря такому сходству с обычным калькулятором, но не имеющий ограничений в способах использования и функциональных возможностях, вытекающих из такого сходства. Эта тема будет снова рассмотрена в разделе о повышении удобства и простоты использования программного обеспечения, или практичности (usability), за счет усовершенствования внешнего вида интерфейса.

На рис. 5.1 показано приложение MiniCalculator. Как видите, это приложение обладает графическим интерфейсом и выглядит как обычный калькулятор. Теперь любой пользователь, которому ранее приходилось работать с обычным калькулятором, сможет очень просто осуществить доступ к основным функциям MiniCalculator. Кроме того, любой пользователь, имеющий опыт работы с базовыми приложениями Windows, сможет легко воспользоваться дополнительными функциями, которые предусмотрены в MiniCalculator, поскольку он также является обычным приложением Windows.

Рис. 5.1. Приложение MiniCalculator

Следует заметить, что для создания приложения MiniCalculator использовались только компоненты C++Builder без каких-либо компонентов сторонних разработчиков. Это позволяет продемонстрировать возможности C++Builder. Все компоненты приложения MiniCalculator перечислены в табл. 5.2 в соответствии со вкладками, в которых они находятся в палитре компонентов.

Таблица 5.2. Компоненты проекта MiniCalculator

Вкладка	Компонент

Standard	`TActionList`
	`TMainMenu`
	`Tpanel`
	`TPopupMenu`

Additional	`TApplicationEvents`
	`TBevel`
	`TBitBtn`
	`TControlBar`
	`TImage`
	`TSpeedButton`

Окончание табл. 5.2

Вкладка	Компонент
Win32	`TImageList` `TStatusBar`
Dialog	`TColorDialog`

Бо*л*ьшая часть времени по созданию приложения MiniCalculator связана с оформлением макета и созданием изображений (например, для кнопок). Все изображения в приложении MiniCalculator были созданы с помощью графического редактора Paint Shop Pro фирмы JASC. Пробная версия Paint Shop Pro находится на прилагаемом к книге компакт-диске.

Повышение практичности благодаря обратной связи с пользователем

Организация постоянной обратной связи с пользователем — прекрасный способ улучшить интерфейс приложения. Для этого обычно используется несколько стандартных способов. Для предоставления дополнительной информации о состоянии приложения (например, о расположении курсора) используется строка состояния. Для предоставления советов могут использоваться контекстные подсказки. Там, где это возможно, для слежения за текущим состоянием длительных операций применяются индикаторы выполнения, а также изменяется форма указателя мыши для обозначения текущей функции. Например, указатель мыши в виде песочных часов часто применяется для обозначения неактивного состояния (т.е. возможно только перемещение указателя мыши), а указатель мыши в виде руки — для обозначения гиперссылки у данного элемента управления.

В этом разделе рассмотрены методы для организации обратной связи с пользователем и проиллюстрированы наиболее важные и распространенные способы их реализации.

Индикаторы выполнения `TProgressBar` и `TCGauge`

Индикатор выполнения прекрасно подходит для обозначения состояния выполнения длительной операции. Благодаря ему пользователь может удостовериться в том, что программа функционирует и выполняет поставленную задачу. В C++Builder предусмотрено два типа индикаторов выполнения: `TProgressBar` и `TCGauge`.

Индикатор выполнения `TProgressBar` находится во вкладке Win32, а индикатор выполнения `TCGauge` — во вкладке Samples палитры компонентов Component Palette. В проекте `ProgressCursor.bpr`, который находится на прилагаемом к книге компакт-диске, эти два типа индикаторов представлены в самых разных конфигурациях. Во многих отношениях более предпочтительным является индикатор `TCGauge`. Это связано с простотой представления информации, которая легко поддается количественной оценке.

В этом смысле индикатор `TProgressBar` уступает индикатору `TCGauge`. Применение сегментации в индикаторе `TProgressBar` является одной из причин, по которой он не очень успешно справляется со своей задачей. Дело в том, что сегментация не имеет никакого отношения к действительному состоянию операции. Следовательно, пользователь сможет только догадываться о реальном прогрессе выполняемого действия. Этот индикатор является одним из стандартных элементов управления Win32, а потому знаком многим пользователям. Если точной оценки состояния операции не требуется, то можно считать, что этот индикатор

справляется со своей задачей. Индикатор TCGauge также имеет свои недостатки, но редактируя его исходный код, который находится в файле cgauges.cpp в каталоге $(BCB)\Examples\Controls\Source, всегда можно изменить его должным образом.

Индикаторы TProgressBar и TCGauge имеют практически одинаковый принцип работы. Например, для индикатора ProgressBar типа TProgressBar и индикатора CGauge типа TCGauge можно с помощью приведенного ниже кода указать минимальное и максимальное начальные значения.

```
// Минимальное значение
ProgressBar->Min = 0;
CGauge->MinValue = 0;

// Максимальное значение
ProgressBar->Max = 100;
CGauge->MaxValue = 100;

// Текущее значение
ProgressBar->Position = 0;
CGauge->Progress = 0;
```

Для увеличения текущего значения на единицу следует использовать такой код.

```
// Увеличение текущего значения на единицу
ProgressBar->Position = ProgressBar->Position + 1;
CGauge->Progress = CGauge->Progress + 1;
```

Попытка увеличения максимального значения на единицу для обоих типов индикаторов ни к чему не приведет. Однако в индикаторе TProgressBar также предусмотрены методы StepIt() и StepBy(). Метод StepIt() увеличивает значение переменной Position объекта TProgressBar на величину, равную значению свойства Step. Метод StepBy()увеличивает значение переменной Position объекта TProgressBar на величину, равную значению передаваемого этому методу целочисленного аргумента типа int. В обоих методах при возрастании значения свойства Position до максимального значения Max оно будет сброшено до нуля, а его увеличение вновь продолжится с самого начала, т.е. с нуля. Это можно заметить при работе с проектом ProgressCursor.bpr.

Выбор индикатора выполнения зависит от предпочтений разработчика, однако при этом всегда следует иметь в виду, что выполнение длительных операций всегда должно сопровождаться индикатором, хотя бы для уведомления пользователя о том, что программа функционирует.

Указатель мыши

Изменение формы указателя мыши для организации обратной связи с пользователем или предоставления ему дополнительной информации — это наиболее распространенный прием при программировании пользовательского интерфейса. В C++Builder эту задачу можно легко выполнить с помощью следующих двух способов.

Свойства Cursor и DragCursor указателя мыши TCursor для некоторого элемента управления определяют форму указателя мыши, который расположен над элементом управления в зависимости от текущих обстоятельств. Например, форма указателя мыши, заданная свойством DragCursor, отображается при перетаскивании элемента управления. А форма указателя мыши, заданная свойством Cursor, отображается при размещении указателя мыши в этом элементе управления. Обратите внимание, что они работают только в тех случаях, когда для свойства Screen->Cursor задано значение crDefault.

Доступ к глобальному указателю мыши осуществляется с помощью свойства Cursor глобальной переменной Screen. При изменении этого глобального значения всегда следует создавать копию исходной формы указателя мыши, чтобы его можно было восстановить по окончании работы с новой формой указателя мыши. Для этого применяется следующая структура.

```
TCursor OriginalCursor = Screen->Cursor;
Screen->Cursor = cr XXXX ; //Указание новой формы указателя мыши
try
{
   // Какие-то действия
}
__finally
{
   // Восстановление исходной формы указателя мыши
   Screen->Cursor = OriginalCursor;
}
```

В проекте ProgressCursor.bpr, который находится на прилагаемом к этой книге компакт-диске, продемонстрированы оба метода изменения формы указателя мыши.

Пользовательские указатели мыши

Для применения пользовательского указателя мыши его сначала следует присвоить свойству-массиву Cursors глобальной переменной Screen. Для пользовательских указателей мыши можно использовать любой положительный индекс, а для встроенных указателей используются индексы от -22 (crSizeAll) до 0 (crDefault). Для получения дескриптора HCURSOR с целью присвоения указателя мыши элементу этого массива нужно использовать WinAPI-функции LoadCursor() или LoadCursorFromFile(). Для загрузки анимированных указателей мыши нужно использовать функцию LoadCursorFromFile(). Например, в проекте ProgressCursor.bpr анимированный курсор загружается из файла Face.ani с помощью следующего кода.

```
Screen->Cursors[crFaceAnimatedCursor] ==
               LoadCursorFromFile("Face.ani");
```

Здесь crFaceAnimatedCursor — константа, равная 1 в списке инициализации конструктора формы Form1. Использование констант позволяет улучшить читабельность кода. Этот индекс применяется для присвоения пользовательского указателя мыши для свойства Cursors глобальной переменной Screen, как показано ниже.

```
Screen->Cursor = crFaceAnimatedCursor;
```

Кстати, циклическое изменение вида анимированного указателя мыши Face создается благодаря перемещению курсора, которому соответствует смена кадров.

Функция LoadCursor() загружает указатель мыши из файла ресурсов. В проекте ProgressCursor.bpr он используется для загрузки указателя мыши Eye. Для этого сначала нужно создать файл ресурса (с расширением .rc) с таким содержанием.

```
EyeCursor CURSOR Eye.cur
```

Затем для присвоения дескриптора HCURSOR свойству Screen->Cursors следует использовать функцию LoadCursor().

```
Screen->Cursors[crCustomEyeCursor] ==
               LoadCursor(HInstance,"EyeCursor");
```

Здесь crCustomEyeCursor — константа, которая содержит индекс из списка инициализации, что также позволит улучшить читабельность кода в дальнейшем (при присвоении указателя мыши для свойства Cursors элемента управления CustomCursorCheckBox).

```
CustomCursorCheckBox->Cursor = crCustomEyeCursor;
```

Как видите, применять пользовательские указатели мыши очень легко. Трудности возникают при их создании, для чего крайне необходимо использовать добротный инструмент редактирования указателя мыши. Более подробная информация об использовании ресурсов приложения предлагается в разделе об использовании предварительно созданных изображений в главе 10. Хотя в этом разделе рассматриваются ресурсы пользовательских свойств и редакторов компонентов, но изложенная в нем информация в равной степени относится и к данной теме.

Строка состояния TStatusBar

Строка состояния TStatusBar (этот компонент находится во вкладке Win32 палитры компонентов Component Palette) представляет собой прекрасный механизм организации обратной связи и предоставления дополнительной информации пользователям приложения. Она также может рассматриваться как дополнительный элемент управления, который реагирует на выполняемые с мышью манипуляции. В простейшем виде строка состояния TStatusBar представляет собой панель (SimplePanel = true), в которой отображается только текстовая информация, в соответствии со значением свойства SimpleText.

Для многих приложений этого вполне достаточно, но строка состояния TStatusBar обладает и другими возможностями. Она может состоять из нескольких панелей, причем каждая из них может в случае необходимости быть "собственным" управляющим элементом (для этого задайте значение psOwnerDraw для свойства Style). Каждая панель строки состояния инкапсулирована в объекте TStatusPanel. В табл. 5.3 показаны наиболее часто используемые свойства объекта TStatusPanel.

Таблица 5.3. Наиболее часто используемые свойства объекта TStatusPanel

Свойство	Применение
Alignment	Определяет способ выравнивания текста AnsiString в свойстве Text строки состояния: taLeftJustify, taCenter или taRightJustify
Bevel	Определяет внешний вид фаски вокруг края панели в строке состояния: pbNone, pbLowered или pbRaised. Если задано значение pbNone, то панель не имеет фаски и не отличается от поля строки состояния, если только она не оформлена как-то иначе
Index	Используемое только для чтения свойство, унаследованное от TCollectionItem, которое указывает индекс (начиная с нуля) панелей внутри строки состояния. Оно используется вместе со свойствами Items и Count свойства Panels строки состояния TStatusBar. Свойство Panels имеет тип TStatusPanels, а Items — это массив объектов типа TstatusPanel. Свойство Index объекта TstatusPanel содержит индекс элемента массива Items. Для итеративного доступа к панелям (Panels->Items) строки состояния следует использовать конструкцию Panels->Count
Style	Определяет способ отображения панели: psText или psOwnerDraw. По умолчанию выбирается psText, что означает отображение строки из свойства Text с помощью шрифта строки состояния с выравниванием согласно свойству Alignment этой панели. Если для свойства Style задано значение psOwnerDraw, то необходимо предоставить необходимый код отображения этой панели в канве строки состояния в обработчике состояния панели OnDrawPanel

Окончание табл. 5.3

Свойство	Применение
Text	Определяет строку, которая будет отображена в панели, если для свойства Style стиля панели задано значение psText. Даже если для свойства Style задано значение psOwnerDraw, это значение может использоваться для хранения значения, которое будет отображено в строке состояния
Width	Определяет ширину панели. Все панели имеют одинаковую высоту, которая задается значением свойства ClientHeight строки состояния, содержащей данную панель. Обратите внимание, что объект TStatusPanel не имеет свойства Visible. Для скрытия панели строки состояния можно задать для свойства Width значение 0

Для доступа к панелям строки состояния необходимо использовать свойство Items свойства Panels объекта TStatusBar. Например, для присвоения строки This is Panel 0 свойству Text первой панели (Index == 0) в многопанельной строке состояния (например, StatusBar1), можно применить следующий код.

```
StatusBar1->Panels->Items[0]->Text = "This is Panel 0";
```

Как уже упоминалось выше (см. табл. 5.4 в описании свойства Index), свойство Panels объекта TStatusBar имеет тип TStatusPanels и является наследником объекта TCollection. По сути свойство Count может применяться для определения количества панелей в строке состояния.

В программе MiniCalculator строка состояния TStatusBar имеет три панели. Первая и третья — пользовательские (Style = psOwnerDraw), а вторая — просто отображает текст из свойства Text (более подробно он рассматривается в следующем разделе о подсказках).

Рассмотрим теперь первую пользовательскую панель. Программа MiniCalculator может отвечать на ввод данных с клавиатуры. Благодаря этому пользователь может работать с этим приложением так же, как с обычным калькулятором, где клавиши клавиатуры компьютера (в основном это цифровая клавиатура) играют роль клавиш калькулятора. Это допускается по умолчанию, но в случае необходимости эту возможность можно запретить. Кроме того, иногда приложение не может реагировать на ввод с помощью клавиатуры, например, когда основная форма приложения теряет фокус ввода. В программе MiniCalculator первая панель (0) отвечает за показ пользователю режима ввода с клавиатуры, а также позволяет ему включить или отключить режим ввода с клавиатуры. Для указания статуса клавиатуры используются три режима: для обозначения режима ввода с клавиатуры и наличия фокуса в основной форме; для обозначения режима ввода с клавиатуры и отсутствия фокуса в основной форме; для обозначения отключения режима ввода с клавиатуры. Эти изображения хранятся в трех компонентах TImage: HasKeyboardFocusImage, HasNoKeyboardFocusImage и DisableKeyboardImage. Для отображения изображений в строке состояния требуется реализовать событие OnDrawPanel объекта TStatusBar. Событие OnDrawPanel имеет следующие три параметра.

```
TStatusBar* StatusBar,
TStatusPanel* Panel,
const TRect&Rect
```

StatusBar — указатель на строку состояния, которая запускает данное событие; Panel — указатель на панель данного события; TRect — структура, которая обозначает прямоугольную область рисования. При рисовании панели параметр Rect задает границы используемого изображения, а свойство Canvas параметра StatusBar применяется непосредственно для рисования.

В листинге 5.1 показано, как это делается в программе MiniCalculator. Учтите, что для простоты здесь опущена часть кода, которая относится к рисованию другой пользовательской панели, где `Panel->Index == 2`. Код рисования этой панели приведен в листинге 5.4.

Листинг 5.1. Реализация события `OnDrawPanel` объекта `TStatusBar` — Часть 1

```
void __fastcall TMainForm::StatusBar1DrawPanel(
                                    TStatusBar* StatusBar,
                                    TStatusPanel* Panel,
                                    const TRect& Rect)
{
   if(Panel->Index ==0)
   {
      if(EnableKeyboardInput)
      {
         if(CanUseKeyboard)
         {
            // Рисование в панели изображения клавиатуры
            StatusBar->Canvas->Draw(
                       Rect.Left,
                       Rect.Top,
                       HasKeyboardFocusImage->Picture->Bitmap);
         }
         else
         {
            StatusBar->Canvas->Draw(                      Rect.Left,
                      Rect.Top,
                      HasNoKeyboardFocusImage->Picture->Bitmap);
         }
      }
      else
      {
         StatusBar->Canvas->Draw(
                    Rect.Left,
                    Rect.Top,
                    DisableKeyboardImage->Picture->Bitmap);
      }
   }
   // Фрагмент кода if(Panel->Index == 2) опущен здесь,
   // см. листинг 5.2
}
```

Код в листинге 5.1 достаточно прост и понятен. Сначала нужно убедиться в том, что данная панель является первой. Для этого проверяется равенство нулю значения свойства `Index`. Для принятия решения о том, какой рисунок будет отображен в панели, проверяются значения следующих двух переменных.

- `EnableKeyboardInput`. Если значение этой переменной равно `true`, то включается режим ввода с помощью клавиатуры; а если оно равно `false` — отключается.

- `CanUseKeyboard`. Если значение этой переменной равно `true`, то основная форма содержит фокус ввода; а если оно равно `false` — то фокус ввода находится за пределами основной формы.

Эти переменные являются свойствами логического типа, что существенно упрощает кодирование строки состояния и всего приложения в целом. Мы еще вернемся к этому вопросу ниже в этой главе.

Независимо от того, какое изображение используется, применяется одинаковый метод рисования `Draw()` свойства `Canvas` строки состояния `StatusBar`. В качестве параметров в этом случае принимаются координаты верхнего левого угла области рисования изображения, а также указатель на объект-наследник объекта `TGraphic`, в данном случае это объект `TBitmap`. В программе MiniCalculator свойство `Height` имеет значение `30`, а свойство `Width` первой панели — значение `72`. Ширина панели включает однопиксельную фаску, даже если для свойства `Bevel` задано значение `pbNone`, которое просто обозначает отсутствие фаски. Помимо однопиксельной фаски, которая окаймляет панель, в верхней части строки состояния располагается двухпиксельная полоса. Следовательно, для вычисления размеров области рисования (параметр `Rect`) нужно выполнить следующие вычисления.

```
UseablePanelHeight = StatusBar->Height - 4 - 2 *
                     StatusBar->BorderWidth;
UseablePanelWidth = Panel->Width - 2;
```

В результате этих вычислений получим, что для рисования первой панели имеется область с размерами 70 на 26 пикселей. Таков размер изображения, который будет размещен в этой панели, т.е. высота строки состояния и ширина панели были сознательно выбраны именно такими, чтобы это изображение поместилось в ней. Полученная в результате этих действий панель показана на рис. 5.2.

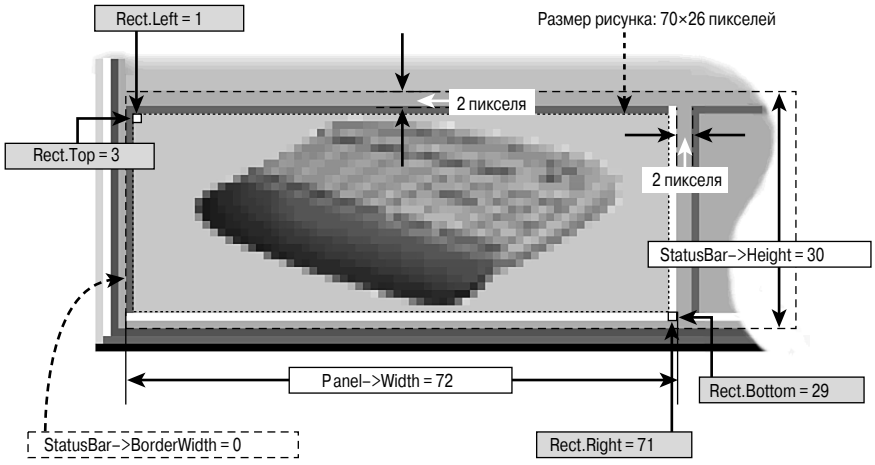


Рис. 5.2. Пользовательская панель строки состояния

Строка состояния в программе MiniCalculator также откликается на действия, которые пользователь выполняет с помощью мыши. При нажатии левой кнопки мыши в первой панели строки состояния переключается значение свойства `EnableKeyboardInput`, что позволяет включать/отключать режим ввода с помощью клавиатуры. Для этого нужно реализовать событие `OnMouseDown` строки состояния. Обработчик события `OnMouseDown` проверяет, находится ли мышь над первой панелью строки состояния. Если при этом будет нажата левая кнопка мыши, то будет переключено значение свойства `EnableKeyboardInput`. В листинге 5.2 представлен код этого обработчика.

Листинг 5.2. Реализация обработчика события `OnMouseDown` строки состояния `TStatusBar`

```
void __fastcall TMainForm::StatusBar1MouseDown(
                                    TObject* Sender,
                                    TMouseButton Button,
                                    TShiftState Shift,
                                    int X,
                                    int Y)
{
   if(Button == mbLeft)
   {
   // Находится ли мышь в первой панели
   // или внутри фаски этой панели ?
      if(   X > 1
         && X < (StatusBar1->Panel->Items[0]->Width - 1)
         && Y > 3
         && Y < (StatusBar1->Height - 1))
      {
         // Если да, переключить режим ввода с клавиатуры
         if(EnableKeyboardInput) EnableKeyboardInput = false;
         else EnableKeyboardInput = true;
      }
   }
}
```

Этот код легко понять, прибегнув к помощи рис. 5.2. При изменении значения свойства `EnableKeyboardInput` вызывается метод задания значения `SetEnableKeyboardInput`, который показан в листинге 5.3.

Листинг 5.3. Метод `SetEnableKeyboardInput`

```
void __fastcall TMainForm::SetEnableKeyboardInput(bool NewEnableKeyboardInput)
{
   if(EnableKeyboardInput != NewEnableKeyboardInput)
   {
      FEnableKeyboardInput = NewEnableKeyboardInput;
      if(EnableKeyboardInput)
      {
         OnKeyDown = CalculatorKeyDown;
         OnKeyUp = CalculatorKeyUp;
         StatusBar1->Panels->Items[1]->Width = 60;
      }
      else
      {
         OnKeyDown = 0;
         OnKeyUp = 0;
         StatusBar1->Panels->Items[1]->Width = 0;
      }
      StatusBar1->Invalidate();
   }
}
```

Отметим в листинге 5.3 следующий факт: при изменении значения FEnableKeyboardInput вызывается метод Invalidate() строки состояния для указания необходимости перерисовки и, следовательно, запуска события OnDrawPanel, которое позволяет обновить изображение в панели с индексом 0. Кроме того, можно заметить, что задаются события OnKeyDown и OnKeyUp основной формы. При указании для них значений 0 программа MiniCalculator не будет отвечать на ввод с клавиатуры, а при указании методов CalculatorKeyDown() и CalculatorKeyUp()программа будет воспринимать ввод с клавиатуры.

Внешний вид панели можно оформить в виде обычной кнопки. Для этого нужно для свойства Bevel задать значение bpRaised. При возникновении события OnMouseDown свойство Bevel получит значение bpLowered, а при возникновении события OnMouseUp свойство Bevel получит значение bpRaised. Кроме того, следует выполнить проверку события OnMouseUp внутри панели. Если оно произошло, то следует проверить, не произошел ли щелчок мышью на этой панели, и отреагировать на это событие соответственно.

Рассмотрим третью панель строки состояния программы MiniCalculator, которая также является пользовательской. Эта панель отображает описание соответствующей кнопки. Весьма вероятно, что это описание будет занимать больше места, чем предусмотрено в этой панели. В таком случае рекомендуется обрезать длинное описание и добавить многоточие (...) в конце строки. Это указывает пользователю на то, что остальная часть строки скрыта. В таком виде панель выглядит более аккуратно, чем в том случае, когда длинный текст просто обрезан краем панели. Для отображения строки таким образом следует использовать WinAPI-функцию DrawText(). Она будет часто использоваться во всей этой главе, поэтому следует более подробно описать ее возможности.

Объявление функции DrawText() выглядит так, как показано ниже.

```
int DrawText(

      HDC     hDC,      // Передача параметра Canvas->Handle
      LPCTSTR lpString, // Передача параметра string.c_str(),
                        // где string имеет тип AnsiString
      int nCount,       // длина строки или -1 для пустой строки
      LPRECT lpRect,    // Указатель на структуру TRect
      UINT uFormat      // Флаги форматирования и рисования
);
```

В случае успешного выполнения функция DrawText() возвращает высоту строки. Именно флаги форматирования текста этой функции делают ее такой универсальной. Они перечислены в табл. 5.4. Учтите, что в табл. 5.4 приведены не в алфавитном порядке, а в логически связанных группах.

Таблица 5.4. Флаги форматирования WinAPI-функции DrawText()

Флаг	Использование
DT_CALCRECT	Текст не отображается; область Trect, указанная параметром lpRect, изменяется для задания границ форматированного текста
DT_LEFT	Выравнивает текст по левому краю области TRect, указанной параметром lpRect
DT_CENTER	Выравнивает текст по центру области TRect, указанной параметром lpRect
DT_RIGHT	Выравнивает текст по правому краю области TRect, указанной параметром lpRect

Окончание табл. 5.4

Флаг	Использование
DT_BOTTOM	Выравнивает текст по нижнему краю области TRect, указанной параметром lpRect. Должен использоваться вместе с флагом DT_SINGLELINE
DT_VCENTER	Выравнивает текст вертикально по центру области TRect, указанной параметром lpRect. Должен использоваться вместе с флагом DT_SINGLELINE
DT_TOP	Выравнивает текст по верхнему краю области TRect, указанной параметром lpRect. Должен использоваться вместе с флагом DT_SINGLELINE
DT_SINGLELINE	Отображает текст в одной строке; при этом сдвиг и перенос строки не могут разорвать ее
DT_END_ELLIPSIS	Если текст не помещается в области Trect, указанной параметром lpRect, то текст обрезается и к нему добавляется многоточие (..). Если задан флаг DT_MODIFYSTRING, то строка может быть изменена
DT_PATH_ELLIPSIS	Аналогичен флагу DT_END_ELLIPSIS, но предназначен для отображения пути к файлу с обратной косой чертой (\). Текст перед этим символом или между ними заменяется многоточием (...), если этот текст имеет очень большую длину. Если задан флаг DT_MODIFYSTRING, то строка также изменяется
DT_MODIFYSTRING	Если задан этот флаг, а также флаги DT_END_ELLIPSIS или DT_PATH_ELLIPSIS, текст будет изменяться при каждом отображении панели. В противном случае он не оказывает никакого влияния
DT_EDITCONTROL	Текст отображается так, как если бы он содержался в многострочном текстовом поле. При этом частично видимые последние строки не отображаются, а ширина символов определяется как среднее значение
DT_EXPANDTABS	Вкладки расширяются. По умолчанию размер такой вкладки равен 8 символам
DT_EXTERNALLEADING	Отображается внешняя высота шрифта (пространство между строками)
DT_NOCLIP	Текст отображается без обрезания. При указании этого флага отображение происходит быстрее
DT_NOPREFIX	Если указан этот флаг, то символы префикса не обрабатываются. Например, амперсанд отображается просто как символ &. Обычно он используется как клавиша ускорения, которая отмечается символом подчеркивания, как в командах меню
DT_RTLREADING	Создает текст справа налево при поддержке двустороннего рисования
DT_TABSTOP	Задает размер вкладки, который по умолчанию он равен 8 символам. Новое значение располагается в битах 8–15 параметра uFormat
DT_WORDBREAK	Если текст очень большой и не помещается в области TRect, указанной параметром lpRect, то строка разбивается и переносится на следующую строку. Операции сдвига или переноса строки также могут разрывать строку

В листинге 5.4 представлен код, необходимый для создания текста в канве строки состояния. Это точно такой же обработчик события, который уже был представлен выше в листинге 5.1. Однако теперь здесь опущен код рисования панели с индексом `Panel->Index == 0`, а приведен код рисования панели с индексом `Panel->Index == 2`.

Листинг 5.4. Реализация события `OnDrawPanel` строки состояния — Часть 2

```
void __fastcall TMainForm::StatusBar1DrawPanel(
                                TStatusBar* StatusBar,
                                TStatusPanel* Panel,
                                const TRect& Rect)
{
   if(Panel->Index == 0)
   {
      // Для простоты эта часть опущена - см. листинг 5.1
   }
   else if(Panel->Index == 2)
   {
      TFontStyles FontStyle;
      TColor OldBrushColor = StatusBar->Canvas->Brush->Color;
      TFontStyles OldFontStyle = StatusBar->Canvas->Font->Style;
      StatusBar->Canvas->Font->Style = FontStyle;
      StatusBar->Canvas->Brush->Color = clSilver;
      StatusBar->Canvas->FillRect(Rect);
      TRect PanelRect = Rect;
      PanelRect.Left += 2;
      PanelRect.Right -= 2,
      DrawText( StatusBar->Canvas->Handle,
                Panel->Text.c_str(),
                -1,
                &PanelRect,
                DT_LEFT
                |DT_NOPREFIX
                |DT_END_ELLIPSIS
                |DT_SINGLELINE
                |DT_VCENTER
                |DrawTextBiDiModeFlagsReadingOnly() );

      StatusBar->Canvas->Font->Style = OldFontStyle;
      StatusBar->Canvas->Brush->Color = OldBrushColor;
   }
}
```

В листинге 5.4 показано, что создание описания кнопки в третьей панели (`Panel>Index == 2`) строки состояния включает четыре этапа.

1. Во-первых, сохранение текущих свойств канвы `StatusBar->Canvas`, которые будут изменены, и присвоение новых значений.
2. Изменение области рисования `Rect` для создания 2-пиксельной границы с двух сторон панели.
3. Создание текста `Panel->Text` в канве `StatusBar->Canvas` с помощью функции `DrawRect()`. Текст выравнивается по левому краю (`DT_LEFT`) и центруется по вертикали (`DT_VCENTER`) внутри области рисования панели `PanelRect`. Кроме того, текст

Panel->Text должен располагаться в одной строке (DT_SINGLELINE), а если строка очень велика, то в ее конце должно отображаться многоточие (DT_END_ELLIPSIS). Также отключается обработка символов префикса (DT_NOPREFIX), а функция DrawTextBiDiModeFlagsReadingOnly() объекта TControl используется для выяснения, следует ли установить флаг DT_RTLREADING.

4. Наконец, свойствам канвы StatusBar->Canvas возвращаются их исходные значения.

Функция DrawText() представляет собой очень мощный инструмент создания текста в канве, а флаги форматирования упрощают настройку внешнего вида такого текста. Мы еще не раз будем использовать эту функцию в этой главе, поэтому стоит познакомиться с ней поближе.

Подсказки

Использование подсказок позволяет улучшить интерфейс, а также предоставить дополнительную информацию или приглашения для пользователя. Свойство ShowHint элемента управления определяет, будет ли отображена на экране строка свойства Hint при размещении указателя мыши над элементом управления. По желанию ответственность за это можно переложить на объект-родитель элемента управления, задав для свойства ParentShowHint значение true. Подсказка элемента управления будет отображена, только если объект-родитель позволяет отображать подсказки. Иногда это упрощает кодирование и позволяет глобально включить и отключить подсказки. Использование всплывающей подсказки при размещении указателя мыши над данным элементом управления, вероятно, является наиболее популярным способом отображения подсказок.

Подсказка может быть короткой, длинной, или одновременно короткой и длинной. Для создания подсказки, содержащей короткую и длинную части, используйте вертикальный символ разделения (|) в тексте подсказки для отделения короткой части (она располагается первой) от длинной. Например, такая подсказка может иметь представленный ниже вид.

```
AnsiString MyHint = "Short Hint|This is a Long Hint";
```

Обратите внимание, что термины *короткая часть* и *длинная часть* обозначают расположение подсказки внутри всей строки подсказки и не относятся к их длине. Они произошли от способа их использования. Короткая часть подсказки является первой частью подсказки и обычно используется как всплывающая подсказка. Вторая, длинная часть подсказки обычно более описательная и отображается в строке состояния. Вследствие такого способа использования первая часть действительно должна быть короткой, а вторая — длинной.

Во всплывающей подсказке отображается только короткая часть подсказки. Длинная часть подсказки передается событию OnHint объекта TApplication. В этом месте длинную часть подсказки можно для отображения на экране передать другому элементу управления, например, в строке состояния. Для использования только короткой части подсказки добавьте символ | в конце строки; в противном случае короткая часть подсказки будет использована так же, как и длинная. Для отображения на экране только длинной части подсказки следует начать строку подсказки с символа |. Для извлечения короткой или длинной части из строки подсказки используйте глобальные функции GetShortHint() и GetLongHint(), соответственно, передавая подсказку в качестве аргумента.

Объект TApplication имеет несколько свойств, которые позволяют настроить внешний вид подсказки в приложении, длительность паузы перед появлением подсказки и продолжительность ее отображения на экране. Наиболее интересные свойства подсказки показаны в табл. 5.5.

Таблица 5.5. Свойства объекта TApplication, предназначенные для управления подсказками

Свойство	Применение
Hint	Содержит длинную часть подсказки для элемента управления, над которым расположен указатель мыши
HintColor	Указывает цвет поля подсказки. По умолчанию используется системный цвет clInfoBk (обычно это бледно-желтый цвет), но для него можно использовать любой цвет
HintHidePause	Указывает длительность отображения всплывающей подсказки при условии, что указатель мыши располагается над этим элементом управления. По умолчанию этот период равен 2500 миллисекундам. Для него можно использовать любое другое значение в миллисекундах
HintPause	Указывает длительность паузы перед показом подсказки после расположения указателя мыши над данным элементом управления. По умолчанию она равна 500 миллисекундам. Для нее можно использовать любое другое значение в миллисекундах
HintShortCuts	Указывает, будет ли показана информация о клавише ускорения в производном от TCustomAction объекте в скобках после строки, указанной в свойстве Hint этого действия. Например, если действие TCopyAction имеет подсказку "Copy" и комбинацию клавиш ускорения <Ctrl+C>, то подсказка этого действия будет отображаться как "Copy (Ctrl+C"
HintShortPause	Указывает время задержки перед активацией другой подсказки после отображения первой подсказки. По умолчанию оно равно 500 миллисекундам. Для него можно использовать любое другое значение в миллисекундах
ShowHint	Указывает, будут ли использоваться подсказки для всего приложения. Если задано значение false, то подсказки отображаться не будут. По умолчанию используется значение true

Помимо свойств, перечисленных в табл. 5.5, объект TApplication также содержит метод ActivateHint() для приведения в действие отображаемой подсказки. Он объявлен в файле $(BCB)\Include\Vcl\Forms.hpp так, как показано ниже.

```
void __fastcall ActivateHint(const Windows::TPoint&CursorPos);
```

Для использования этой функции нужно просто передать в качестве параметра текущую позицию указателя мыши с помощью свойства CursorPos объекта Mouse. Объект TApplication отображает подсказку для элемента управления, над которым находится указатель мыши. Например, так, как показано ниже.

```
TApplication->ActivateHint(Mouse->CursorPos);
```

Объект TScreen имеет свойство HintFont, которое позволяет настраивать шрифт, используемый для всплывающей подсказки.

Ручное управление подсказками

В программе MiniCalculator для каждой панели строки состояния создана отдельная подсказка. Такая организация работы подсказок связана с двумя проблемами. Панели TStatusPanel не имеют собственного свойства Hint, потому что они представляют только некоторую область канвы строки состояния TStatusBar. И только строка состояния

TStatusBar имеет свою подсказку. Для отображения подсказок для разных панелей, необходимо определить расположение указателя мыши внутри строки состояния TStatusBar и соответственным образом задать значение свойства Hint объекта TStatusBar. Кроме того, при отображении подсказки для данного элемента управления, она не показывается снова до тех пор, пока указатель мыши не покинет территорию этого элемента управления и опять не вернется в нее. Это значит, что если указатель мышь располагается в строке состояния над первой панелью и отображается некая подсказка, то при перемещении указателя мыши над второй панелью новая подсказка не будет отображена. Для этого нужно вывести указатель мыши за пределы строки состояния, а затем вернуть его в нужную панель строки состояния. Если нужно показать подсказку для каждой панели, не покидая сроку состояния, то необходимо организовать принудительную перерисовку подсказки строки состояния TStatusBar.

В программе MiniCalculator это делается с помощью события OnMouseMove строки состояния TStatusBar. В листинге 5.5 показан возможный способ реализации этого способа.

Листинг 5.5. Реализация события OnMouseMove при создании отдельных подсказок для каждой панели строки состояния

```
void __fastcall TMainForm::StatusBar1MouseMove(TObject* Sender,
                                               TShiftState Shift,
                                               int X,
                                               int Y)
{
   int BorderWidth = StatusBar1->BorderWidth;

   TRect Panel0(StatusBar1->ClientOrigin.x        //Слева
                + BorderWidth + 1,

                StatusBar1->ClientOrigin.y        //Сверху
                + 3 + BorderWidth,

                StatusBar1->ClientOrigin.x        //Справа
                + BorderWidth
                + StatusBar1->Panels->Items[0]->Width - 1,

                StatusBar1->ClientOrigin.y + 3    //Снизу
                + StatusBar1->Height - 1);

   TRect Panel1(Panel0.Right + 2 + 1,             //Слева
                Panel0.Top,                       //Сверху то же
                Panel0.Right + 2                  //Справа
                + StatusBar1->Panels->Items[1]->Width - 1,
                Panel0.Bottom);                   //Снизу то же

   TRect Panel2(Panel1.Right + 2 + 1,             //Слева
                Panel0.Top,                       //Сверху то же
                Panel1.Right + 2                  //Справа
                + StatusBar1->Panels->Items[2]->Width - 1,
                Panel0.Bottom);                   //Снизу то же

// Наблюдать, где находится указатель мыши,
// и показать нужную подсказку :-)
```

```
// Использовать WinAPI-функцию для проверки наличия
// указателя мыши в любой области панели Rect.
// Если указатель мыши находится в области панели,
// отобразить соответствующую подсказку.
// В противном случае присвоить StatusBar1->Hint = ""

// BOOL PtInRect(
//    CONST RECT *lprc, // адрес структуры прямоугольника
//    POINT pt          // структура точки
// );

if(PtInRect(&Panel0, Mouse->CursorPos))
{
   if(EnableKeyboardInput)
   {
      StatusBar1->Hint = "Click to Disable Keyboard Intput|";
   }
   else
   {
      StatusBar1->Hint = "Click to Enable Keyboard Input|";
   }
   if(StatusBar1->Tag != 0)
   {
      StatusBar1->Tag = 0;
      Application->ActivateHint();
   }
}
else if(PtInRect(&Panel1, Mouse->CursorPos))
{
   StatusBar1->Hint = "Keyboard Short Cut|";
   if(StatusBar1->Tag != 1)
   {
      StatusBar1->Tag = 1;
      Application->ActivateHint();
   }
}
else if(PtInRect(&Panel2, Mouse->CursorPos))
{
   StatusBar1->Hint = "Button Function|";
   if(StatusBar1->Tag != 2)
   {
      StatusBar1->Tag = 2;
      Application->ActivateHint();
   }
   else
   {
      //No Hint
      StatusBar1->Tag = StatusBar1->Panels->Count;
      StatusBar1->Hint = "|";
   }
}
```

Следует отметить в листинге 5.5 создание структуры TRect для представления области панелей строки состояния. Каждая такая область содержит только внутреннюю часть панели (без фаски) и не включает 2-пиксельное пространство между панелями. Для того чтобы вспомнить структуру внешнего вида строки состояния, полезно еще раз взглянуть на рис. 5.2.

Затем используется WinAPI-функция PtInRect() для проверки наличия указателя мыши в панели. Функция PtInRect() имеет следующее объявление.

```
BOOL PtInRect(
   CONST RECT *lprc, // Адрес структуры TRect
   POINT pt          // Структура TPoint
);
```

Для этого достаточно передать функции адрес структуры TRect, в которой следует проверить наличие указателя мыши, а также передать структуре TPoint информацию о положении мыши. Если указатель мыши находится в рамках структуры TRect, представляющей панель и если указатель мыши попал в эту панель впервые, то свойство Hint объекта StatusBar1 получает строку и вызывается метод ActivateHint() объекта TApplication для отображения этой подсказки.

Вызов метода ActivateHint() приводит к немедленному отображению подсказки. Это означает отсутствие паузы, которая предваряет появление обычной подсказки. В некоторых ситуациях пауза не нужна, но в этом примере она потребуется. Для этого нужно организовать вспомогательные функции, которые позволили бы выполнить эту задачу.

Во-первых, нужно обратить внимание на свойство Tag объекта TStatusBar, которое указывает, отображена ли уже подсказка для данной панели. При возникновении события OnMouseMove в области панели подсказка отображается только в том случае, если это первое событие OnMouseMove для этой панели. В таком случае гарантируется корректное поведение приложения, а именно, подсказка отображается только один раз для элемента управления в течение всего промежутка времени, когда указатель мыши находится над этим элементом управления. Для повторного отображения подсказки указатель мыши должен выйти за пределы элемента управления и вернуться в него вновь. В рассматриваемом примере указатель мыши должен выйти за пределы панели и вернуться в нее вновь. Это приведет к тому, что свойству Tag объекта StatusBar1 будет присвоено другое значение. Индекс каждой панели присваивается свойству Tag объекта StatusBar1 при первом вызове обработчика события OnMouseMove. Если указатель мыши покидает строку состояния и возвращается в нее вновь, значение свойства Tag несущественно, потому что в этом случае не нужно принудительно отображать подсказку. При входе указателя на территорию элемента управления, содержащего подсказку, будет показана текущая подсказка. Следовательно, важно убедиться в том, что свойство Hint строки состояния TStatusBar имеет значение "" или "|". Оба эти значения символизируют собой пустые строки подсказок. Обратите внимание также на то, что если событие OnMouseMove произойдет в строке состояния, а не в какой-либо из панелей, значение свойства Tag устанавливается равным количеству панелей в строке состояния. В этом случае нет панели с таким индексом, и это позволяет повторно отобразить подсказку, если указатель мыши покинет панель, а не строку состояния, а затем не вернется в какую-то из панелей. Если ширина строки состояния (т.е. значение свойства BorderWidth) велика, то уход и возврат в панель будет не очень заметным.

Сложнее выглядит код отображения подсказки после некоторой паузы, как это происходит с обычными подсказками. Для этого следует использовать WinAPI-функцию SetTimer(), чтобы использовать функцию обратного вызова по окончании заданного промежутка времени. И только после этого можно будет отобразить подсказку. Для организации такого поведения потребуется использовать несколько функций. В листинге 5.6 показаны дополнительные объявления, которые необходимо сделать в таком случае в заголовочном файле формы.

Листинг 5.6. Дополнительные объявления, необходимые для организации ручного управления подсказками

```
UINT HintTimerHandle;
static void CALLBACK HintTimerCallback(HWND Wnd,
                                       UINT Msg,
                                       UINT TimerID,
                                       DWORD Time);
void __fastcall      HintTimerExpired();
void __fastcall      DisplayHint(int Pause);
void __fastcall      StopHintTimer();
```

Функция обратного вызова `HintTimerCallback()` объявлена как статическая в объявлении класса формы. Дело в том, что указатель на эту функцию будет передан WinAPI-методу `SetTimer()`, а он ожидает указатель обычной функции. Переменная `HintTimerHandle` предназначена для хранения дескриптора, возвращенного функцией `SetTimer()` для остановки таймера, когда он уже не нужен. Реализация этих функций показана в листинге 5.7.

Листинг 5.7. Реализация функций для ручного управления задержкой отображения подсказок

```
//---------------------------------------------------------//
void CALLBACK TMainForm::HintTimerCallback(HWND Wnd,
                                           UINT Msg,
                                           UINT TimerID,
                                           DWORD Time)
{
   // Для безопасного преобразования типов
   TObject* VCLObject = reinterpret_cast<TObject*>(Wnd);
   TForm1* Form1Object = dynamic_cast<TForm1*>(VCLObject);
   if(Form1Object) Form1Object->HintTimerExpired();
}
//---------------------------------------------------------//
void __fastcall TMainForm::HintTimerExpired()
{
   StopHintTimer();
   Application->ActivateHint(Mouse->CursorPos);
}
//---------------------------------------------------------//
void __fastcall TMainForm::DisplayHint(int Pause)
{
   StopHintTimer();
   HintTimerHandle =
       SetTimer(this,
                0,
                Pause,
                reinterpret_cast<TIMERPROC>(HintTimerCallback));
   if(HintTimerHandle == 0) Application->CancelHint();
}
//---------------------------------------------------------//
void __fastcall TMainForm::StopHintTimer()
{
   if(HintTimerHandle != 0)
   {
```

```
      KillTimer(this, HintTimerHandle);
      HintTimerHandle = 0;
   }
}
//---------------------------------------------------------//
```

Именно с функции `DisplayHint()` все и начинается. Она вызывается обработчиком события `OnMouseDown` вместо `Application->ActivateHint()` с `Application->HintPause` в качестве единственного аргумента. Все строки в листинге 5.5, которые содержат `Application->ActivateHint();`, нужно заменить строками `DisplayHint(Application->HintPause);`.

Аргумент `Application->HintPause` предназначен для извлечения текущей паузы отображения подсказки, которая используется в данном приложении. Это гарантирует совместимость при работе с другими подсказками. Хотя в некоторых случаях он может потребоваться для передачи другого значения.

При вызове функции `DisplayHint()` сначала останавливается таймер, используемый в текущий момент. Затем устанавливаются: новый таймер, передающий в качестве аргумента указатель `this`, благодаря чему функция `HintTimerExpired()` может быть вызвана из статической функции обратного вызова; задержка в миллисекундах (`Pause`); а также указатель на функцию обратного вызова для получения сообщения о том, что таймер закончил свою работу. Обратите внимание на то, что для функции обратного вызова нужно явным образом выполнить преобразование к типу `TIMERPROC`. Для этого используется ключевое слово `reinterpret_cast`, поскольку в данном случае мы имеем дело с функциональными указателями. При неудачной установке таймера (т.е. функция `SetTimer()` возвращает значение `0`) с помощью вызова функции `Application->CancelHint()` отменяется текущая подсказка. Без этого новую подсказку отобразить не удастся.

После окончания работы таймера вызывается функция обратного вызова. Она имеет четыре параметра, но наибольший интерес для нас представляет только первый параметр — `HWND Wnd`. Именно в нем хранится указатель `this` для вызова функции `SetTimer()`. Далее для отображения подсказки нужно вызвать функцию `HintTimerExpired()`. Так как параметр `Wnd` не является указателем, его тип сначала нужно преобразовать к типу `TObject*`, используя ключевое слово `reinterpret_cast`. Затем нужно выполнить динамическое преобразование (с помощью ключевого слова `dynamic_cast`) этого типа `TObject*` к типу `TForm1*`. Это дает нам возможность проверить истинность указателя перед его разыменованием. Разыменование неверного указателя приведет к нарушению доступа, чего хотелось бы избежать.

Наконец, функция `HintTimerExpired()` прекращает выполнение текущего таймера подсказки за счет вызова функции `StopHintTimer()`, а затем отображает подсказку с помощью функции `Application->ActivateHint()`. Функция `StopHintTimer()` просто проверяет наличие таймера, который нужно остановить (проверяя переменную `FHintTimerHandle`). Если такой таймер имеется, его работа прекращается с помощью WinAPI-фунцкии `KillTimer()`. А затем для переменной `FHintTimerHandle` задается значение `0`.

При такой организации подсказок переход указателя мыши от одной панели к другой в строке состояния будет сопровождаться последовательным отображением их подсказок, как это бывает с обычными подсказками.

Настраиваемые подсказки

Иногда для пользователя требуется организовать отображение настраиваемых подсказок. В программе MiniCalculator всплывающая подсказка отображается, когда указатель мыши располагается над кнопкой Memory Recall. При этом в подсказке отображается содержимое памяти калькулятора. Попробуем оформить подсказку в виде основного дисплея калькулятора, который показан на рис. 5.1.

Для создания настраиваемой подсказки необходимо создать новый класс окна подсказки, производный от класса THintWindow. Затем необходимо с помощью оператора __classid присвоить тип TMetaClass* глобальной переменной HintWindowClass.

Сначала рассмотрим процесс создания класса, производного от класса THintWindow. В табл. 5.6 представлены виртуальные методы класса THintWindow и их краткое описание. Для удобства при описании назначения функции указан тип возвращаемого функцией значения и список параметров.

Таблица 5.6. Виртуальные методы класса THintWindow

Метод	Применение
CreateParams()	void CreateParams(TCreateParams& Params) Этот метод следует переопределить для управления типом окна, создаваемого для представления подсказки
Paint()	void Paint(void) Этот метод следует переопределить для управления способом перерисовки подсказки на экране. Используйте свойство ClientRect окна подсказки для получения границ перерисовываемой области, а свойство Canvas окна подсказки для самой перерисовки
CalcHintRect()	Windows::TRect CalcHintRect(int MaxWidth, const AnsiString Ahint, void*AData) Этот метод следует переопределить для установки нужного размера окна подсказки. Для выполнения нужных вычислений следует использовать три параметра. В результате этот метод возвратит структуру TRect, которая содержит размеры клиентской области окна подсказки
ActivateHint()	void ActivateHint(const Windows::TRect& Rect, const AnsiString AHint) Этот метод следует переопределить для указания места отображения подсказки. В классе ThintWindow функция ActivateHint() отображает окно подсказки с координатами, указанными в параметре Rect, если оно не будет скрыто. В противном случае, структура Rect модифицируется таким образом, что используется ближайшее положение на экране. Метод ActivateHint() также задает значение свойства Caption класса THintWindow. Свойство Caption используется в методе CalcHintRect() для определения размеров окна подсказки, необходимых для размещения подсказки, а также используется методом Paint() для создания текста подсказки в окне подсказки
ActivateHintDat a()	void ActivateHintData(const Windows::TRect&Rect, const AnsiString AHint, void*AData) Этот метод следует переопределить для использования дополнительного параметра void* AData. В противном случае этот метод аналогичен методу ActivateHint(). В исходной реализации этого метода в классе THintWindow параметр Adata игнорируется

Окончание табл. 5.6

Метод	Применение
`IsHintMsg()`	`bool IsHintMsg(tagMSG& Msg)` Этот метод следует переопределить для указания сообщений, которые требуют скрытия окна подсказки. В исходной реализации этого метода в классе `THintWindow` это необходимо для всех сообщений мыши, клавиатуры, команд и сообщений активизации. Глобальный объект `Application` вызывает метод `IsHintMsg()` для проверки сообщений во время отображения окна подсказки. Если возвращается значение `true`, окно подсказки скрывается

Переопределяя виртуальные методы класса `THintWindow`, можно нужным образом настроить внешний вид класса подсказки. В листинге 5.8 показано определение класса подсказки `TCalculatorHintWindow`, производного от класса `THintWindow`, который используется в программе MiniCalculator.

Листинг 5.8. Определение класса подсказки `TCalculatorHintWindow`

```
class TCalculatorHintWindow :public THintWindow
{
   typedef THintWindow inherited;
protected:
   virtual void __fastcall Paint(void);
   virtual void __fastcall CreateParams(TCreateParams &Params);
public:
   __fastcall virtual TCalculatorHintWindow(
                               Classes::TComponent* AOwner);

   virtual void __fastcall ActivateHint(
                                  const Windows::TRect&Rect,
                                  const AnsiString AHint);

   virtual void __fastcall ActivateHintData(
                                  const Windows::TRect& Rect,
                                  const AnsiString AHint,
                                  void*AData);

   virtual Windows::TRect __fastcall CalcHintRect(
                                        int MaxWidth,
                                        const AnsiString AHint,
                                        void*AData);

   virtual bool __fastcall IsHintMsg(tagMSG& Msg);

   __property BiDiMode ;
   __property Caption ;
   __property Color ;
   __property Canvas ;
   __property Font ;
public:
```

```
   inline __fastcall virtual ~TCalculatorHintWindow(void)
   { }

public:
   inline __fastcall TCalculatorHintWindow(HWND ParentWindow)
                     :THintWindow(ParentWindow)
   { }
};
```

Из представленных в этом объявлении методов только методы `CreateParams()`, `Paint()` и `CalcHintRect()` переопределены и отличаются от одноименных методов родительского класса. Остальные просто вызывают унаследованные методы родительского класса `THintWindow`. В листинге 5.9 приведена реализация класса `TCalculatorHintWindow`.

Листинг 5.9. Реализация класса `TCalculatorHintWindow`

```
//-------------------------------------------------------------//
//                      CONSTRUCTOR                            //
//-------------------------------------------------------------//
__fastcall
TCalculatorHintWindow::TCalculatorHintWindow(
                                Classes::TComponent* AOwner)
                               : THintWindow(AOwner)
{
   Canvas->Font->Name = "Arial";
   Canvas->Font->Color = clBlack;
}
//-------------------------------------------------------------//
//                        CreateParams                         //
//-------------------------------------------------------------//
void __fastcall TCalculatorHintWindow::CreateParams(
                                         TCreateParams &Params)
{
   inherited::CreateParams(Params);

   Params.Style = WS_POPUP;
   Params.WindowClass.style = Params.WindowClass.style |
                              CS_SAVEBITS;

   if(NewStyleControls) // Проверка глобальной
                        // переменной NewStyleControls
   {
      Params.ExStyle = WS_EX_TOOLWINDOW; // Окно подсказки Hint
                                         // в виде панели Tool
      AddBiDiModeExStyle(Params.ExStyle);
   }
}

//-------------------------------------------------------------//
//                              Paint                          //
```

```
//-------------------------------------------------------------//
void __fastcall TCalculatorHintWindow::Paint(void)
{
   TRect Rect = ClientRect;

   Canvas->Brush->Color = clBlack;
   Canvas->FillRect(Rect);

   Rect.Left += 4;
   Rect.Top += 4;
   Rect.Right -= 4;
   Rect.Bottom -= 4;

   Frame3D(Canvas,Rect,clBtnShadow,clBtnHighlight,1);

   Canvas->Brush->Color = TColor(0xB4CDBB);
   Canvas->FillRect(Rect);

   Rect.Left += 1;
   Rect.Top += 5;
   Rect.Right -= 1;
   Rect.Bottom -= 1;

   DrawText(Canvas->Handle,
            Caption.c_str(),
            -1,
            &Rect,
            DT_RIGHT
            |DT_NOPREFIX
            |DT_SINGLELINE
            |DrawTextBiDiModeFlagsReadingOnly());
}

//-------------------------------------------------------------//
//                        CalcHintRect                         //
//-------------------------------------------------------------//
Windows::TRect __fastcall
TCalculatorHintWindow::CalcHintRect(int MaxWidth,
                                    const AnsiString AHint,
                                    void*AData)
{
   TRect Rect(0, 0, MaxWidth, 0);
   DrawText( Canvas->Handle,
             AHint.c_str(),
             -1,
             &Rect,
             DT_CALCRECT
             |DT_SINGLELINE
             |DT_NOPREFIX
             |DrawTextBiDiModeFlagsReadingOnly() );
```

```
   // Нужно, чтобы минимальная ширина окна подсказки
   // по крайней мере в три раза превышала высоту
   if((Rect.Right - Rect.Left) < 3 * (Rect.Bottom - Rect.Top))
   {
      Rect.Right = Rect.Left + 3 * (Rect.Bottom - Rect.Top);
   }

   Rect.Right +=20;
   Rect.Bottom +=12;

   return Rect;
}
//--------------------------------------------------------------//
//                            ActivateHint                      //
//--------------------------------------------------------------//
void __fastcall TCalculatorHintWindow::ActivateHint(
                              const Windows::TRect& Rect,
                              const AnsiString AHint)
{
   inherited::ActivateHint(Rect,AHint);
}

//--------------------------------------------------------------//
//                         ActivateHintData                     //
//--------------------------------------------------------------//
void __fastcall
TCalculatorHintWindow::ActivateHintData(
                                  const Windows::TRect&Rect,
                                  const AnsiString AHint,
                                  void*AData)
{
   inherited::ActivateHintData(Rect, AHint, AData);
}

//--------------------------------------------------------------//
//                            IsHintMsg                         //
//--------------------------------------------------------------//
bool __fastcall TCalculatorHintWindow::IsHintMsg(tagMSG& Msg)
{
   return inherited::IsHintMsg(Msg);
}
//--------------------------------------------------------------//
```

Рассмотрим теперь более подробно некоторые аспекты реализации кода, представленного в листинге 5.9, и начнем с конструктора.

Он устроен очень просто. В качестве шрифта выбирается Arial черного цвета (`clBlack`). Это позволяет приводить подсказку с таким же шрифтом, который используется в основном дисплее калькулятора MiniCalculator. Переписанный метод `CreateParams()` практически ничем не отличается от одноименного метода родительского класса `THintWindow`, за исключе-

нием того, что для стиля окна используется WS_POPUP, а стиль WS_BORDER не используется, чтобы исключить очерчивание границы вокруг окна. Остальная часть кода остается такой же. Флаг CS_SAVEBITS, добавленный в поле WindowClass.style параметров Params обозначает, что область экрана под окном подсказки должна быть покрыта подсказкой, как рисунком, а затем снова отображена после удаления подсказки. Это значит, что сообщение WM_PAINT не потребуется посылать всем окнам, которые закрываются подсказкой. Это наиболее практичный способ при работе с такими малыми окнами как подсказка. Наконец, проверим значение глобальной переменной NewStyleControls. Если оно равно true (для операционных систем Win9x и выше), то добавим необходимые расширенные стили, WS_EX_TOOLWINDOW для создания окна подсказки в виде панели инструментов, а также необходимые флаги для двунаправленной поддержки, используя функцию-член AddBiDiModeExStyle(). Более подробную информацию об этой функции можно найти в оперативной справке C++Builder.

Фактическая работа этого класса выполняется в методах CalcHintRect() и Paint(). Метод CalcHintRect() сначала создает переменную Rect типа TRect, задавая значение параметра MaxWidth для свойства Right. Затем адрес переменной Rect передается WinAPI-функции DrawText() вместе со строкой AHint и дескриптором Handle канвы Canvas окна подсказки. Вызов этой функции имеет следующий вид.

```
DrawText(Canvas->Handle, // Дескриптор канвы
         AHint.c_str(),  // Возвращает строку с символом конца
         -1,             // Обозначает, что строка содержит
                         // символ конца строки
         &Rect,          // Область TRect,
                         // размер которой нужно определить
         DT_CALCRECT     // Вызов функции,
                         // которая вычисляет размер области TRect
         |DT_SINGLELINE  // Подсказка в виде одной строки
         |DT_NOPREFIX    // Не обрабатывать префиксные символы (как &)
         |DrawTextBiDiModeFlagsReadingOnly());
```

Флаг DT_CALCRECT указывает, что на самом деле в подсказке отображается не текст, а область типа TRect, необходимая для отображения текста строки AHint с шириной MaxWidth и дескриптором канвы Canvas->Handle.

Сразу после вычисления области отображения подсказки TRect нужно проверить правильность выбранного соотношения высоты и ширины подсказки. При этом преследуется цель отобразить подсказку в виде уменьшенной версии основного дисплея, чтобы ширина подсказки была по крайней мере в три раза больше ее высоты. Кроме того, нужно слева и сверху от текста предоставить дополнительное пространство. Следовательно, нужно прибавить некоторую постоянную величину к возвращаемым значениям Bottom и Right объекта TRect. Это позволяет сдвинуть текст в окне подсказки вправо и вниз. Вот код этих действий.

```
if((Rect.Right - Rect.Left) < 3 * (Rect.Bottom - Rect.Top))
{
   Rect.Right = Rect.Left + 3 * (Rect.Bottom - Rect.Top);
}
Rect.Right  += 20;
Rect.Bottom += 12;
return Rect;
```

Соотношение высоты и ширины, а также константы отступа выбраны произвольным образом и основаны на соображениях здравого смысла о наиболее привлекательном внешнем виде.

Наконец, окно подсказки отображается с помощью переписанного метода `Paint()`. Функциональную часть этого метода можно разбить на две части: в первой — отображается фон окна подсказки, а во второй — текст подсказки. Ниже приведен код первой части.

```
TRect Rect = ClientRect;

Canvas->Brush->Color = clBlack;
Canvas->FillRect(Rect);

Rect.Left += 4;
Rect.Top += 4;
Rect.Right -= 4;
Rect.Bottom -= 4;

Frame3D(Canvas, Rect, clBtnShadow, clBtnHighlight, 1);

Canvas->Brush->Color = TColor(0xB4CDBB);
Canvas->FillRect(Rect);
```

Сначала извлекается структура `TRect`, которая задает границы отображаемой области. Она закрашивается черным цветом, а затем функция VCL-библиотеки `Frame3D()` используется для создания фаски шириной в 4 пикселя. Затем окруженная этой рамкой область закрашивается цветом, который используется в основном дисплее калькулятора MiniCalculator. Наконец, в окне подсказки отображается текст подсказки с помощью показанного ниже кода.

```
Rect.Left += 1;
Rect.Top += 5;
Rect.Right -= 1;
Rect.Bottom -= 1;

DrawText( Canvas->Handle,
          Caption.c_str(),
          -1,
          &Rect,
          DT_RIGHT        // Выравнивание текста по правому краю
          |DT_NOPREFIX    // Без префиксов
          |DT_SINGLELINE  // Текст располагается в одной строке
          |DrawTextBiDiModeFlagsReadingOnly() );
```

Сначала с помощью переменной `Rect` задается область, в которой будет отображен текст. Слева, справа и снизу создается рамка шириной в 1 пиксель, а сверху — рамка шириной 5 пикселей. Затем для отображения текста подсказки в канве `Canvas` окна подсказки используется WinAPI-функция `DrawText()`. Здесь используются те же флаги, которые использовались в методе `CalcHintRect()`, за исключением флага `DT_CALCRECT` (здесь выполняется рисование, а не вычисления), и дополнительный флаг `DT_RIGHT` для выравнивания текста по правому краю области `Rect`. На рис. 5.3 показан внешний вид окна подсказки. В данном примере в окне подсказки показано не число π, а результат деления 22 на 7, который часто используется как грубое приближение числа π.

Для использования класса окна подсказки `TCalculatorWindow` нужно присвоить его тип глобальной переменной `HintWindowClass` так, как показано ниже.

```
HintWindowClass = __classid(TCalculatorHintWindow);
```

Более подробно эта тема рассматривается в следующем разделе.

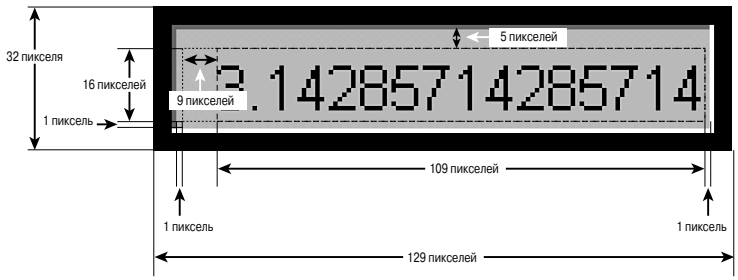


Рис. 5.3. Настраиваемое окно подсказки `TCalculatorHintWindow`

Событие `OnHint` объекта `TApplication`

Компонент `TApplicationEvents`, который находится во вкладке Additional панели компонентов Component Palette, упрощает работу с событиями объекта `TApplication` и особенно — с событием `OnHint` объекта `TApplication`.

При запуске события `OnHint` объекта `TApplication` свойство `Hint` объекта `TApplication` содержит длинную строку подсказки из свойства `Hint` того элемента управления, который вызвал возникновение этого события. Напомним, что если символ | не используется в подсказке для разделения ее короткой и длинной частей, то строка, представляющая подсказку этого элемента управления, будет использована в обоих случаях: как короткая и как длинная. В реализации обработчика события `OnHint` объекта `TApplication` считывание значения длинной части подсказки выполняется очень просто. В этом случае считывается значение свойства `Hint` глобального объекта `Application`, например, так, как показано ниже.

```
AnsiString CurrentLongHint = Application->Hint;
```

В программе MiniCalculator обработка связанных с подсказкой событий выполняется с помощью обработчика события `OnHint` объекта `TApplication`. Особо стоит отметить, что в программе MiniCalculator нам потребуется отобразить в строке состояния обе части подсказки, а потому для некоторых всплывающих подсказок используется другой класс окна подсказки.

Сначала рассмотрим отображение информации в нескольких панелях строки состояния. Для этого нужно разделить подсказку `Application->Hint` на две строки. В первой строке будет содержаться комбинация клавиш ускоренного доступа для каждой кнопки калькулятора, а во второй — описание этой кнопки. Ранее уже было показано использование символа | для разделения короткой и длинной части подсказки. Его можно также использовать для разбиения длинной части подсказки на две половины. Для этого нужно ввести символ | в то место длинной части подсказки, где ее предполагается разбить. Например, клавиша = калькулятора MiniCalculator не имеет короткой подсказки, зато ее длинная часть разбита на две половины: `"Enter"` и `"Equals"`. Строка `"Enter"` обозначает клавишу быстрого доступа для этой кнопки, а строка `"Equals"` содержит описание этой кнопки. Свойство `Hint` этой кнопки содержит в формате `"КороткаяЧасть|ДлиннаяЧасть1|ДлиннаяЧасть2"` значение `"|Enter|Equals"`.

В свойстве `Application->Hint` содержится только строка `"Enter|Equals"`. Следовательно, для извлечения первой части длинной подсказки можно использовать глобальную функцию `GetShortHint()`, а для извлечения второй части длинной подсказки — глобальную функцию `GetLongHint()`. Для исключения первой или второй части длинной подсказки нужно в соответствующем месте такой строки добавить символ |. Дело в том, что при отсутствии символа | функции `GetShortHint()` и `GetLongHint()` вернут целую строку. Например, строка

```
"||LastLongHint"
```

в свойстве Application->Hint идентична строке "|LastLongHint". Тогда при вызове функции GetShortHint() будет получен результат "", а при вызове функции GetLongHint() — результат "LastLongHint". Аналогично, для отображения только первой части длинной подсказки нужно использовать формат "|FirstLongHint|".

Для размещения первой части длинной подсказки во второй панели строки состояния, а второй части — в третьей панели строки состояния, нужно поместить следующий код в обработчик события Application->OnHint.

```
StatusBar1->Panels->Items[1]->Text =
                    GetShortHint(Application->Hint);
StatusBar1->Panels->Items[2]->Text =
                    GetLongHint(Application->Hint);
```

Рассмотрим подробно отображение всплывающих окон подсказок для различных элементов управления программы MiniCalculator. В предыдущем разделе об использовании настраиваемых подсказок уже были описаны способы создания собственных окон подсказок. Кроме того, в нем рассказывалось как присвоить созданное окно подсказки глобальной переменной HintWindowClass с помощью оператора __classid. Для переключения между классами различных окон подсказки нужно считать значение Application->Hint в обработчике события Application->OnHint и присвоить соответствующее значение переменной HintWindowClass. В листинге 5.10 показана реализация обработчика события Application->OnHint из программы MiniCalculator.

Листинг 5.10. Реализация обработчика события Application->OnHint с помощью обработчика TApplicationEvents->OnHint

```
void __fastcall TMainForm::ApplicationEvents1Hint(TObject
                                                    *Sender)
{
   if(Application->Hint=="Ctrl+V|Memory Recall"
      || Application->Hint=="LCD")
   {
      Application->HintHidePause = 10000; // Пауза 10 секунд
      HintWindowClass = __classid(TCalculatorHintWindow);
   }
   else
   {
      Application->HintHidePause =
                          HintDisplayTime; // Задается
                                           // в конструкторе
      HintWindowClass = __classid(THintWindow);
   }

   if(Application->Hint != "LCD")
   {
      StatusBar1->Panels->Items[1]->Text =
                         GetShortHint(Application->Hint);
      StatusBar1->Panels->Items[2]->Text =
                         GetLongHint(Application->Hint);
   }
}
```

Это специализированное окно подсказки предполагается показать после заданной паузы после размещения указателя мыши над кнопкой Memory Recall (SpeedButtonMemoryRecall) или дисплеем калькулятора (LCDScreen). При этом время отображения подсказки будет длиннее, чем обычно, чтобы дать пользователю возможность считать в подсказке даже очень сложное число. Длинная подсказка используется для указания того элемента управления, к которому она относится. Для дисплея LCDScreen длинную подсказку отображать не нужно, поэтому подсказка не отображается в панелях строки состояния, если ее значение равно "LCD", т.е. — длинной части подсказки надписи TLabel для элемента управления LCDScreen.

Для переменной HintDisplayTime в конструкторе задается значение переменной HintHidePause объекта Application. Это позволяет восстановить значение переменной Application->HintHidePause в исходное состояние при отображении стандартного окна подсказки.

Наконец, рассмотрим, как короткие части подсказки присваиваются свойствам Hint этих двух элементов управления. Для них используется одинаковый метод, поэтому рассмотрим только свойство Hint дисплея калькулятора LCDScreen. Всплывающую подсказку следует отображать, только если ширина отображаемого числа больше ширины дисплея калькулятора LCDScreen. Для этого содержимое дисплея LCDScreen проверяется при каждом его обновлении с помощью функции-члена UpdateLCDScreen(). В листинге 5.11 показан способ реализации этого метода.

Листинг 5.11. Реализация метода UpdateLCDScreen()

```
void __fastcall TMainForm::UpdateLCDScreen(
                                    const AnsiString& NewNumber)
{
   int NumberWidth = LCDScreen->Canvas->TextWidth(NewNumber);

   if(Operation == coComplete)
   {
      if( (NumberWidth >= LCDScreen->Width)
          &&(LCDScreen->Alignment == taRightJustify) )
      {
         LCDScreen->Alignment = taLeftJustify;
      }
      else if( (NumberWidth < LCDScreen->Width)
               &&(LCDScreen->Alignment != taRightJustify) )
      {
         LCDScreen->Alignment = taRightJustify;
      }
   }
   else if(LCDScreen->Alignment != taRightJustify)
   {
      LCDScreen->Alignment = taRightJustify;
   }

   LCDScreen->Caption = NewNumber;

   int pos = LCDScreen->Hint.Pos("|");
   int length = LCDScreen->Hint.Length();
```

```
   AnsiString LCDScreenHint
              = LCDScreen->Hint.SubString(pos, length-pos+1);
   LCDScreen->Hint = NewNumber + LCDScreenHint;

   if(NumberWidth >= LCDScreen->Width)
      LCDScreen->ShowHint = true;
   else LCDScreen->ShowHint = false;
}
```

Функция `UpdateLCDScreen()` вызывается всякий раз при изменении отображаемого в дисплее значения, где параметр (строка типа `AnsiString`) передается как константа. Определим сначала ширину в пикселях, которая потребуется для отображения этого числа в канве `Canvas` дисплея калькулятора `LCDScreen`. На основе этой информации, с учетом выполняемой калькулятором операции и текущего параметра `LCDScreen->Width`, зададим оптимальное выравнивание текста. Затем присвоим значение `NewNumber` свойству `Caption` дисплея `LCDScreen`. Остальная часть кода выполняет следующие два действия. Во-первых, новое значение `NewNumber` присваивается свойству `Hint` дисплея `LCDScreen`, так что значение свойства `Hint` становится равным `"NewNumber|LCD"`.

Наконец, в зависимости от ширины нового числа и ширины дисплея `LCDScreen` следует соответственным образом задать значение `ShowHint` дисплея `LCDScreen`. Если новое число шире дисплея `LCDScreen`, то для значения `LCDScreen->ShowHint` следует задать значение `true`. В противном случае нужно отменить отображение подсказки, задав значение `false` для свойства `LCDScreen->ShowHint`.

Повышение практичности за счет улучшенного управления фокусом ввода

Благодаря удобному управлению вводом данных со стороны пользователя и фокусом ввода, максимально упрощается работа с программой. *Фокус ввода* (*input focus*) определяет элемент управления пользовательского интерфейса, с которым в данный момент взаимодействуют стандартные устройства ввода. Под стандартными устройствами ввода подразумеваются клавиатура и мышь, но не устройства, подключенные к последовательным и параллельным портам. Хотя такие устройства также могут быть связаны с фокусом ввода, этот способ обычно не используется при работе с пользовательскими элементами управления. Говорят, что элемент управления имеет фокус ввода в том случае, когда он может отвечать на те или иные действия со стороны стандартного устройства ввода.

После получения фокуса ввода элемент управления может выполнить два действия: сам отреагировать на ввод или переместить фокус ввода к другому элементу управления, который отреагирует на ввод. Ниже кратко рассматриваются обе эти ситуации, а также приводятся примеры их применения.

Отклик на ввод

Программа MiniCalculator откликается на ввод данных с помощью клавиатуры и мыши. Эти функциональные возможности могут быть включены/отключены по желанию пользователя. Программа также предоставляет обратную связь с пользователем для обозначения активации отклика программы на ввод данных с клавиатуры. Для организации отклика со сторо-

ны клавиатуры в основной форме MainForm выполняется обработка событий OnKeyDown и OnKeyUp. Для реализации событий OnKeyDown и OnKeyUp используются две функции — CalculatorKeyDown() и CalculatorKeyUp(). Эти функции присваиваются событиям MainForm с помощью метода установки свойства EnableKeyboardInput. При создании основной формы MainForm для свойства EnableKeyboardInput задается значение true, которое вызывает метод SetEnableKeyboardInput(). Реализация этого метода показана в листинге 5.12.

Листинг 5.12. Реализация метода SetEnableKeyboardInput()

```
void __fastcall TMainForm::SetEnableKeyboardInput(bool NewEnableKeyboardInput)
{
   if(EnableKeyboardInput != NewEnableKeyboardInput)
   {
      FEnableKeyboardInput = NewEnableKeyboardInput;
      if(EnableKeyboardInput)
      {
         OnKeyDown = CalculatorKeyDown;
         OnKeyUp = CalculatorKeyUp;
         StatusBar1->Panels->Items[1]->Width = 60;
      }
      else
      {
         OnKeyDown = 0;
         OnKeyUp = 0;
         StatusBar1->Panels->Items[1]->Width = 0;
      }
      StatusBar1->Invalidate();
   }
}
```

Переменная FEnableKeyboardInput инициализируется значением false, что гарантирует выполнение блока первого оператора if. Обратите внимание на следующие две строки кода, в которых события присваиваются их функциям-обработчикам.

```
OnKeyDown = CalculatorKeyDown;
OnKeyUp = CalculatorKeyUp;
```

Код реализации метода CalculatorKeyDown() показан в листинге 5.13.

Листинг 5.13. Реализация метода CalculatorKeyDown()

```
void __fastcall TMainForm::CalculatorKeyDown(TObject* Sender,
                                             WORD& Key,
                                             TShiftState Shift)
{
   switch(Key)
   {
      case VK_NUMPAD1 :
      case '1'        : ButtonDown(cb1);
                        ButtonPressNumber(cb1);
```

```
                  break;
case VK_NUMPAD2 :
case '2'        : ButtonDown(cb2);
                  ButtonPressNumber(cb2);
                  break;
case VK_NUMPAD3 :
case '3'        : ButtonDown(cb3);
                  ButtonPressNumber(cb3);
                  break;
case VK_NUMPAD4 :
case '4'        : ButtonDown(cb4);
                  ButtonPressNumber(cb4);
                  break;
case VK_NUMPAD5 :
case '5'        : ButtonDown(cb5);
                  ButtonPressNumber(cb5);
                  break;
case VK_NUMPAD6 :
case '6'        : ButtonDown(cb6);
                  ButtonPressNumber(cb6);
                  break;
case VK_NUMPAD7 :
case '7'        : ButtonDown(cb7);
                  ButtonPressNumber(cb7);
                  break;
case VK_NUMPAD8 :
case '8'        : ButtonDown(cb8);
                  ButtonPressNumber(cb8);
                  break;
case VK_NUMPAD9 :
case '9'        : ButtonDown(cb9);
                  ButtonPressNumber(cb9);
                  break;
case VK_NUMPAD0 :
case '0'        : ButtonDown(cb0);
                  ButtonPress0();
                  break;
case VK_DECIMAL : ButtonDown(cbPoint);
                  ButtonPressPoint();
                  break;
case VK_PRIOR   : ButtonDown(cbExponent);
                  ButtonPressExponent();
                  break;
case VK_ADD      : ButtonDown(cbAdd);
                   ButtonPressOperation(coAdd);
                   break;
case VK_SUBTRACT : ButtonDown(cbSubtract);
                   ButtonPressOperation(coSubtract);
                   break;
```

```
      case VK_MULTIPLY : ButtonDown(cbMultiply);
                         ButtonPressOperation(coMultiply);
                         break;
      case VK_DIVIDE   : ButtonDown(cbDivide);
                         ButtonPressOperation(coDivide);
                         break;
      case VK_RETURN   : ButtonDown(cbEquals);
                         ButtonPressEquals();
                         break;
      case VK_BACK     : ButtonDown(cbBackspace);
                         ButtonPressBackspace();
                         break;
      case VK_DELETE   : if(Shift.Contains(ssCtrl))
                         {
                            ButtonDown(cbAllClear);
                            ButtonPressAllClear();
                         }
                         else
                         {
                            ButtonDown(cbClear);
                            ButtonPressClear();
                         }
                         break;
      case 'C' : if(Shift.Contains(ssCtrl))
                 {
                    ButtonDown(cbMemoryAdd);
                    ButtonPressMemoryAdd();
                 }
                 break;
      case 'V' : if(Shift.Contains(ssCtrl))
                 {
                    ButtonDown(cbMemoryRecall);
                    ButtonPressMemoryRecall();
                 }
                 break;
   }
}
```

Как и большинство других функций-обработчиков нажатий клавиш, функция `CalculatorKeyDown()` включает большой блок оператора `switch`, который после определения нажатой клавиши передает управление функции, выполняющей связанную с ней обработку. Сначала вызывается метод `ButtonDown()`, который задает значение `true` для свойства `Down` соответствующей клавиши калькулятора. Это дает пользователю возможность визуально убедиться в правильности нажатой клавиши. Затем для выполнения нужной обработки вызывается функция самой клавиши. Метод `CalculatorKeyUp()` практически идентичен методу `CalculatorKeyDown()`, за исключением нескольких небольших различий. Код реализации метода `CalculatorKeyUp()` показан в листинге 5.14.

Листинг 5.14. Код реализации метода `CalculatorKeyUp()`

```
void __fastcall TMainForm: CalculatorKeyUp(TObject *Sender,
                                           WORD &Key,
                                           TShiftState Shift)
{
   switch(Key)
   {
      case VK_NUMPAD1 :
      case '1'        : ButtonUp(cb1);
                        break;
      case VK_NUMPAD2 :
      case '2'        : ButtonUp(cb2);
                        break;
      case VK_NUMPAD3 :
      case '3'        : ButtonUp(cb3);
                        break;
      case VK_NUMPAD4 :
      case '4'        : ButtonUp(cb4);
                        break;
      case VK_NUMPAD5 :
      case '5'        : ButtonUp(cb5);
                        break;
      case VK_NUMPAD6 :
      case '6'        : ButtonUp(cb6);
                        break;
      case VK_NUMPAD7 :
      case '7'        : ButtonUp(cb7);
                        break;
      case VK_NUMPAD8 :
      case '8'        : ButtonUp(cb8);
                        break;
      case VK_NUMPAD9 :
      case '9'        : ButtonUp(cb9);
                        break;
      case VK_NUMPAD0 :
      case '0'        : ButtonUp(cb0);
                        break;
      case VK_DECIMAL : ButtonUp(cbPoint);
                        break;
      case VK_PRIOR   : // Клавиша <PageUp> для кнопки Exponent
                        // Button работает как переключатель,
                        // поэтому не потребуется обрабатывать
                        // событие KeyUp ее отпускания
                        break;
      case VK_ADD     : ButtonUp(cbAdd);
                        break;
      case VK_SUBTRACT : ButtonUp(cbSubtract);
                         break;
      case VK_MULTIPLY : ButtonUp(cbMultiply);
                         break;
```

```
      case VK_DIVIDE   : ButtonUp(cbDivide);
                         break;
      case VK_RETURN   : ButtonUp(cbEquals);
                         break;
      case VK_BACK     : ButtonUp(cbBackspace);
                         break;
      case VK_DELETE   : if(SpeedButtonAllClear->Down)
                         {
                            ButtonUp(cbAllClear);
                         }
                         else
                         {
                            ButtonUp(cbClear);
                         }
                         break;
      case 'C'         : ButtonUp(cbMemoryAdd);
                         break;
      case 'V'         : ButtonUp(cbMemoryRecall);
                         break;
   }
}
```

Прежде всего заметим, что здесь отсутствует обработка события `KeyUp` отпускания клавиши <Page Up> (`VK_PRIOR`) для кнопки Exponent. Дело в том, что при нажатии кнопки Exponent ее состояния "нажата/отпущена" взаимно переключаются. Следовательно, требуется обработать только событие ее нажатия, которое заключается в вызове метода `ButtonPressExponent()` и установке состояния кнопки в ее текущее состояние.

Также следует отметить, что обработчик нажатия комбинации клавиш <Ctrl+C> (см. листинг 5.13) проверяет наличие значения `ssCtrl` в наборе параметров `Shift`. Однако обработчик события отпускания этой комбинации клавиш отсутствует в листинге 5.14. Дело в том, что нас интересует только, нажата ли клавиша <Ctrl>, когда нажата клавиша <C>, а не когда отпущена клавиша <C>. Проверка отпускания этой комбинации клавиш может привести к тому, что кнопка Memory Add останется нажатой, если клавиша <Ctrl>, будет отпущена перед отпусканием клавиши <C>. Очевидно, что следует избегать возникновения таких ситуаций.

Наконец, отметим, что код обработчика нажатия клавиши <Delete> (`VK_DELETE`), т.е. нажатия кнопок `SpeedButtonClear` и `SpeedButtonAllClear` отличается наличием значения `ssCtrl` в параметре `Shift` (см. листинг 5.13). Однако код обработчика события отпускания этой клавиши (см. листинг 5.14) не проверяет состояние клавиши <Ctrl>. Вместо этого проверяется значение свойства `Down` кнопки `SpeedButtonAllClear`, т.е. нажата ли она. Если кнопка нажата, то она отпускается; в противном случае (т.е. когда нажата кнопка `SpeedButtonClear`) она должна быть отпущена.

Перемещение фокуса ввода

В программе MiniCalculator основная форма (`MainForm`) должна иметь фокус ввода, чтобы она могла реагировать на ввод с клавиатуры. Однако в некоторых ситуациях фокус ввода может покидать основную форму. Эти ситуации отслеживаются, и фокус ввода возвращается в основную форму. Фокус ввода может покидать основную форму в двух случаях. Во-первых,

когда теряет фокус ввода приложение в целом. Эта ситуация не рассматривается здесь, потому что потеря фокуса ввода основной формой в этом случае закономерна. Во-вторых, основная форма может потерять фокус ввода, когда его получает незакрепленный элемент управления, например, при выборе, перетаскивании или изменении размеров незакрепленного элемента управления. Дисплей `LCDPanel` может быть откреплен от основной формы. Следовательно, в программе MiniCalculator следует отслеживать случаи открепления панели `LCDPanel` (т.е. когда `LCDPanel>Floating == true`) и получения ею фокуса ввода. Для возвращения фокуса в основную форму в этом случае нужно использовать метод `SetFocus()` основной формы `MainForm`. Например, основная форма теряет фокус, если изменять размер откепленной панели `LCDPanel`. Следовательно, для возвращения фокуса ввода в основную форму нужно вставить в обработчик события `OnResize` дисплея `LCDPanel` приведенный ниже код.

```
if(LCDPanel->Floating)SetFocus();
```

Выше в этой главе в разделе об использовании строки состояния уже рассматривались способы отслеживания состояния фокуса ввода основной формы. При создании интерфейса пользователя часто используется следующий способ перевода фокуса ввода: по завершении редактирования фокус ввода автоматически переносится в следующее текстовое поле. Это позволяет существенно упростить заполнение текстовых полей. Пример применения этого метода приводится в описании класса `FromToForm` из проекта `Focus.bpr`, который находится на прилагаемом к книге компакт-диске. Читателю потребуется только создать событие `OnKeyUp` для каждого текстового окна. В этом примере предположим, что пользователь вводит два 4-значных числа в каждом текстовом поле. После заполнения первого текстового поля фокус ввода перемещается во второе текстовое поле. После заполнения второго текстового поля фокус ввода перемещается в кнопку OK. Если второе текстовое поле заполнено, а первое — нет, фокус ввода возвращается в первое поле. В листингах 5.15 и 5.16 приведены способы обработки событий в текстовых полях `Edit1` и `Edit2`, соответственно.

Листинг 5.15. Реализация обработчика события `Edit1KeyUp()`

```
void __fastcall TFromToForm::Edit1KeyUp(TObject* Sender,
                                        WORD& Key,
                                        TShiftState Shift)
{
   if(Edit1->Text.Length() == Edit1->MaxLength)
   {
      FinishNumberComplete = true;
      if(StartNumberComplete)
      {
         BitBtnOK->Enabled = true;
         BitBtnOK->SetFocus();
      }
      else Edit2->SetFocus();
   }
   else
   {
      FinishNumberComplete = false;
      BitBtnOK->Enabled = false;
   }
}
```

Листинг 5.16. Реализация обработчика события `Edit2KeyUp()`

```
void __fastcall TFromToForm::Edit2KeyUp(TObject* Sender,
                                        WORD& Key,
                                        TShiftState Shift)
{
   if(Edit2->Text.Length() == Edit2->MaxLength)
   {
      StartNumberComplete = true;
      if(FinishNumberComplete)
      {
         BitBtnOK->Enabled = true;
         BitBtnOK->SetFocus();
      }
      else Edit1->SetFocus();
   }
   else
   {
      StartNumberComplete = false;
      BitBtnOK->Enabled = false;
   }
}
```

Чтобы при отображении формы `FromToForm` фокус ввода находился в первом текстовом окне (`Edit1`), следует задать значение 0 для его свойства `TabOrder`. Для свойства `TabOrder` второго текстового окна `Edit2` зададим значение 1, для кнопки `BitBtnOK` — значение 2, а для кнопки `BitBtnCancel` — значение 3. Свойство `TabOrder` упрощает управление порядком перехода между элементами управления в группе, а потому следует тщательно подумать о том, какой порядок перехода между ними является наиболее оптимальным. Откройте проект `Focus` и внимательно изучите его. Учтите, что в текстовые поля этого проекта можно вводить любые символы. Но можно, например, создать код для фильтрации символов, которые не являются числами. Еще более эффективное решение заключается в использовании инструмента редактирования, который позволяет отфильтровывать целые строки символов в текстовом поле. Пример такого компонента приводится в разделе о переопределении динамических функций главы 11.

Повышение практичности за счет улучшения внешнего вида

Улучшение внешнего вида пользовательского интерфейса означает не только создание внешне приятного интерфейса, но и предоставление дополнительной информации на основе символов, цвета, макета и формы. Оптимизируя внешний вид пользовательского интерфейса приложения, можно сделать его более интуитивно понятным и узнаваемым. Таким образом, можно одним выстрелом убить сразу двух зайцев. Пользователям потребуется меньше времени не только на изучение интерфейса, благодаря узнаваемости одних элементов и интуитивной простоте других, но и на работу с этим интерфейсом. Очевидно, что использование этих преимуществ значительно повысит эффективность использования приложения.

Как же выбрать наилучший способ представления пользовательского интерфейса? Это зависит от типа разрабатываемого приложения. При создании приложения, выполняющего за-

дачу, имеющую широко распространенный и понятный аналог в реальном мире, в качестве руководства по созданию пользовательского интерфейса лучше всего использовать именно этот аналог. Если такого аналога в реальном мире нет, то постарайтесь сделать интерфейс максимально узнаваемым. Это обычно означает, что пользовательский интерфейс должен быть похож на другие хорошо известные приложения. Особое внимание следует уделить тем приложениям, которые выполняют аналогичную задачу или тип задачи.

Рассмотрим в качестве примера приложение для набора телефонного номера. Как должен выглядеть интерфейс такого приложения? Для достижения максимальной узнаваемости лучше всего создать интерфейс, имеющий вид обычного телефона. Большинству пользователей, вероятно, более знакомым покажется интерфейс в виде телефона, а не текстового окна, в которое нужно ввести номер телефона. Однако, реализовать этот вариант не так уж просто. Нужно помнить о контексте, в котором используется данный интерфейс (т.е. о том, что это все-таки компьютерная программа), и выполняемой задаче. Создаваемое приложение является компьютерной программой, и пользователь ожидает от нее определенного поведения. Например, после щелчка на кнопке он предполагает увидеть эффект нажатия. Если эффект нажатия не будет имитирован должным образом, то создание такого интерфейса — пустая трата времени, независимо от того, насколько он привлекателен. Нужно также проявить осторожность при копировании компонентов из реального мира, чтобы искусственно не налагать на приложение ограничений реального мира. Например, интерфейс в виде кнопочной панели телефона будет иметь очень привлекательный внешний вид. Но если пользователям будет позволено набирать номер только с помощью щелчков мышью на каждой цифре, то такой интерфейс им вряд ли понравится. Дело в том, что набор номера телефона часто проще и быстрее выполнить с помощью клавиатуры, а потому эту возможность также следует учесть.

Функциональность интерфейса важна не менее его внешнего вида. В случае номеронабирателя необходимо учесть тот случай, когда возможность выполнения телефонных звонков может оказаться вторичной задачей по отношению к основной задаче приложения. Например, приложение может быть предназначено для получения и отправки факсимильных сообщений с возможностью их редактирования. В таком случае интерфейс приложения в виде обычного телефона может оказаться не очень удачным. Гораздо более знакомым и удобным для пользователей может оказаться традиционный многодокументный интерфейс на основе нескольких файлов.

При выборе многодокументного интерфейса на основе файлов следует учесть другие факторы внешнего вида. Например, первая команда меню обычно имеет название File. Кроме того, среди команд меню обычно всегда присутствует команда Window и т.д. Нетрудно убедиться, что определение наиболее удачного для пользователей типа интерфейса представляет собой довольно сложную задачу с учетом многих факторов.

Проектирование внешнего вида интерфейса тесно связано с его функциями приложения, а потому этот этап создания приложения может продолжаться достаточно долго. В программе MiniCalculator пользовательский интерфейс был выбран в виде обычного калькулятора (хотя его функциональность значительно превосходит ограничения, налагаемые на обычный калькулятор). Это значит, что кнопки приложения снабжены символьной, а не текстовой информацией. В результате интерфейс содержит много графических элементов и практически полностью состоит из изображений кнопок. Использование кнопок вместо текста обладает тем преимуществом, что для них не требуются дополнительные ресурсы при использовании этой программы в разных языковых и географических зонах. Это замечание, конечно не имеет никакого отношения к текстовым строкам, которые содержат описание кнопок в третьей панели строки состояния программы MiniCalculator.

Для обозначения кнопки или команды меню желательно использовать какой-либо символ, даже если они сопровождаются текстом. Пользователь сможет прочесть текст и мысленно связать его с этим символом. Позднее, когда он запомнит все символы, ему не надо будет чи-

тать этот текст. Следует предоставить пользователю возможность удалять текстовые описания с элементов управления, если он уже не пользуется ими. Это дает возможность освободить рабочее пространство на экране компьютера, что особенно важно при работе с приложениями с MDI-интерфейсом, насыщенным многочисленными панелями инструментов и компонентов, где необходимо освободить максимально возможное пространство в родительском окне для работы с его дочерними окнами. Полезным может также оказаться возможность перемещения текстовых надписей.

Далее в этом разделе описываются способы использования символов и текста для улучшения внешнего вида интерфейса.

Кнопки только с символами

Для создания кнопок без текста или с текстом как части символа на кнопке лучше всего использовать кнопку типа `TSpeedButton`. Преимущество этого типа кнопки заключается в том, что во время щелчка или получения фокуса на ней не отображается прямоугольник получения фокуса, а потому хорошо оформленная кнопка не будет искажена. Кроме того, кнопка типа `TSpeedButton` имеет свойство `Down`, благодаря чему имеется еще одно дополнительное состояние. Свойство `GroupIndex` также полезно для управления кнопками в составе целых групп и позволяет сэкономить большое количество строк кода. Недостатком кнопки типа `TSpeedButton` является отсутствие дескриптора окна и некоторых событий, которые имеются у кнопки типа `TBitBtn`. Дело в том, что она происходит от типов `TGraphicControl` и `TWinControl`. Окончательный выбор типа кнопки все же должен сделать программист. Однако представленная здесь информация применима для любой кнопки. Обычно кнопку типа `TButton` не рекомендуется использовать, если только она не будет содержать какое-то изображение.

Кнопку можно полностью покрыть созданным вами изображением. Именно таким образом были сделаны кнопки в программе MiniCalculator. Однако в этом случае следует правильно задать размеры изображения и кнопки. На рис. 5.4 показано изображение для кнопки сложения (+).

Рис. 5.4. Рельефный макет кнопок типа `TSpeedButton` и `TBitBtn`

Каждое изображение представляет некое состояние кнопки. Часто кнопка в состоянии `Clicked` (щелкнута) и `Down` (нажата) выглядит одинаково. Размеры кнопки и изображения устанавливаются очень просто. Задайте размер кнопки равным размеру изображения, которое предполагается разместить на кнопке. Затем прибавьте 1-пиксельную границу сверху и слева от рисунка, а 2-пиксельную границу снизу и справа кнопки. Полученный вид кнопки показан на рис. 5.4. Обратите внимание, что изображения в состояниях `Clicked` и `Down` автоматически сдвигаются вниз и вправо. Следовательно, размер изображения в таких состояниях будет немного меньше. Это достигается размещением белой или (предпочтительно) прозрачной 2-пиксельной границы снизу и справа от каждого изображения. Белый цвет использован для ясности на рис. 5.4. Для использования изображения в кнопке сохраните рельефное изображение кнопки в растровом формате (bitmap) и присвойте его свойству `Glyph` (Глиф) в режиме конструирования формы. Не забудьте задать соответствующее значение для свойства `NumGlyphs`. Проектируя кнопки, следует иметь в виду, что при нажатии кнопки вокруг нее слева и сверху

появится черная граница, а справа и снизу — белая. Изображения следует создавать с учетом этих особенностей, т.к. в противном случае кнопки будут выглядеть необычно.

Для двух кнопок памяти калькулятора MiniCalculator используются надписи M+ и MR. Однако следует избегать использования таких надписей, поскольку они являются не символами, а сокращенной формой текстовой записи функций кнопки. Для кнопки записи в память (M+) лучше использовать какой-нибудь символ копирования, а для кнопки считывания содержимого памяти (MR) — какой-нибудь символ вставки. Читателю предлагается в качестве упражнения самостоятельно создать такие символы.

Свойство GroupIndex класса TSpeedButton

В программе MiniCalculator широко используется свойство GroupIndex класса TSpeedButton для управления поведением кнопок, используемых в интерфейсе. Например, три кнопки для указания применяемой системы счисления — восьмеричной, десятичной или шестнадцатеричной использует одно и то же значение 2 свойства GroupIndex, а также значение false свойства AllowAllUp. Никакие другие кнопки этого интерфейса не имеют такого значения свойства GroupIndex. Это значит, что только одна кнопка систем счисления может быть нажата в определенный момент времени, причем одна из них непременно должна быть нажата. Для достижения этой цели не потребуется создавать никакого специального кода. Достаточно просто задать для этих кнопок значение свойства GroupIndex. Кнопка Exponent (SpeedButtonExponent) имеет отличное от других значение 3 свойства GroupIndex, что дает возможность нажимать или отпускать ее независимо от состояния других кнопок интерфейса. Остальные кнопки интерфейса используют значение 1 свойства GroupIndex, а также значение true для свойства AllowAllUp. Это значит, что в любой момент времени все кнопки могут быть отпущены, а нажимать их можно только по одной. Этот эффект также достигается без написания какого-либо кода, что делает свойство GroupIndex класса TSpeedButton очень полезным инструментом создания интерфейса.

Подавление мерцания

Для подавления эффекта мерцания при перемещении панели с элементами управления (типа TSpeedButton) в программе MiniCalculator, для их свойства ControlStyle следует задать значение csOpaque в режиме конструктора формы MainForm. Значение csOpaque означает, что клиентская часть элемента управления будет заполнена, а потому не потребуется выполнять перерисовку расположенных под ним других элементов управления. Чтобы почувствовать разницу, закомментируйте в коде конструктора формы MainForm строки, где устанавливается флаг csOpaque, и запустите программу.

Изображения, дополняющие текст

В программе MiniCalculator в раскрывающихся и всплывающих меню наряду с текстом используются изображения. Проще всего это сделать для команды меню TMenuItem, добавив изображение в качестве значения его свойства Bitmap. Если эту команду меню не предполагается отключать и для нее достаточно использовать изображение с размерами 16×16 пикселей, то это — простой и удобный способ разместить изображение рядом с командой меню. Однако свойство Bitmap непригодно в случаях, когда команду меню требуется отключить. Дело в том, что изображение для отключенной команды меню будет генерировано на основе созданного изображения. Во многих случаях генерированное изображение выглядит удовлетворительно, но если это не так, то придется генерировать собственное изображение и на-

страиваемую команду меню. Аналогично, для использования изображения с размерами, отличными от 16×16 пикселей, следует использовать собственную команду меню.

В программе MiniCalculator для добавления изображений к командам меню ControlBarPopup используется свойство Bitmap, но команда View основной линейки команд меню MainMenu является пользовательской применяется для включения нестандартных изображений в команды меню. Создать пользовательскую команду меню гораздо проще, чем кажется. Для этого необходимо использовать не-NULL значение свойства Images в родительском меню или задать значение true для свойства OwnerDraw. Затем необходимо создать обработчик события OnDrawItem или обработчик события OnAdvancedDrawItem. Событие OnAdvancedDrawItem предоставляет более подробную информацию о состоянии команды меню и потому предпочтительнее использовать именно ее. Единственный обработчик событий для всех команд в линейке меню View основного меню совместно используется всеми командами меню этой линейки. Свойство Tag команды меню используется для определения изображения, которое должно отображаться обработчиком. Этот обработчик называется ViewMenuItemsAdvancedDrawItem() и именно он отвечает за обработку событий OnAdvancedDrawItem команд меню конструктора основной формы MainForm. Например, для команды меню View⇨Display используется следующий код.

```
ViewDisplay->OnAdvancedDrawItem = ViewMenuItemsAdvancedDrawItem;
```

В листинге 5.17 приведен код реализации метода ViewMenuItemsAdvancedDrawItem().

Листинг 5.17. Реализация метода ViewMenuItemsAdvancedDrawItem()

```
void __fastcall TMainForm::ViewMenuItemsAdvancedDrawItem(
                                      TObject* Sender,
                                      TCanvas* ACanvas,
                                      const TRect& ARect,
                                      TOwnerDrawState State)
{
   TMenuItem*MenuItem = dynamic_cast<TMenuItem*>(Sender);

   if(MenuItem)
   {
      //Этап 1 - Сохранение изменяемых свойств канвы ACanvas
      TColor OldFontColor = ACanvas->Font->Color;
      TColor OldBrushColor = ACanvas->Brush->Color;

      int TextOffset =ARect.Left+1;

      try
      {
         // Этап 2 - Создание отмеченной области и
         //          изображения. Вид изображения зависит от
         //          типа выбранного элемента и установки флага.
         std::auto_ptr<Graphics::TBitmap> CheckedImage(new Graphics::TBitmap());
         std::auto_ptr<Graphics::TBitmap> ToolbarImage(new Graphics::TBitmap());

         // Этап 3
         ViewMenuImageList->GetBitmap(MenuItem->Tag, ToolbarImage.get());
         ToolbarImage.get()->Transparent = true;
```

```
    // Этап 4
    if(State.Contains(odChecked))
    {
       if(State.Contains(odSelected))
       {
          MenuCheckImageList->GetBitmap(1,
                                        CheckedImage.get());
       }
       else
       {
          MenuCheckImageList->GetBitmap(0,
                                        CheckedImage.get());
       }

       ACanvas->Draw(ARect.Left+1,
                     ARect.Top+2,
                     CheckedImage.get());
    }

    // Этап 5
    ACanvas->Draw(ARect.Left+21,
                  ARect.Top+2,
                  ToolbarImage.get());

    TextOffset = ARect.Left +60;
}
__finally
{
    // Этап 6
    if(State.Contains(odSelected))
    {
       ACanvas->Font->Color = clHighlightText;
       ACanvas->Brush->Color = clHighlight;
    }
    else
    {
       ACanvas->Font->Color = clWindowText;
       ACanvas->Brush->Color = clMenu;
    }

    ACanvas->FillRect(Rect(TextOffset,
                       ARect.Top,
                       ARect.Right,
                       ARect.Bottom ));

    // Теперь отображается текст :@)
    // Используйте WinAPI-функцию DrawText, так как
    // она корректно отображает символы подчеркивания :-)
```

```
        DrawText(ACanvas->Handle,
                 MenuItem->Caption.c_str(),
                 MenuItem->Caption.Length(),
                 &Rect(TextOffset+2, ARect.Top+2,
                       ARect.Right, ARect.Bottom),
                 DT_EXPANDTABS|DT_SINGLELINE|DT_LEFT );

        // Этап 7
        ACanvas->Font->Color = OldFontColor;
        ACanvas->Brush->Color = OldBrushColor;
      }
   }
}
```

Код функции `ViewMenuItemsAdvancedDrawItem()` отображает три объекта. Во-первых, — изображение установленного или снятого флажка для команды меню. Затем он создает рисунок шириной 36 и высотой 18 пикселей, который представляет панель, отображение которой на экране можно изменять. Наконец, он создает текстовое описание панели с подчеркиванием клавиши быстрого доступа. Для этого используется WinAPI-функция `DrawText()`, которая была описана выше в разделе об использовании строки состояния `TStatusBar`. Отображаемые на экране рисунки хранятся в списке изображений `ViewMenuImageList`, а индекс каждого изображения соответствует свойству `Tag` команды меню, которая вызвала это событие. Указатель на каждую команду меню, которая запускает событие, извлекается с динамическим приведением (`dynamic_cast`) типа `Sender` к типу `TMenuItem*`. Остальную часть функции можно описать следующим образом.

1. Сохранение свойств канвы `ACanvas`, которые предполагается изменить.
2. Создание двух изображений: флажка и панели. Для указателя на класс изображения `TBitmap` используется шаблон класса `auto_ptr<>` библиотеки C++ Standard Library, а потому он будет автоматически удален при возникновении исключительной ситуации. Эти и все другие действия вплоть до этапа 5 выполняются внутри блока `try`. Этап 6 выполняется внутри блока `__finally`. Именно в нем приводятся действия по созданию текста. Если исключительная ситуация возникает при создании изображений возле команды меню, то конструкция `try/_finally` гарантирует выполнение кода создания текста.
3. Извлечение изображения для панели, основанного на свойстве `Tag` команды меню, из списка изображений `ViewMenuImageList`. Далее для свойства `Transparent` задается значение `true`. Обратите внимание, как используется метод `get()` для извлечения указателя на `TBitmap`.
4. Если параметр `State` содержит флаг `odChecked`, то рядом с командой меню отображается флажок, снятый или установленный. Что именно будет отображено на экране, зависит от наличия в параметре `State` флага `odSelected`. Изображение флажка извлекается из списка `MenuCheckImageList`.
5. Затем отображается панель и задается значение `60` для переменной `TextOffset`, что позволяет создать промежуток между флажком и изображением панели. Это значение определяет место, в котором начинается отображение команды меню с текстовым описанием.

6. Далее отображается текстовое описание команды меню. Отступ текста определяется величиной переменной `TextOffset`. Если исключительная ситуация возникла в предыдущем блоке `try`, то это значение будет равно `0`, в противном же случае — `60`, что позволяет рисовать изображения возле команды меню. Сначала задается цвет для текста и фона на основе наличия флага `odSelected` параметра `State`. Далее фон заполняется цветом `Brush->Color`, а потом с помощью WinAPI-функции `DrawText()` отображается текст с цветом `Font->Color`.
7. По окончании отображения команды меню свойствам канвы `ACanvas` возвращаются прежние значения.

При реализации события `OnAdvancedDrawItem` осталась невыполненной еще одна задача. Для каждой команды меню нужно создать событие `OnMeasureItem`, чтобы обеспечить достаточно пространства для изображений в пользовательских командах меню. Здесь, как и при создании события `OnAdvancedDrawItem`, для всех команд меню используется одинаковый код этого события. С этой целью событие `OnMeasureItem` приравнивается функции `ViewMenuItemsMeasureItem()` для каждой команды меню в конструкторе основной формы `MainForm`. Код реализации этой функции представлен в листинге 5.18.

Листинг 5.18. Код реализации функции `ViewMenuItemsMeasureItem()`

```
void __fastcall TMainForm::ViewMenuItemsMeasureItem(
                                        TObject* Sender,
                                        TCanvas*ACanvas,
                                        int& Width,
                                        int& Height)
{
   TMenuItem*MenuItem = dynamic_cast<TMenuItem*>(Sender);

   if(MenuItem)
   {
      Height = 22;
      Width = ACanvas->TextWidth(MenuItem->Caption) + 62;
   }
}
```

Для определения инициатора события выполняется динамическое приведение типа `Sender` к типу `TMenuItem`. В случае удачного завершения этой операции для параметра `Height` задается значение `22` для указания высоты изображений панели плюс 2-пиксельная граница над и под каждым изображением. Параметр `Width` содержит значение ширины изображения в пикселях, которое будет занято элементом `Caption` команды меню с текущей канвой команды меню, указанной параметром `ACanvas`. Для получения этих сведений используется метод `TextWidth`. К этой величине добавляется значение `62` для изображения флажка (шириной 18 пикселей), изображения панели (шириной 36 пикселей), границ между ними, края команды меню, а также текстового описания (из свойства `Caption`).

Как видите, можно довольно легко и быстро создавать пользовательские команды меню, которые могут существенно отличаться от стандартных команд меню, улучшить их внешний вид и предоставлять более гибкие методы работы с ними. За счет совместного использования обработчиков событий группами команд меню можно до минимума сократить размер кода, что делает пользовательские команды меню еще более привлекательными.

Цвет как визуальная подсказка

Использование цвета позволяет создать более интуитивно понятный интерфейс. На примере программы MiniCalculator можно убедиться, как цвет кнопок используется для визуального выделения групп кнопок в соответствии с их назначением. Серые кнопки предназначены для выполнения операций, черные — для ввода значений, оранжевые — для редактирования или очистки значения, синие — для функций памяти, а зеленые — для изменения системы счисления. Использование цвета, таким образом, позволяет мысленно связать некоторые функции с вполне определенными кнопками, даже если вам неизвестны эти функции. На подбор наиболее удачного цвета может уйти достаточно много времени, поскольку результат зависит от персональных особенностей восприятия цветов пользователями, например дальтониками. Удачный выбор цветов для улучшения внешнего вида и полезности интерфейса зависит от множества факторов, но применение цветов для создания групп представляет собой простой и эффективный метод, который можно легко реализовать во многих ситуациях. В зависимости от типа приложения цвет также можно использовать для других очевидных задач.

Форма элементов управления

Обычно при программировании интерфейса желательно избежать присущей Windows прямоугольной формы объектов и творчески отнестись к созданию непрямоугольных элементов управления. Для создания разнообразных форм существует несколько способов, но здесь будет представлен только один простой способ, который можно вполне успешно применять во многих ситуациях.

Для создания непрямоугольного элемента управления (наследника класса `TWinControl`) рекомендуется использовать WinAPI-функции создания областей и присвоить полученную область данному элементу управления. Функция создания области используется для описания отображаемой области. Области могут быть прямоугольными, эллиптическими, многоугольными или комбинированными. Они могут быть закрашенными, инвертированными и окруженными рамкой. Кроме того, они также могут быть использованы для проверки наличия в них курсора мыши. Так как области могут иметь нерегулярную форму, их можно использовать для создания несимметричных кнопок. Область является объектом графического устройства (graphics device interface — GDI), а потому должна быть создана явно. Для создания области используйте одну из WinAPI-функций, перечисленных в табл. 5.7.

Таблица 5.7. Функции создания области

Функция	Описание
`CreateEllipticRgn()`	Создает эллиптическую область по заданным координатам левого верхнего и правого нижнего углов прямоугольника, окаймляющего этот эллипс
`CreateEllipticRgnIndirect()`	Создает эллиптическую область по заданному указателю на структуру `TRect`, т.е. прямоугольник, окаймляющий этот эллипс
`CreatePolygonRgn()`	Создает многоугольную область по заданному массиву точек, которые определяют эту область, и режиму закрашивания, который определяет способ закрашивания частей многоугольника
`CreatePolyPolygonRgn()`	Создает один или несколько многоугольников по заданному массиву точек. Также указывается способ закрашивания частей каждого многоугольника

Окончание табл. 5.7

Функция	Описание
`CreateRectRgn()`	Создает прямоугольную область по заданным координатам левого верхнего и правого нижнего углов прямоугольника, окаймляющего эту область
`CreateRectRgnIndirect()`	Создает прямоугольную область по заданному указателю на структуру `TRect`, т.е. прямоугольник, окаймляющий эту область
`CreateRoundRectRgn()`	Создает прямоугольную область с закругленными углами по заданным координатам левого верхнего и правого нижнего углов прямоугольника, окаймляющего эту область, и по ширине и высоте эллипса, который используется для скругления углов прямоугольника
`ExtCreateRegion()`	Создает область на основе преобразования существующей области
`CombineRgn()`	Создает область на основе комбинации двух других областей по заданному режиму их объединения

Более подробную информацию о перечисленных в табл. 5.7 функциях создания областей можно найти в оперативной справке о функциях Win32 SDK. Еще один источник информации по этой теме и связанным с Win32 GUI вопросам можно найти в прекрасной книге *Win32 Programming*, Rector and Newcomer, Addison-Wesley, 1997.

Сразу после создания области WinAPI-функцию `SetWindowRgn()` можно использовать для присвоения этой области оконному элементу управления. Объявление функции `SetWindowRgn()` выглядит следующим образом.

```
int SetWindowRgn(
   HWND hWnd,   // дескриптор окна, область которого нужно задать
   HRGN hRgn,   // дескриптор области
   BOOL bRedraw // флаг перерисовки окна - для видимых элементов
                // он обычно равен TRUE
                );
```

Возвращаемое значение, не равное нулю, означает успешное выполнение функции. В качестве примера использования областей для создания непрямоугольных элементов управления в программе MiniCalculator применяется панель кнопок для представления таких констант, как, например, число π. Они представляют собой компоненты `TBitBtn`, которые имеют вид прямоугольных областей со скругленными углами. Для иллюстрации необходимого для этого кода в листинге 5.19 показан фрагмент кода из конструктора основной формы `MainForm`, который предназначен для создания прямоугольной кнопки `ConstantPieBitBtn` для числа π с скругленными углами.

Листинг 5.19. Создание прямоугольной кнопки со скругленными углами

```
// Этап 1 - Создание области
HRGN hRoundRectRegion1
  = CreateRoundRectRgn(
          0,                          // Левый край
          0,                          // Верхний край
```

```
          ConstantPieBitBtn->Width+1,  // Правый край
          ConstantPieBitBtn->Height+1, // Нижний край
          14,                          // Ширина эллипса
          14 );                        // Высота эллипса
// Этап 2 - Присвоение области кнопке
SetWindowRgn( ConstantPieBitBtn->Handle,
              hRoundRectRegion1,
              TRUE );
```

Обычно по окончании работы область удаляется с помощью макроса `DeleteRgn()` с дескриптором области в качестве единственного аргумента. Однако при присвоении дескриптора области с помощью функции `SetWindowRgn()` операционная система получает права владения этой областью. Поэтому не следует выполнять какие-либо дальнейшие вызовы функций на основе дескриптора области, особенно не следует удалять его.

Чтобы создать изображение для непрямоугольной кнопки, придется немного поэкспериментировать. Для оформления скругленных углов придется их сгладить. Эффективность сглаживания зависит от правильного выбора цветов, чтобы углы кнопки плавно смешались с фоном. Удачный пример такого сглаживания можно найти, рассматривая кнопки в программе MiniCalculator.

При этом следует иметь в виду, что для успешного оформления непрямоугольных кнопок, вероятно, придется создать собственный компонент-кнопку и переопределить метод `OnPaint()`, чтобы получить полный контроль над перерисовкой кнопки. Использование класса `TBitBtn` для этой цели не будет удовлетворять строгим требованиям. Фокус ввода прямоугольника и используемая по умолчанию обработка щелчков мыши становятся практически бесполезными при работе с непрямоугольными кнопками. Однако совсем нетрудно создать собственный компонент-кнопку. В главе 9, где описывается создание настраиваемых пользовательских компонентов, приводятся более подробные сведения об этом.

Повышение практичности благодаря возможности настройки пользовательского интерфейса

Прекрасный способ повышения практичности заключается в предоставлении пользователю инструментов для настройки внешнего вида пользовательского интерфейса. Они могут быть очень простыми, например для изменения цвета разных элементов интерфейса, или сложными — позволяя откреплять части интерфейса или переупорядочивать их. Большое значение также имеет возможность изменения размеров интерфейса и скрытия некоторых элементов интерфейса по мере надобности. Среди всех этих методов самым простым, вероятно, является управление цветом компонентов интерфейса. Все, что нужно для этого сделать, — это предоставить пользователю доступ к свойствам `Color` элементов управления пользовательского интерфейса. В некоторых случаях это не совсем удобно, например при работе с интерфейсом, насыщенном графическими элементами, потому что для настройки будут доступны только небольшие области интерфейса. Типичным примером такой ситуации является программа MiniCalculator. Для удовлетворения ожиданий пользователя рекомендуется там, где это возможно, использовать только системные цвета. В табл. 5.8 перечислены системные цвета, и приведена их краткая характеристика.

Таблица 5.8. Системные цвета

Системный цвет	Описание
clBackground	Текущий цвет фона рабочего стола Windows
clActiveCaption	Текущий цвет строки заголовка активного окна
clInactiveCaption	Текущий цвет строки заголовка неактивного окна
clMenu	Текущий цвет фона команд меню
clWindow	Текущий цвет фона окна
clWindowFrame	Текущий цвет рамки окна
clMenuText	Текущий цвет текста команд меню
clWindowText	Текущий цвет текста в окне
clCaptionText	Текущий цвет текста в строке заголовка активного окна
clActiveBorder	Текущий цвет рамки активного окна
clInactiveBorder	Текущий цвет рамки неактивного окна
clAppWorkSpace	Текущий цвет рабочей области приложения
clHighlight	Текущий цвет фона выбранного текста
clHighlightText	Текущий цвет выбранного текста
clBtnFace	Текущий цвет поверхности кнопки
clBtnShadow	Текущий цвет тени кнопки
clGrayText	Текущий цвет “серого” текста
clBtnText	Текущий цвет текста на кнопке
clInactiveCaptionText	Текущий цвет текста в строке заголовка неактивного окна
clBtnHighlight	Текущий цвет подсвеченного текста на кнопке
cl3DDkShadow	Тень от трехмерного элемента управления
cl3Dlight	Яркий цвет для трехмерного элемента управления (для ярко освещенных краев)
clInfoText	Цвет текста для элементов управления ToolTip
clInfoBk	Цвет фона для элементов управления ToolTip

Например, для текста в окне следует использовать цвет clWindowText, а для подсвеченного текста — цвет clHighlightText. Эти цвета уже указаны в предпочтениях пользователя и, следовательно, будут удачным вариантом оформления интерфейса. В этом разделе основное внимание уделяется изменению размеров, выравниванию, скрытию и закреплению элементов пользовательского интерфейса. В программе MiniCalculator предусмотрены все эти возможности, а потому именно ее мы и рассмотрим в качестве примера. Остальная часть этого раздела разбита на несколько подразделов, каждый из которых посвящен описанию отдельного метода.

Закрепление элементов управления

В программе MiniCalculator основной дисплей может быть откреплен от остальной части интерфейса, а затем перемещен в другое место с независимым от этого изменением размеров. Панель основного дисплея LCDPanel имеет тип TPanel. Чтобы предоставить пользователю возможность откреплять эту панель от основной формы, необходимо выполнить три действия.

1. Задать для свойства LCDPanel->DragKind значение dkDock.
2. Задать для свойства LCDPanel->DragMode значение dmAutomatic.
3. Задать для свойства MainForm->DockSite значение true.

Все эти действия выполняются в режиме конструктора формы с помощью панели объектов Object Inspector. Этих действий достаточно для того, чтобы сделать панель LCDPanel закрепляемой, но чтобы работа с ней была простой и приятной, потребуется выполнить некоторые дополнительные действия.

Необходимо рассмотреть те изменения, которые произойдут при откреплении дисплея LCDPanel от основной формы MainForm. Это не так просто, как может показаться на первый взгляд. Во-первых, вероятно, стоит создать обработчик события MainForm->OnUnDock. Однако он будет бесполезен из-за внутренней ошибки библиотеки VCL, которая приводит к тому, что событие OnUnDock не возникает при первом откреплении элемента управления. Если потребуется изменить размер такого элемента управления, то очевидно, что оно будет выполнено совсем не так, как требуется. Лучше всего в такой ситуации создать обработчик события OnDockEnd и проверить значение свойства Floating дисплея. Если его значение равно true и это первый вызов события OnDockEnd, то элемент управления будет откреплен. Это событие происходит одновременно с событием OnUnDock, поэтому пользователь не заметит никакой разницы. Единственное дополнительное требование заключается в том, что придется использовать отдельную переменную для обозначения первого вызова события OnEndDock при выполнении открепления. Дело в том, что событие OnEndDock вызывается в конце каждого перемещения открепленного элемента управления. Для этого в рассматриваемом примере используем логическую переменную FirstLCDPanelEndDock. Добавим строку bool FirstLCDPanelEndDock; в определение класса основной формы MainForm и инициализируем ее значением true в конструкторе основной формы MainForm так, как показано ниже.

```
FirstLCDPanelEndDock = true;
```

Необходимый код обработчика события OnEndDock дисплея LCDPanel показан в листинге 5.20.

Листинг 5.20. Реализация обработчика события LCDPanel->OnEndDock

```
void __fastcall TMainForm::LCDPanelEndDock(TObject *Sender,
                                           TObject *Target,
                                           int X,
                                           int Y)
{
   if(LCDPanel->Floating)
   {
      SetFocus();
   }
   if(FirstLCDPanelEndDock)
   {
      if(LCDPanel->Floating) FirstLCDPanelEndDock = false;
      Height = Height - LCDPanel->Height;
   }
}
```

Если событие OnEndDock дисплея LCDPanel возникло впервые в текущей последовательности откреплений (т.е., дисплей LCDPanel только что откреплен, и значение переменной FirstLCDPanelEndDock равно true), то следует изменить размер основной формы MainForm, вычитая высоту Height дисплея LCDPanel из текущей высоты Height основной формы

MainForm. Это действие выполняется всегда, даже если дисплей еще не отсоединился от основной формы, поскольку высота Height дисплея LCDPanel складывается снова с высотой основной формы MainForm в обработчике события OnDockDrop основной формы MainForm, которое возникает при присоединении дисплея LCDPanel. Это происходит при каждой первой попытке откреплення дисплея LCDPanel, когда возможно открепить и закрепить дисплей LCDPanel с помощью одного действия.

При этом не потребуется перемещать панель ButtonsControlBar и строку состояния StatusBar1 основной формы MainForm, потому что для их свойства Align заданы значения alClient и alBottom, соответственно. Теперь дисплей LCDPanel можно открепить, а основная форма MainForm автоматически изменит свои размеры. Обратите внимание. что до изменения размеров основной формы MainForm сначала необходимо задать значение false для переменной FirstLCDPanelEndDock, но только в том случае, когда дисплей LCDPanel отсоединен (LCDPanel->Floating == true). Это объясняется тем, что при первой попытке откреплення дисплея можно открепить и снова закрепить его с помощью одного действия. При этом дисплей LCDPanel может быть не отсоединен, а значение false для переменной FirstLCDPanelUnDock будет означать, что этот код не будет выполнен при следующем откреплении.

Обратите внимание, что при каждом вызове функции LCDPanelEndDock() и в том случае, когда переменная LCDPanel->Floating равна true, вызывается функция SetFocus() основной формы MainForm. Это позволяет сохранить фокус ввода с помощью клавиатуры в основной форме MainForm. Более подробно эта тема уже рассматривалась в разделе о повышении практичности за счет управления фокусом ввода.

Закрепление дисплея LCDPanel на основной форме MainForm организовано немного сложнее, чем открепление. Во-первых, нужно реализовать обработчик события OnGetSiteInfo основной формы MainForm. Это событие передает по ссылке параметр InfluenceRect типа TRect. Переменная типа TRect указывает место в форме, где будет активировано закрепление элемента управления при его размещении над этим местом. Это позволяет указать области закрепления одних элементов управления на других. В программе MiniCalculator область закрепления определяется высотой Height дисплея LCDPanel и шириной ClientWidth основной формы MainForm, с началом в верхней левой точке основной формы. Код этого обработчика события показан в листинге 5.21.

Листинг 5.21. Обработчик события MainForm->OnGetSiteInfo

```
void __fastcall TMainForm::FormGetSiteInfo(TObject* Sender,
                                           TControl* DockClient,
                                           TRect& InfluenceRect,
                                           TPoint& MousePos,
                                           bool& CanDock)
{
   if(DockClient->Name == "LCDPanel")
   {
      InfluenceRect.Left = ClientOrigin.x;
      InfluenceRect.Top = ClientOrigin.y;
      InfluenceRect.Right = ClientOrigin.x+ClientWidth;
      InfluenceRect.Bottom = ClientOrigin.y+DockClient->Height;
   }
}
```

Во-первых, внутри обработчика события FormGetSiteInfo() необходимо проверить, указывает ли DockClient (указатель на элемент управления TControl, который вызвал появление этого события) на дисплей LCDPanel. Если это так, то определяется место закрепления, в котором дисплей LCDPanel может быть опущен согласно указанным значениям параметра InfluenceRect. При этом не используются остальные параметры MousePos и CanDock. Параметр MousePos является ссылкой на текущее положение курсора, а параметр CanDock используется для определения возможности закрепления. Если для параметра CanDock задано значение false, то область DockClient не может выполнять закрепление.

Теперь нужно создать код обработчика события OnDockOver основной формы MainForm. Это событие позволяет обеспечить визуальную обратную связь с пользователем для указания места закрепления элемента управления, если этот элемент управления находится над областью закрепления (например, указатель мыши находится в области InfluenceRect) и является закрепляемым (CanDock == true). Значение свойства Source->DockRect, т.е. указатель TDragDropObject, используется для определения области закрепления, которая предстает перед пользователем в виде прямоугольника. В листинге 5.22 показан код обработчика события MainForm->OnDockOver.

Листинг 5.22. Код обработчика события MainForm->OnDockOver

```
void __fastcall TMainForm::FormDockOver(TObject* Sender,
                                        TDragDockObject* Source,
                                        int X,
                                        int Y,
                                        TDragState State,
                                        bool& Accept)
{
   if(Source->Control->Name == "LCDPanel")
   {
      TRect DockingRect(ClientOrigin.x,
                        ClientOrigin.y,
                        ClientOrigin.x + ClientWidth,
                        ClientOrigin.y +
                           Source->Control->Height );
      Source->DockRect = DockingRect;
   }
}
```

При размещении закрепляемого элемента управления над областью InfluenceRect (как определено в OnGetSiteInfo) на экране над областью Source->DockRect, определенной в коде обработчика события OnDockOver, появятся прямоугольные контуры, обозначающие положение элемента управления. Это дает пользователю визуальное представление о месте закрепления элемента управления при отпускании кнопки мыши. В этом случае значение Source->DockRect задается равным высоте Height элемента управления и ширине ClientWidth основной формы, с началом прямоугольной области в области ClientOrigin. Фактически эта область равна области InfluenceRect, указанной в OnGetsiteInfo.

Остальные параметры, т.е. горизонтальное положение указателя мыши X; вертикальное положение указателя мыши Y; состояние мыши State типа TDragState по отношению к элементу управления; а также параметр Accept при этом не используются. Значение false параметра Accept предотвращает закрепление элемента управления.

Наконец, нужно создать код обработчика события MainForm->OnDockDrop. Это событие позволят изменить размер элемента управления для подгонки к размеру области DockRect, указанной в коде обработчика события OnDockOver. Благодаря этому можно выполнять любую необходимую обработку, например изменять размеры формы или свойство Anchors или Align. Код обработчика события MainForm->OnDockDrop показан в листинге 5.23.

Листинг 5.23. Код обработчика события MainForm->OnDockDrop

```
void __fastcall TMainForm::FormDockDrop(TObject* Sender,
                                        TDragDockObject* Source,
                                        int X,
                                        int Y)
{
   if(Source->Control->Name == "LCDPanel")
   {
      Source->Control->Top = 0;
      Source->Control->Left = 0;
      Source->Control->Width = ClientWidth;

      // Предоставить место ...
      Height = Height + Source->Control->Height;

      // Изменение значения свойства Align дисплея LCDPanel
      Source->Control->Align = alTop;

      // Изменение значения флага FirstLCDPanelEndDock
      FirstLCDPanelEndDock = true;
   }
}
```

Показанный в листинге 5.23 код обработчика события FormDockDrop() не так уж прост, как это может показаться на первый взгляд. Во-первых, следует изменить размер дисплея LCDPanel для подгонки его по верхнему краю формы. Затем нужно предоставить место для закрепляемой панели за счет увеличения высоты Height основной формы MainForm на величину высоты Height дисплея LCDPanel. Поскольку для свойства Align при откреплении дисплея LCDPanel задается значение alNone, для свойства LCDPanel->Align нужно задать значение alTop. Для свойства FirstLCDPanelEndDock нужно задать значение true, чтобы приготовиться к следующему откреплению дисплея LCDPanel.

Здесь следует сделать два замечания. Во-первых, нужно настроить высоту Height основной формы MainForm *до* установки значения alTop свойства Align дисплея LCDPanel. Если значение alTop задать для свойства LCDPanel->Align до настройки высоты Height основной формы MainForm, то придется дважды настраивать высоту Height основной формы MainForm. Дело в том, что размер основной формы MainForm будет автоматически изменен для подгонки дисплея LCDPanel, если для свойства LCDPanel->Align задано значение alTop и в форме недостаточно места для размещения дисплея. Соответственно, изменение высоты Height основной формы MainForm вручную приведет к удвоению заданной высоты. Изменение сначала свойства Height основной формы MainForm позволяет решить эту проблему, потому что для размещения дисплея LCDPanel будет предоставлено достаточно места. После присвоения значения alTop свойству Align уже не потребуется автоматическое изменение размеров.

Во-вторых, не потребуются перестановки других элементов управления в основной форме MainForm, потому что их на ней только два, причем у обоих свойство Align имеет значение, отличное от alNone. Свойство ButtonsControlBar->Align панели кнопок ButtonsControlBar (типа TControlBar) имеет значение alClient, а свойство StatusBar1->Align строки состояния StatusBar1 (типа TStatusBar) имеет значение alBottom. После предоставления дополнительной высоты в форме эти элементы управления будут перемещены автоматически.

Хотя возможности открепления и закрепления элементов управления в программе MiniCalculator несколько ограничены, но их вполне достаточно. Более подробный пример использования возможностей открепления и закрепления в C++Builder содержится в проекте dockex.bpr, который находится в каталоге $(BCB)\Examples\Docking основного каталога C++Builder 5.

Изменение размеров

В зависимости от контекста и поставленной задачи, для изменения размеров можно использовать события OnResize и OnConstrainedResize. В программе MiniCalculator используются оба эти события: при изменении размеров дисплея LCDPanel нужно обновить выравнивание по обоим краям текста в надписях с помощью события OnResize, а при изменении размеров панели ButtonsControlBar нужно с помощью события OnConstrainedResize не допустить ее уменьшения до размера, меньше суммарного размера ее кнопок. Управление размерами также контролируется с помощью указания значений для свойств Align, Anchors, AutoSize, Constraints, Height, Left, Top и Width элементов управления. Эти свойства, за исключением свойства AutoSize, также рассматриваются в программе MiniCalculator. По умолчанию свойство AutoSize имеет значение false. Указание для него значения true вызывает автоматическое изменение размеров элемента управления для подгонки расположения его содержимого.

Свойство Align

Свойство Align используется для создания областей, которые изменяют свой размер в соответствии с особыми правилами. Для создания областей с изменяемыми размерами рядом с областями с постоянными размерами следует использовать комбинацию вложенных панелей и соответствующим образом задавать значения свойства Align для создания желаемого эффекта. Существует неограниченное количество возможных структур, и чтобы создать нужную структуру, придется немного поэкспериментировать. Комбинируя использование выравниваемых компонентов панели с помощью тщательно заданных ограничений Constraints, можно создать интерфейс с очень точным изменением его размеров.

Проект Panels.bpr, который имеется на прилагаемом к книге компакт-диске, содержит пример использования компонентов панели для создания интерфейса с переменными размерами. Он содержит панели со свойствами Align, которые имеют разные значения и обладают разными качествами при изменении размеров формы Form1. Ограничения Contraints для панели Panel1 и Panel2 заданы такими, что позволяют повысить стабильность интерфейса, предотвращая чрезмерное уменьшение интерфейса.

В программе MiniCalculator интерфейс разбит на три основных области, содержащие элементы управления. В верхней части формы располагается панель дисплея LCDPanel со значением alTop свойства Align. Там же находится панель кнопок ButtonsControlBar со значением alClient свойства Align, а в нижней части формы располагается панель строки состояния

StatusBar1 со значением alBottom свойства Align. Для ограничений минимальной (MinHeight) и максимальной (MaxHeight) высоты дисплея LCDPanel и строки состояния StatusBar1 заданы значения их фактической высоты Height. Поэтому высота дисплея не может быть изменена. На рис. 5.5 показан внешний вид интерфейса калькулятора с указанием этих параметров.

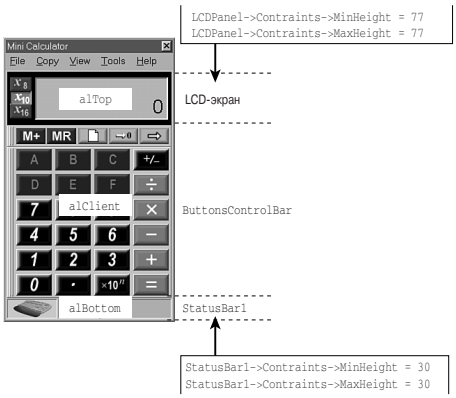


Рис. 5.5. Значения свойства Align для элементов управления интерфейса калькулятора MiniCalculator

При изменении размеров калькулятора MiniCalculator изменяются только ширина дисплея LCDPanel и строки состояния StatusBar1, тогда как у панели кнопок ButtonsControlBar могут меняться и высота, и ширина.

Свойство Anchors

Свойство Anchors (привязки) аналогично свойству Align, но обладает дополнительными степенями свободы. Привязки очень важны для правильного размещения дисплея LCDPanel в верхней части калькулятора MiniCalculator. Дисплей LCDPanel создан с помощью двух панелей TPanel, трех надписей TLabel и трех кнопок TSpeedButton. Все они располагаются в панели дисплея LCDPanel типа TPanel. Ограничение минимальной ширины Constraints->MinWidth дисплея LCDPanel содержит значение 227 пикселей, а ограничения минимальной Constraints->MinHeight и максимальной Constraints->MaxHeight высоты заданы равными 77 пикселям (как показано на рис. 5.5). Высота дисплея не может быть изменена, а ширина не может быть меньше 227 пикселей.

Еще одна панель для фона дисплея BackgroundDisplay типа TPanel располагается в верхней части дисплея LCDPanel с помощью привязок akLeft, akRight, akTop и akBottom. Это гарантирует изменение размеров панели BackgroundPanel при изменении размеров дисплея LCDPanel, причем с сохранением размеров границы. Так как высота дисплея LCDPanel не меняется во время выполнения приложения, привязки akTop и akBottom в данном случае не потребуются, но их включение позволяет упростить изменение размеров высоты дисплея LCDPanel при создании приложения.

На рис. 5.6 показаны привязки, используемые для трех надписей TLabel фоновой панели BackgroundPanel. Этот рисунок будет использован при обсуждении методов работы с указанными надписями.

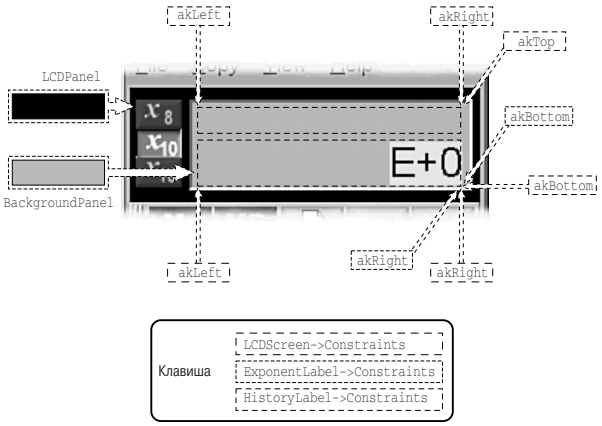


Рис. 5.6. Привязки, используемые для трех надписей `TLabel` дисплея `LCDPanel`

Надпись `LCDScreen` привязан к фоновой панели `BackgroundPanel` привязками `akLeft`, `akRight` и `akBottom`. Можно использовать привязку `akTop`, но не хотелось бы менять высоту дисплея `LCDScreen` во время создания приложения при изменении высоты панели `BackgroundPanel` или панели `LCDPanel`. Для панели `LCDScreen` привязка `akBottom` также необязательна во время выполнения приложения, но ее очень удобно использовать при создании приложения. Если высота панели `LCDPanel` изменяется во время создания приложения, то соответственно будет изменяться высота панели `BackgroundPanel`. Высота дисплея `LCDScreen` меняться не будет, но он будет перемещаться для сохранения заданного промежутка до нижней части панели `BackgroundPanel`. Привязки `akLeft` и `akRight` гарантируют, что левые и правые края во время создания и выполнения приложения будут одинаковы.

Надпись `HistoryLabel` привязана к фоновой панели `BackgroundPanel` практически так же, как и дисплей `LCDScreen`. Ее свойство `Anchors` содержит привязки `akLeft` и `akRight`, но вместо привязки `akBottom` используется привязка `akTop`. Как и в случае дисплея `LCDScreen`, нужны только привязки `akLeft` и `akRight`, а привязка `akBottom` лишь упрощает манипулирование надписью `HistoryLabel` при создании приложения. Если высота дисплея `LCDPanel` изменяется во время создания приложения, то расстояние между надписью `HistoryLabel` и краем фоновой панели `BackgroundPanel` будет оставаться постоянным. Следовательно, будет изменяться только промежуток между надписью `HistoryPanel` и дисплеем `LCDScreen`.

Наконец, надпись `ExponentLabel` привязан к фоновой панели `BackgroundPanel` с помощью привязок `akRight` и `akBottom`. Расположение этих привязок такое же, как и положение привязок `akRight` и `akBottom` дисплея `LCDScreen`. Кроме того, как и для дисплея `LCDScreen`, привязка `akBottom` не нужна во время выполнения программы, а включена только для упрощения работы с этой надписью при создании приложения.

Следует отметить, что положение привязки можно изменять во время выполнения приложения без необходимости повторной привязки к этому месту. Это происходит с дисплеем `LCDScreen` при редактировании степеней. При нажатии кнопки `SpeedButtonExponent` надпись `LCDScreen` должен быть сжат и удален из надписи `ExponentLabel`. Это делается таким образом, что при отображении надписи `ExponentLabel` пользователь может видеть ее одновременно с дисплеем `LCDScreen`. Код перемещения и изменения размеров дисплея `LCDScreen` для видимой надписи `ExponentLabel` выглядит следующим образом.

```
int LCDScreenLeft = LCDScreen->Left;
LCDScreen->Width = LCDScreen->Width - ExponentLabel->Width;
LCDScreen->Left = LCDScreenLeft;
```

Изменяя ширину дисплея LCDScreen, необходимо соответствующим образом переместить левую или правую привязку. Выбор перемещаемой привязки определяется значением свойства Alignment дисплея LCDScreen. Если это значение равно taRightJustify, то переместить нужно левую привязку, если taLeftJustify — правую, а если taCenter — обе. Обращаясь к предыдущему фрагменту кода, можно сделать вывод, что в данном случае при изменении ширины дисплея LCDScreen, свойство Alignment которого имеет значение taRightJustify, придется переместить левую привязку. Так как правая привязка остается неизменной, то придется уменьшить ширину дисплея LCDScreen без перемещения ее за пределы надписи ExponentLabel, которая стала видимым. Настройка свойства Left дисплея LCDScreen происходит таким образом, что оно имеет прежнее значение, что позволяет при желании сдвинуть дисплей LCDScreen влево. Существует альтернативный подход, который заключается в изменении выравнивания дисплея LCDScreen по левому краю taLeftJustify до изменения ширины Width дисплея LCDScreen. После изменения этой ширины нужно просто вернуть свойству Alignment прежнее значение taRightJustify. Приведенный ниже код аналогичен приведенному ранее фрагменту кода.

```
LCDScreen->Alignment = taLeftJustify;
LCDScreen->Width = LCDScreen->Width - ExponentLabel->Width;
LCDScreen->Alignment = taRightJustify;
```

Свойство Anchors — очень удобный инструмент для управления макетом интерфейса. Наиболее часто кнопки привязок используются для диалоговых окон с изменяемыми размерами. В проекте Panels.bpr, который находится на прилагаемом к книге компакт-диске, приведен пример такой работы с двумя кнопками TBitBtn в красной (clRed) панели Panel5. При изменении размеров формы эти кнопки располагаются на неизменном расстоянии от правого края формы. Аналогичный эффект может быть достигнут за счет размещения кнопок непосредственно в форме на заданном расстоянии от правого края с указанием правой привязки для каждой кнопки (akRight == true).

Ограничения Constraints

Ограничения Constraints представляют собой очень эффективный механизм ограничения размеров элемента управления. При работе с ними важно помнить, что нельзя изменять размеры элемента управления с нарушением его ограничений или ограничений его видимого элемента управления.

В проекте Panels.bpr приводится пример использования ограничений Constraints для предотвращения чрезмерного уменьшения основной формы. Для свойства Constraints->MinHeight панелей Panel1 и Panel2 задано одинаковое значение 300. Это значит, что клиентская часть формы Form1 (основной формы) не может быть меньше 300×300. Для минимальной высоты MinHeight и ширины MinWidth формы Form1 можно задать другие ограничения, но для достижения такого же результата (300×300) нужно учесть разницу между высотой формы Height и клиентской части ClientHeight, а также шириной формы Width и клиентской части ClientWidth.

В MiniCalculator также используются ограничения Constraints. Чтобы гарантировать видимость основного меню для основной формы MainForm, задано 52 пикселя в качестве минимальной высоты Constraints->MinHeight и 248 пикселей в качестве минимальной ширины Constraints->MinWidth. Это также гарантирует видимость строки состояния StatusBar1, так как заданы только ограничения ее минимальной MinHeight и максимальной высоты MaxHeight

(оба значения равны 30, т.е. высоте Height строки состояния StatusBar1). Дисплей LCDPanel имеет собственное ограничение минимальной ширины MinWidth, так как он может открепляться/закрепляться и его размер можно менять независимо от основной формы MainForm. Для ограничений минимальной MinHeight и максимальной высоты MaxHeight дисплея LCDPanel задано значение 77, т.е. значение высоты Height дисплея LCDPanel. Следовательно, высота дисплея LCDPanel не может быть изменена. Панель кнопок ButtonsControlBar обладает собственными ограничениями, которые определяются динамически во время выполнения приложения, так что они всегда могут быть подогнаны под размеры элементов управления, которые она содержит, независимо от их положения в ней. Для этого потребуется создать соответствующий код события OnConstrainedResize панели кнопок ButtonsControlBar.

Событие OnConstrainedResize

Событие OnConstrainedResize используется для обновления ограничений элемента управления при изменении его размеров. Как уже было сказано, в программе MiniCalculator динамически обновляются ограничения панели кнопок ButtonsControlBar типа TControlBar. Это делается для ограничения минимальной высоты и ширины панели кнопок и постоянной подгонки ее размеров в соответствии с содержащимися в ней элементами управления, независимо от их расположения внутри панели. Код обработчика события ButtonsControlBar->OnConstrainedResize приводится в листинге 5.24.

Листинг 5.24. Код обработчика события ButtonsControlBar->OnConstrainedResize

```
void __fastcall TMainForm::ButtonsControlBarConstrainedResize(
                                      TObject* Sender,
                                      int& MinWidth,
                                      int& MinHeight,
                                      int& MaxWidth,
                                      int& MaxHeight)
{
   GetControlBarMinWidthAndHeight(ButtonsControlBar,
                                  MinWidth, MinHeight);
}
```

Событие OnConstrainedResize имеет четыре параметра, каждый из которых соответствует одному ограничению Constraints элемента управления, вызвавшего появление этого события: MinWidth, MinHeight, MaxWidth и MaxHeight.

Обработчик события ButtonsControlBarConstrainedResize() выполняет ограниченное количество действий: вызывает функцию с исчерпывающим названием GetControlBarMinWidthAndHeight(), передавая в качестве аргументов указатель на данную панель кнопок, а также параметры MinWidth и MinHeight для вычисления и присвоения им новых значений. Объявление функции GetControlBarMinWidthAndHeight() имеет следующий вид.

```
void __fastcall GetControlBarMinWidthAndHeight(
                               TCustomControlBar* ControlBar,
                               int& MinWidth,
                               int& MinHeight);
```

Оба параметра типа int передаются функции GetControlBarMinWidthAndHeight() по ссылке, что позволяет изменять переданные значения. Код функции GetControlBarMinWidthAndHeight() приведен в листинге 5.25.

Листинг 5.25. Код функции `GetControlBarMinWidthAndHeight()`

```
void __fastcall
TMainForm::GetControlBarMinWidthAndHeight(
                          TCustomControlBar* ControlBar,
                          int& MinWidth,
                          int& MinHeight)
{
   int MinLeft = 0;
   int MinTop = 0;
   int MaxRight = 0;
   int MaxBottom = 0;

   bool FirstVisible = true;

   for(int i=0; i<ControlBar->ControlCount; ++i)
   {
      if(ControlBar->Controls[i]->Visible)
      {
         if(FirstVisible)
         {
            MinLeft   = ControlBar->Controls[i]->Left - 11;
            MinTop    = ControlBar->Controls[i]->Top - 2;
            MaxRight  = ControlBar->Controls[i]->Left
                        + ControlBar->Controls[i]->Width + 2;
            MaxBottom = ControlBar->Controls[i]->Top
                        + ControlBar->Controls[i]->Height + 2;
            FirstVisible = false;
         }
         else
         {
            if((ControlBar->Controls[i]->Left-11) < MinLeft)
            {
               MinLeft = ControlBar->Controls[i]->Left-11;
            }
            if((ControlBar->Controls[i]->Top-2) < MinTop)
            {
               MinTop = ControlBar->Controls[i]->Top-2;
            }
            if((ControlBar->Controls[i]->Left
                + ControlBar->Controls[i]->Width
                + 2) > MaxRight)
            {
               MaxRight = ControlBar->Controls[i]->Left
                          + ControlBar->Controls[i]->Width + 2;
            }
            if((ControlBar->Controls[i]->Top
                + ControlBar->Controls[i]->Height
                + 2) > MaxBottom)
            {
```

```
                MaxBottom = ControlBar->Controls[i]->Top
                            + ControlBar->Controls[i]->Height
                            + 2;
             }
          }
       }
   }
   MinWidth = (MaxRight - MinLeft);
   MinHeight = (MaxBottom - MinTop);
}
```

Выполняемые функцией `GetControlBarMinWidthAndHeight()` действия очень просты. Для каждого видимого элемента управления панели кнопок вычисляется значение левого, правого, верхнего и нижнего краев, включая их границу. Граница включает маркер, который используется для перемещения элемента управления внутри панели. В этом примере граница имеет ширину 11 пикселей слева и ширину 2 пикселя с других сторон. Каждое из этих значений сравнивается со значениями `MinLeft`, `MaxRight`, `MinTop` и `MaxBottom` и обновляется соответствующим новым значением. Наконец, вычисляются нужные значения ограничений `MinWidth` и `MinHeight`.

```
MinWidth = (MaxRight - MinLeft);
MinHeight = (MaxBottom - MinTop);
```

Теперь они являются новыми значениями ограничений `MinWidth` и `MinHeight` для панели `ButtonsControlBar`.

Событие `OnResize`

Событие `OnResize` используется для обновления внешнего вида элемента управления или выполнения других действий, связанных с изменением размеров элемента управления. Событие `OnResize` вызывается после изменения размеров элемента управления. Его не следует использовать для обновления значений свойства `Constraints`. Вместо него для этого рекомендуется использовать событие `OnConstrainedResize`, потому что события `OnResize` и `OnConstrainedResize` вызываются в такой последовательности: сначала `OnResize`, а потом `OnConstrainedResize` [затем `OnResize`]. Если значения свойства `Constraints` изменены в свойстве `OnConstrainedResize`, то вызывается также событие `OnResize`, которое позволяет выполнить дополнительное изменение размеров. При обновлении ограничений `Constraints` с помощью события `OnResize` этот код будет выполнен дважды.

В программе MiniCalculator событие `OnResize` дисплея `LCDPanel` используется для правильного выравнивания содержащихся в ней надписей. Код обработчика события `OnResize` приведен в листинге 5.26.

Листинг 5.26. Код обработчика события `LCDPanel->OnResize`

```
void __fastcall TMainForm::LCDPanelResize(TObject *Sender)
{
   UpdateLCDScreen(LCDScreen->Caption);
   UpdateHistoryLabel(HistoryLabel->Caption);
   if(LCDPanel->Floating && MainForm->Visible) SetFocus();
}
```

Обработчик события OnResize выполняет три действия: обновляет надпись дисплея LCDScreen, затем обновляет надпись HistoryLabel и, наконец, переводит фокус ввода в основную форму, если эта панель является плавающей (Floating), т.е. откреплена от основной формы. Учтите, что функция SetFocus() вызывается, только если основная форма видима. Такая проверка нужна потому, что событие OnResize может произойти еще до отображения основной формы, например если размер панели изменяется для удовлетворения какого-то предыдущего параметра интерфейса. Более подробно этот вопрос рассматривается ниже в разделе о повышении практичности за счет запоминания предпочтений пользователя.

При вызове функции UpdateLCDScreen() ей в качестве единственного параметра передается текущая надпись LCDScreen->Caption. Важно, что при этом передается именно текущая надпись Caption и проверяется необходимость подгонки его отображения в дисплее LCDPanel нового размера. Код этой функции UpdateLCDScreen() уже приводился в листинге 5.11, но для удобства читателя он повторяется в листинге 5.27.

Листинг 5.27. Код функции UpdateLCDScreen()

```
void __fastcall TMainForm::UpdateLCDScreen(
                           const AnsiString& NewNumber)
{
   int NumberWidth = LCDScreen->Canvas->TextWidth(NewNumber);

   if(Operation == coComplete)
   {
      if(   (NumberWidth >= LCDScreen->Width)
         && (LCDScreen->Alignment == taRightJustify))
      {
         LCDScreen->Alignment = taLeftJustify;
      }
      else if(   (NumberWidth < LCDScreen->Width)
              && (LCDScreen->Alignment != taRightJustify))
      {
         LCDScreen->Alignment = taRightJustify;
      }
   }
   else if(LCDScreen->Alignment != taRightJustify)
   {
      LCDScreen->Alignment = taRightJustify;
   }

   LCDScreen->Caption = NewNumber;

   int pos = LCDScreen->Hint.Pos("|");
   int length = LCDScreen->Hint.Length();
   AnsiString LCDScreenHint
              = LCDScreen->Hint.SubString(pos, length-pos+1);
   LCDScreen->Hint = NewNumber + LCDScreenHint;

   if(NumberWidth >= LCDScreen->Width) LCDScreen->ShowHint = true;
   else LCDScreen->ShowHint = false;
}
```

Действия функции UpdateLCDScreen() можно разбить на три этапа.

1. Проверка размера (в пикселях), необходимого для отображения строки NewNumber. (В этом случае строка NewNumber фактически является текущим значением свойства Caption дисплея LCDScreen). Для этого используется метод TextWidth() канвы TCanvas.
2. Указание правильного выравнивания текста (свойство Alignment дисплея LCDScreen) в зависимости от того, завершена ли операция. Если операция только что завершена, свойство Operation получает значение coComplete, а дисплей LCDScreen недостаточно широк для отображения всего числа целиком, то его надпись выравнивается по левому краю (LCDScreen>Alignment = taLeftJustify). Операция считается завершенной, когда отображается ее результат, а значит, в этом случае гораздо важнее увидеть начало полученного числа, а не его конец. Если в дисплее достаточно места для отображения этого числа, то применяется выравнивание текста по правому краю дисплея LCDScreen, так как оно является предпочтительным типом выравнивания (LCDScreen>Alignment = taRightJustify). В противном случае, т.е. если операция не завершена, дисплей LCDScreen отображает значение, которое редактируется или может редактироваться в данный момент. Для редактируемого числа применяется выравнивание по правому краю (LCDScreen>Alignment = taRightJustify), чтобы всегда была видна редактируемая часть числа.
3. Наконец, если число NewNumber очень велико для полного отображения в дисплее LCDScreen, то в таком случае строка NewNumber передается подсказке Hint дисплея LCDScreen. Этот вопрос более подробно рассматривается в предыдущем разделе об использовании настраиваемых пользовательских подсказок.

После вызова функции UpdateLCDScreen() вызывается функция UpdateHistoryLabel(), которая выполняет аналогичную задачу, т.е. подбирает режим выравнивания в надписи HistoryLabel в соответствии с шириной текста HistoryLabel->Caption. Однако, в отличие от UpdateLCDScreen(), функция UpdateHistoryLabel() не выполняет подгонку размера надписи HistoryLabel. В результате функция UpdateHistoryLabel() имеет гораздо более простой код, который представлен в листинге 5.28.

Листинг 5.28. Код функции UpdateHistoryScreen()

```
void __fastcall TMainForm::UpdateHistoryLabel(
                                const AnsiString& NewHistory)
{
   int HistoryWidth =
       HistoryLabel->Canvas->TextWidth(NewHistory);

   if(   (HistoryWidth >= HistoryLabel->Width)
      && (HistoryLabel->Alignment == taLeftJustify))
   {
      HistoryLabel->Alignment = taRightJustify;
   }
   else if(   (HistoryWidth < HistoryLabel->Width)
           && (HistoryLabel->Alignment != taLeftJustify))
   {
      HistoryLabel->Alignment = taLeftJustify;
   }

   HistoryLabel->Caption = NewHistory;
}
```

В функции UpdateLCDScreen() сначала применяется функция TextWidth() для определения ширины (в пикселях) строки NewHistory типа AnsiString. Если строка NewHistory очень длинна для полного отображения в надписи HistoryLabel, то следует выровнять текст по правому краю надписи HistoryLabel, чтобы на экране был виден самый последний результат. В противном случае текст следует выровнять по левому краю надписи HistoryLabel. Это приведет к созданию эффекта прокрутки текста налево после достижения правого края надписи HistoryLabel.

Наконец, в конце обработчика события OnResize выполняется проверка откреплённого состояния дисплея LCDPanel от основной формы MainForm. Если это так, то основная форма MainForm теряет фокус ввода, так как изменение размеров дисплея LCDPanel переведет его в дисплей. Следовательно, для возврата фокуса ввода в основную форму потребуется вызвать функцию SetFocus(). Это гарантирует возобновление обратной связи со стороны основной формы MainForm с операциями ввода с помощью клавиатуры.

Панель кнопок TControlBar

Панель кнопок TControlBar обычно используется как визуальный контейнер типа TToolBar. На самом деле она может использоваться как визуальный контейнер и для других элементов управления, а не только для панелей инструментов. Причем элементы управления, содержащиеся в панели типа TControlBar, могут даже иметь разные типы. Программа MiniCalculator использует панель типа TControlBar как визуальный контейнер для компонентов типа TPanel. В этом примере (см. рис. 5.5) используется панель ButtonsControlBar типа TControlBar. При создании программы необходимо изменить некоторые свойства панели ButtonsControlBar.

- DockSite = false

  Элементы управления в панели типа TControlBar мы не собираемся создавать откреплёнными, причем дисплей LCDPanel также не нужно прикреплять к этой панели.

- RowSize = 31

  Высота панели должна быть кратна 31, т.е. высоте ее строк. Значение RowSnap также равно true, что гарантирует выравнивание элементов управления в строках шириной 31 пиксель.

- AutoDrag = false

  Строки этой панели должны быть неоткреплёнными при перетаскивании их с панели. Наоборот, нужно чтобы они оставались в панели.

- AutoDock = false

  Не нужно прикреплять к этой панели никакие другие элементы, поэтому следует отключить автоматическое закрепление.

Для использования кнопочной панели поместим в нее панели, которые будут использоваться как строки клавиш калькулятора. Все остальное кнопочная панель выполнит самостоятельно. Она добавит маркер строки в левой части панели (две вертикальные линии, которые используются для перетаскивания элемента управления панели) и рамку вокруг нее. Для выполнения этой задачи может потребоваться слегка изменить расположение некоторых компонентов.

По мере создания интерфейса MiniCalculator часто приходится перемещать кнопки в другие элементы интерфейса. Поочередное выполнение каждого такого действия представляет собой очень рутинную и связанную с возможностью появления ошибок задачу. Однако при этом нельзя выбрать сразу несколько кнопок с помощью создания вокруг них прямоугольного контура без выбора того элемента управления, в котором они находятся. Тем не менее есть один простой способ выбора большой группы элементов управления без выбора того элемента управления, в котором они находятся. Для этого нужно просто открыть форму в текстовом формате (щелкнув на ней правой кнопкой мыши и выбрав команду View as Text из контекстного меню). Далее нужно просто вырезать и вставлять нужные фрагменты текстового описания элементов управления.

Использование кнопочной панели для управления расположением панелей в MiniCalculator значительно упрощает используемый для интерфейса код, а также позволяет настроить интерфейс согласно предпочтениям пользователей. В текущем состоянии панель `ButtonsControlBar` позволяет добиться гораздо большей гибкости в работе, но придется приложить дополнительные усилия для надежного перемещения строк панели, например, чтобы они не перекрывались при этом. Однако при работе с ними имеется некоторая степень свободы. В программе MiniCalculator предусмотрена возможность выравнивания отдельных или всех элементов панели `ButtonsControlBar` по левому или по правому краям. Кроме того, предусмотрено уменьшение размера этой панели до суммарного размера всех ее элементов управления. Эта функция доступна благодаря контекстному меню со следующими командами.

1. Snap to Fit

   Размер панели `ButtonsControlBar` уменьшается до суммарного размера всех ее элементов управления.

2. Align Left

   Строка панели под текущим указателем мыши в момент вызова контекстного меню выравнивается по левому краю панели `ButtonsControlBar`.

3. Align Right

   Строка панели под текущим указателем мыши в момент вызова контекстного меню выравнивается по правому краю панели `ButtonsControlBar`.

4. Align All Left

   Все строки панели `ButtonsControlBar` выравниваются по левому краю.

5. Align All Right

   Все строки панели `ButtonsControlBar` выравниваются по правому краю.

Команды 1, 4 и 5 доступны всегда, а команды 2 и 3 — только если контекстное меню `ControlBarPopupMenu` открывается над одним из элементов управления панели `ButtonsControlBar`.

Подгонка размера панели `TControlBar` под размеры ее содержимого

Рассмотрим код, используемый для подгонки размера панели `TControlBar` под размеры ее содержимого. При выборе команды меню `SnapToFit1` из контекстного меню `ControlBarPopupMenu`, выполняется ее обработчик события `OnClick`. Обработчик вызывает вспомогательную функцию `FitToControlBar()`, которая имеет следующее объявление.

```
void __fastcall FitToControlBar(TCustomControlBar* ControlBar);
```

Единственный параметр этой функции содержит указатель на панель, размер которой нужно подогнать под размеры ее содержимого. Код функции `FitToControlBar()`представлен в листинге 5.29.

Листинг 5.29. Код функции `FitToControlBar()`

```
void __fastcall TMainForm::FitToControlBar(
                           TCustomControlBar* ControlBar)
{
   int MinWidth = 0;
   int MinHeight = 0;

   GetControlBarMinWidthAndHeight(ControlBar, MinWidth, MinHeight);

   int WidthDifference = ButtonsControlBar->Width - MinWidth;
   int HeightDifference = ButtonsControlBar->Height - MinHeight;

   Width = Width - WidthDifference;
   Height = Height - HeightDifference;
}
```

Для изменения размера панели сначала вызывается функция `GetControlBarMinWidthAndHeight()`, код которой показан в листинге 5.25. Это позволяет получить минимальные значения высоты и ширины панели. Затем эти значения используются в вычислениях разницы между текущими размерами и размерами, которые нужно задать для этой панели. Потом эти значения вычитаются из значений ширины `Width` и высоты `Height` основной формы. Эти операции вычитания выполняются для ширины/высоты основной формы, а не панели `ButtonsControlBar`, потому что свойство выравнивания `Align` панели `ButtonsControlBar` имеет значение `alClient`. В этом случае размеры панели `ButtonsControlBar` будут автоматически изменены для подгонки к размерам формы.

Выравнивание элементов управления панели `TControlBar`

В программе MiniCalculator предусмотрена возможность выравнивания отдельных или сразу всех элементов управления в панели `TControlBar`. Элементы управления могут выравниваться по левому или по правому краю. Как уже говорилось, для выбора типа выравнивания используется контекстное меню `ControlBarPopupMenu`. Перечень доступных пользователю функций (команд контекстного меню) зависит от места в панели `ButtonsControlBar`, где вызвано контекстное меню `ControlBarPopupMenu`. Если оно вызвано над одним из элементов управления панели `ButtonsControlBar`, то пользователь сможет выровнять расположение как одного этого элемента, так и всех сразу. А если контекстное меню `ControlBarPopupMenu` панели `ButtonsControlBar` вызвано за пределами элементов управления панели, то пользователь сможет выровнять расположение только сразу всех элементов панели.

Для определения места появления контекстного меню `ControlBarPopupMenu` в панели `ButtonsControlBar` нужно создать соответствующий код обработчика события `OnContextPopup` панели `ButtonsControlBar`. При этом предполагается, что для свойства `PopupMenu` панели `ButtonsControlBar` задано значение `ControlBarPopupMenu`. Код обработчика события `ButtonsControlBar->OnContextPopup` панели `ButtonsControlBar` приведен в листинге 5.30.

Листинг 5.30. Код обработчика события `ButtonsControlBar->OnContextPopup`

```
void __fastcall TMainForm::ButtonsControlBarContextPopup(
                                        TObject *Sender,
                                        TPoint &MousePos,
                                        bool &Handled)
{
   // Где в панели Control Bar произошел щелчок правой кнопкой
   // мыши, т.е. над каким элементом управления?
   TRect*ControlRects =
      new TRect[ButtonsControlBar->ControlCount];

   try
   {
      for(int i=0; i<ButtonsControlBar->ControlCount; ++i)
      {
         if(ButtonsControlBar->Controls[i]->Visible)
         {
            ControlRects[i] =
               ButtonsControlBar->Controls[i]->BoundsRect;
            ControlRects[i].Left -= 11;
            ControlRects[i].Top -= 2;
            ControlRects[i].Right += 2;
            ControlRects[i].Bottom += 2;
         }
         else
         {
            ControlRects[i] = TRect(0,0,0,0);
            // Без прямоугольника
         }
      }

      for(int i=0; i<ButtonsControlBar->ControlCount; ++i)
      {
         if(PtInRect(&ControlRects[i], MousePos))
         {
            AlignLeft1->Visible = true;
            AlignRight1->Visible = true;
            ControlBarPopupMenu->Tag
               = ButtonsControlBar->Controls[i]->Tag;
            break;
         }
         else
         {
            AlignLeft1->Visible = false;
            AlignRight1->Visible = false;
            ControlBarPopupMenu->Tag = cbpNone;
         }
      }
   }
   __finally
```

```
   {
      delete [] ControlRects;
   }
}
```

Для определения места расположения указателя мыши при вызове контекстного меню над одним из видимых элементов управления панели ButtonsControlBar мы используем массив значений типа TRect, в котором хранятся границы каждого элемента управления. Если элемент управления не виден, то прямоугольная область TRect не имеет размера, так что указатель мыши не может находится в области TRect этого элемента управления при вызове контекстного меню. Для определения области TRect, занятой элементом управления панели ButtonsControlBar, считывается значение свойства BoundsRect этого элемента управления, а затем оно изменяется для включения маркера строки (11 пикселей с левой стороны) и рамки (по 2 пикселя с других сторон).

Для определения координат указателя мыши последовательно перебираются прямоугольные области TRect всех элементов управления с использованием WinAPI-функции PtInRect(). Если указатель найден, следует задать для свойства Tag контекстного меню ControlBarPopupMenu значение метки, того элемента управления, над которым находится указатель мыши. Свойство Tag используется для хранения информации о том элементе управления, над которым находился указатель мыши в момент вызова контекстного меню. Эта информация используется для определения элемента управления, который следует выровнять с помощью обработчика события OnClick контекстного меню ControlBarPopupMenu, если пользователь предпочтет выровнять только один элемент управления. Кроме того, если указатель мыши в момент вызова контекстного меню находился в элементе управления, то команды меню AlignLeft1 и AlignRight1 для выравнивания отдельных элементов управления следует сделать видимыми.

Обратите внимание на конструкцию try/_finally, которая используется для защиты ресурсов, т.е. для удаления массива ControlRects, даже если будет сгенерирована исключительная ситуация.

Рассмотрим более подробно использование свойства Tag для элементов управления в панели ButtonsControlBar для обозначения каждого элемента управления. В конструкторе основной формы MainForm метке каждого компонента кнопочной панели присваивается некоторое значение. При перечислении панелей, которые содержатся в кнопочной панели ButtonsControlBar, для большей читабельности используем конструкцию TControlBarPanel перечислимого типа enum, как показано ниже.

```
enum TControlBarPanel { cbpFunctionButtons,
                        cbpNumberButtons,
                        cbpNone };
```

В конструкторе основной формы MainForm запишем следующие строки.

```
FunctionButtonsPanel->Tag = cbpFunctionButtons;
NumberButtonsPanel->Tag = cbpNumberButtons;
```

Если нужно добавить в кнопочную панель ButtonsControlBar большее количество панелей, то необходимо просто вставить нужное количество элементов в конструкцию TControlBarPanel для новых панелей и присвоить их соответствующим меткам Tag в конструкторе основной формы MainForm. Например, при добавлении дополнительной панели ConstantsButtonsPanel следовало бы вставить элемент cbpConstantsButtons в конструкцию TControlBarPanel и добавить следующие строки в конструкторе основной формы MainForm.

```
ConstantsButtonsPanel->Tag = cbpConstantsButtons;
```

Теперь новая панель готова к работе.

Рассмотрим способы выравнивания элементов управления в кнопочной панели ButtonsControlBar. В панели типа TControlBar нет никаких свойств, которые управляли бы выравниванием содержащихся в ней строк, поэтому такое выравнивание придется выполнить вручную. Принципы подобного выравнивания не очень просты, а потому и соответствующий код будет довольно сложным. Для иллюстрации основных принципов рассмотрим в качестве примера элемент управления Panel.

Для выравнивания элементов управления по левому краю кнопочной панели зададим значение 11 для свойства Left того элемента управления, который нужно выровнять. Это позволяет учесть маркер строки с левой стороны строки, в которой находится элемент управления. При этом строка будет сдвинута в левую часть кнопочной панели. Для этого используется показанная ниже строка кода.

```
Panel->Left = 11;
```

При изменении размера кнопочной панели элемент управления останется сдвинутым в левую часть кнопочной панели. На самом деле этот тип выравнивания используется по умолчанию в панели типа TControlBar, что упрощает выравнивание по левому краю.

Код выравнивания по правому краю выглядит несколько сложнее. Если нужно просто выровнять элемент управления по текущей правой стороне кнопочной панели, то следует задать такое значение для свойства Left этого элемента управления, чтобы его правый край сдвинулся к текущей правой стороне кнопочной панели. Для этого нужно создать следующую строку кода.

```
Panel->Left = ButtonsControlBar->ClientWidth - Panel->Width - 2;
```

Если размер кнопочной панели уменьшается, то данный элемент управления остается выровненным по правому краю, а если увеличивается — располагается в соответствии со значением свойства Left. Для принудительного выравнивания элемента управления по правому краю нужно задать для его свойства Left значение больше, чем максимально возможная ширина кнопочной панели. Это достигается указанием ширины экрана в качестве значения свойства Left.

```
Panel->Left = Screen->Width;
```

Теперь после изменения размеров кнопочной панели элемент управления останется выровненным по правому краю кнопочной панели.

В листинге 5.31 представлен код обработчика события AlignLeft1Click() для команды контекстного меню, которая приводит к выравниванию одного элемента управления по левому краю кнопочной панели.

Листинг 5.31. Код обработчика события AlignLeft1Click()

```
void __fastcall TMainForm::AlignLeft1Click(TObject *Sender)
{
   ArrangeControlBarBands(ButtonsControlBar,
                          TControlBarPanel(ControlBarPopupMenu->Tag),
                          cbaLeft);
}
```

Обработчик события AlignLeft1Click() для фактического перемещения элемента управления вызывает вспомогательную функцию ArrangeControlBarBands(). В качестве первого аргумента ей передается указатель на кнопочную панель. Второй аргумент содержит

метку Tag выравниваемого элемента управления. Наконец, третий аргумент cbaLeft обозначает выравнивание данного элемента управления по левому краю. Объявление функции ArrangeControlBarBands() имеет следующий вид.

```
void __fastcall ArrangeControlBarBands(
                              TCustomControlBar* ControlBar,
                              TControlBarPanel CurrentBandTag,
                              TControlBarAlignment Alignment);
```

Конструкция TControlBarAlignment типа enum объявляется, как показано ниже.

```
enum TControlBarAlignment { cbaLeft,
                            cbaRight,
                            cbaAllLeft,
                            cbaAllRight };
```

Она используется функцией ArrangeControlBarBands() для определения необходимого типа выравнивания. Код функции ArrangeControlBarBands() показан в листинге 5.32 и, как видите, он достаточно сложен.

Листинг 5.32. Код функции ArrangeControlBarBands()

```
void __fastcall
TMainForm::ArrangeControlBarBands(
                        TControlBar* ControlBar,
                        TControlBarPanel CurrentBandTag,
                        TControlBarAlignment Alignment)
{
   // Этап 1
   std::list<TControlBandInfo>BandList;

   TControlBandInfo ControlBarBand;

   // Этап 2
   for(int i=0; i<ControlBar->ControlCount; ++i)
   {
      if(ControlBar->Controls[i]->Tag == CurrentBandTag)
      {
         ControlBarBand =
                  TControlBandInfo(ControlBar->Controls[i],
                  ControlBar->Controls[i]->Left,
                  ControlBar->Controls[i]->Top,
                  ControlBar->Controls[i]->Height + 2,
                  ControlBar->Controls[i]->Visible);
      }
      BandList.push_back(TControlBandInfo(
                        ControlBar->Controls[i],
                        ControlBar->Controls[i]->Left,
                        ControlBar->Controls[i]->Top,
                        ControlBar->Controls[i]->Height + 2,
                        ControlBar->Controls[i]->Visible));
   }
```

```
// Этап 3
if(Alignment == cbaLeft)
{
   // То же, что и BandList.sort();
   BandList.sort(std::less<TControlBandInfo>());
   ControlBarBand.Left = 11;
}
else if(Alignment == cbaRight)
{
   BandList.sort(std::greater<TControlBandInfo>());
   ControlBarBand.Left = Screen->Width;
}

// Этап 4
std::list<TControlBandInfo>::iterator pos;
bool NoFreeColumn = false;

for(pos = BandList.begin(); pos != BandList.end(); ++pos)
{
   // Этап 5
   if(    CurrentBandTag != pos->Control->Tag
       && pos->Visible )
   {
      // Этап 6
      if(    (Alignment == cbaLeft)
          && (pos->Left < (ControlBarBand.Control->Width +
                           2)))
      {
         NoFreeColumn = true;
      }
      if(    (Alignment == cbaRight)
          && (   (ControlBar->ClientWidth -
                   (pos->Left+pos->Control->Width+2))
               < (ControlBarBand.Control->Width +2) ) )
      {
         NoFreeColumn = true;
      }

      // Этап 7
      if(    ControlBarBand.Top >= pos->Top
          && ControlBarBand.Top < (pos->Top + pos->Height
                                   + 2) )
      {
         // Нет места для сдвига в следующий пустой столбец
         if(NoFreeColumn)ControlBarBand.Top =
                                 pos->Top + pos->Height + 2;
         else break;
      }
      // Этап 8
      else if( ControlBarBand.Top < pos->Top )
```

```
      {
         // Есть место
         std::list<TControlBandInfo>::iterator pos2;
         int Offset = 0;
         bool FirstVisibleControl = true;

         // Этап 9
         for(pos2 = pos; pos2 != BandList.end(); ++pos2)
         {
            if(    pos2->Visible
                && FirstVisibleControl
                && pos2->Control->Tag != CurrentBandTag)
            {
               // Первый элемент управления
               Offset = 2
                        + ControlBarBand.Top
                        + ControlBarBand.Height - pos2->Top;

               FirstVisibleControl = false;
            }
            if(pos2->Visible)pos2->Top = pos2->Top + Offset;
         }
         break;
      }
      NoFreeColumn = false;
   }
}

// Этап 10
for(pos = BandList.begin(); pos != BandList.end(); ++pos)
{
   pos->Control->Visible = false;
   if(pos->Control->Tag == CurrentBandTag)
   {
      pos->Left = ControlBarBand.Left;
      pos->Top = ControlBarBand.Top;
   }
}

// Этап 11
if(Alignment == cbaLeft)
{
   BandList.sort(std::less<TControlBandInfo>());
}
else if(Alignment == cbaRight)
{
   BandList.sort(std::greater<TControlBandInfo>());
}

// Этап 12
```

```
   for(pos = BandList.begin(); pos != BandList.end(); ++pos)
   {
      pos->Control->Top = pos->Top;
      pos->Control->Left = pos->Left;
      pos->Control->Visible = pos->Visible;
   }
}
```

Метод `ArrangeControlBarBands()` использует класс `TControlBandInfo`, определение которого приводится в листинге 5.33. В этом листинге особо следует отметить операторы `<` и `>`. Они применяются для сортировки списка объектов типа `TControlBandInfo`. Оба оператора сортируют полоски элементов управления сверху вниз, при одинаковом верхнем значении: оператор `<` сортирует слева направо, а оператор `>` — справа налево, т.е. для выравнивания по левому и по правому краю, соответственно. В остальном структура этого класса достаточно проста и позволяет использовать его с классами-контейнерами стандартной библиотеки C++ Standard Library. Более подробную информацию об этом можно найти в разделе, посвященном стандартной библиотеке C++ Standard Library, главы 4.

Возвращаясь к листингу 5.32, можно подытожить действия, выполняемые функцией `ArrangeControlBarBands()`. Перечисленные ниже пункты относятся к этапам, которые указаны в комментариях кода этой функции. Например, пункт 1 относится к комментарию `// Этап 1`.

1. Создание с помощью конструкции `list<>` стандартной библиотеки C++ Standard Library списка объектов `BandList` типа `TControlBandInfo` и одной переменной `ControlBarBand` типа `TControlBandInfo`. Она будет использоваться для хранения информации о текущей выравниваемой полоске. Для использования конструкции `list<>` необходимо включить директиву `#include <list>` в файл с исходным кодом.
2. Затем последовательно перебираются элементы управления кнопочной панели `ButtonsControlBar`, и информация `TControlBandInfo` о них добавляется в список `BandList` вместе с соответствующей информацией о каждой полоске. При этом выполняется проверка, не нужно ли выровнять данный элемент управления. Если нужно, то дополнительная копия информации о нем передается переменной `ControlBarBand`.
3. Список с информацией `TControlBandInfo` сортируется, в зависимости от типа выравнивания. Для выравнивания по левому краю используется оператор `<(sdt::less<TControlBandInfo>())`, а для выравнивания по правому краю — оператор `>(sdt::greater<TControlBandInfo>())`. Функции `less<>` и `greater<>` стандартной библиотеки C++ Standard Library применяются для указания члену-функции `sort()` списка `list<>` той функции, которую нужно выполнить. На самом деле, это не придется делать, если использовать оператор `<`, потому что этот оператор по умолчанию используется членом-функцией `sort()`. На этом этапе также необходимо задать нужное значение для свойства `Left` кнопочной панели `ControlBarBand` в соответствии с заданным типом выравнивания. Для использования функций `less<>` и `greater<>` необходимо включить директиву `#include <functional>` в файл с исходным кодом.
4. Создается итератор для последовательного перебора элементов списка с объектами `TControlBandInfo`. Более подробная информация об итераторах приведена в разделе, посвященном стандартной библиотеке C++ Standard Library, главы 4.
5. Проверим, не является ли данный элемент управления кнопочной панелью `ControlBarBand`, а также видимости кнопочной панели `ControlBarBand`.

6. Затем на основе заданного выравнивания убедимся в отсутствии свободного места с соответствующей стороны текущего элемента управления (`NoFreeColumn = true`). Это имеет большое значение, если текущий элемент управления занимает ту же строку или те же строки, что и кнопочная панель `ControlBarBand`.
7. Если выравниваемый элемент управления занимает одну строку с другим элементом управления и за ним нет свободного места (`NoFreeColumn = true`), то его нужно переместить в следующую строку. А если свободное место есть, поиск и работа цикла прекращаются.
8. В противном случае, т.е. если найдено свободное пространство, для свойства `Top` кнопочной панели `ControlBarBand` задается положение свободного места.
9. Затем последовательно перебираются остальные элементы управления, которые располагаются ниже текущего элемента управления, и их свойство `Top` изменяется на величину, необходимую для размещения текущего элемента управления.
10. Теперь, вычислив новые положения полосок, нужно последовательно перебрать их и задать значение `false` для свойства `Visible`. Когда достигнут выравниваемый элемент управления, эти значения копируются из кнопочной панели `ControlBarBand` в эквивалентный ему элемент списка. Элементы управления указаны как скрытые, чтобы можно было переместить их в новое положение в кнопочной панели `ButtonsControlBar`, произвольным образом меняя их значения.
11. После сохранения координат нового положения элементов кнопочной панели в списке `BandList` еще раз отсортируем список, чтобы самый верхний элемент был размещен в нем первым.
12. Наконец, еще раз выполним последовательный перебор элементов управления, присваивая новое значение для свойств `Top` и `Left`, а также возвращая им прежний видимый статус.

Несмотря на эти пояснения, код функции `ArrangeControlBarBands()` по-прежнему сложен для восприятия, и его можно улучшить. Кнопочная панель не разбита на столбцы, поэтому элементы управления могут перемещаться в ней без крайней необходимости. Тем не менее этот код демонстрирует основные принципы ручного управления выравниванием элементов.

Листинг 5.33. Определение класса `TControlBandInfo`

```
class TControlBandInfo
{
public:
   TControl*Control;
   int Left;
   int Top;
   int Height;
   bool Visible;

   // Конструктор
   inline __fastcall TControlBandInfo() : Control(0),
                                          Left(0),
                                          Top(0),
                                          Height(0),
                                          Visible(false)
   {}
```

```
// Конструктор
inline __fastcall TControlBandInfo(TControl* control,
                                   int left,
                                   int top,
                                   int height,
                                   bool visible)
                 : Control(control),
                   Left(left),
                   Top(top),
                   Height(height),
                   Visible(visible)
{}

// Конструктор копии
inline __fastcall TControlBandInfo(
                 const TControlBandInfo& ControlBandInfo)
         : Control(ControlBandInfo.Control),
           Left(ControlBandInfo.Left),
           Top(ControlBandInfo.Top),
           Height(ControlBandInfo.Height),
           Visible(ControlBandInfo.Visible)
{}

// Оператор =
TControlBandInfo& operator=(
                 const TControlBandInfo& ControlBandInfo)
{
   Control = ControlBandInfo.Control;
   Left = ControlBandInfo.Left;
   Top = ControlBandInfo.Top;
   Height = ControlBandInfo.Height;
   Visible = ControlBandInfo.Visible;
   return *this;
}

// Оператор ==
bool operator==(
     const TControlBandInfo& ControlBandInfo) const
{
   if(    Control == ControlBandInfo.Control
       && Left == ControlBandInfo.Left
       && Top == ControlBandInfo.Top
       && Height ==ControlBandInfo.Height
       && Visible ==ControlBandInfo.Visible)
   {
      return true;
   }
   else return false;
}
```

```
   // Оператор <
   bool operator<(const TControlBandInfo&ControlBandInfo)const
   {
      if(Top < ControlBandInfo.Top) return true;
      else if(    Top == ControlBandInfo.Top
                && Left < ControlBandInfo.Left) return true;
      else return false;
   }

   // Оператор >
   bool operator>(const TControlBandInfo& ControlBandInfo) const
   {
      if(Top < ControlBandInfo.Top) return true;
      else if(    Top == ControlBandInfo.Top
                && Left > ControlBandInfo.Left) return true;
      else return false;
   }
};
```

Единственное отличие между выравниванием одного и всех элементов управления заключается в том, что для выравнивания всех элементов кнопочной панели нужно вызвать функцию `ArrangeControlBarBands()`. Для сравнения в листинге 5.34 показан код обработчика события `OnClick` для выравнивания всех элементов кнопочной панели по правому краю. Обратите внимание, что элементы управления упорядочены, т.е. создается и сортируется список с объектами `TControlBandInfo`. Затем из отсортированного списка извлекаются значения типа `TControl*` для получения метки `Tag` и использования ее в вызове функции `ArrangeControlBarBands()`. Каждый элемент управления затем выравнивается по правому краю. Наконец, после сортировки все элементы управления снова еще раз последовательно перебираются, и их свойства `Left` получают значение `Screen->Width`, чтобы гарантировать выравнивание по правому краю при изменении размеров кнопочной панели.

Листинг 5.34. Код обработчика события `AlignAllRight1Click()`

```
void __fastcall TMainForm::AlignAllRight1Click(TObject *Sender)
{
   std::list<TControlBandInfo> BandList;
   for(int i=0; i<ButtonsControlBar->ControlCount; ++i)
   {
      BandList.push_back(
          TControlBandInfo(
                  ButtonsControlBar->Controls[i],
                  ButtonsControlBar->Controls[i]->Left,
                  ButtonsControlBar->Controls[i]->Top,
                  ButtonsControlBar->Controls[i]->Height + 2,
                  ButtonsControlBar->Controls[i]->Visible));
   }
   BandList.sort(std::greater<TControlBandInfo>());
```

```
   std::list<TControlBandInfo>::iterator pos;
   for(pos = BandList.begin(); pos != BandList.end(); ++pos)
   {
      ArrangeControlBarBands(ButtonsControlBar,
                             TControlBarPanel(pos->Control->Tag),
                             cbaRight);
   }

   BandList.sort(std::greater<TControlBandInfo>());
   for(pos = BandList.begin(); pos != BandList.end(); ++pos)
   {
      pos->Control->Left = Screen->Width;
   }
}
```

Использование описанных выше методов гарантирует достаточно предсказуемое выравнивание, т.е. размещение элемента управления в новом месте не приведет к замене существующего элемента управления, если он использует ту же строку. Однако существует несколько других причин возникновения достаточно противоречивых ситуаций. Тем не менее такое поведение позволяет адекватно управлять внешним видом интерфейса.

Управление видимостью

Возможность отображать или скрывать отдельные части интерфейса представляет собой относительно простой способ управления внешним видом приложения со стороны пользователей. Изменяя значение свойства `Visible` для данного элемента управления, можно отображать или скрывать его в интерфейсе. Это позволяет предоставить пользователю возможность выбора дополнительных функций приложения, которые одни пользователи сочтут нужными, а другие — нет. В таком случае нужные функции будут определены как видимые, а ненужные — как скрытые. Основным условием при этом является приемлемый внешний вид интерфейса после отображения/скрытия выбранных функций. Иначе говоря, после скрытия элемента управления в интерфейсе не должны оставаться большие пустые области, а отображение элемента управления не должно искажать текущий макет интерфейса.

В программе MiniCalculator предусмотрена возможность скрытия/отображения любого элемента кнопочной панели `ButtonsControlBar`, а также основного дисплея (`LCDPanel`) или строки состояния (`StatusBar1`). Это выполняется с помощью команд меню View программы. Эта команда меню уже рассматривалась в предыдущем разделе об использовании символов наряду с текстом. Здесь мы рассмотрим конкретные действия, выполняемые командами меню View. При вызове и отображении на экране команд меню View элементы управления с изменяемой видимостью последовательно проверяются обработчиком события `View1Click()` щелчка мыши на команде меню View (объект `View1`). Это нужно для определения текущих видимых и скрытых элементов. Код обработчика события `View1Click()` достаточно прост; в этом легко можно убедиться, обратившись к листингу 5.35.

Листинг 5.35. Код обработчика события `View1Click()`

```
void __fastcall TMainForm::View1Click(TObject *Sender)
{
   if(LCDPanel->Visible) ViewDisplay->Checked = true;
```

```
   else ViewDisplay->Checked = false;

   if(NumberButtonsPanel->Visible)
      ViewNumberButtons->Checked = true;
   else ViewNumberButtons->Checked = false;

   if(FunctionButtonsPanel->Visible)
      ViewFunctionButtons->Checked = true;
   else ViewFunctionButtons->Checked = false;

   if(StatusBar1->Visible) ViewStatusBar->Checked = true;
   else ViewStatusBar->Checked = false;
}
```

Необходимость скрытия дисплея `LCDPanel` вызывает сомнение, но его повторное отображение все же может потребоваться, поскольку он может временно скрываться при откреплении от основной формы и перемещении. Следовательно, для дисплея потребуется создать отдельную команду меню. После щелчка на команде меню View⇨Display дисплей `LCDPanel` станет видимым или скрытым, в зависимости от его текущей видимости. В листинге 5.36 приведен код обработчика события `ViewDisplayClick()` для выбора команды меню View⇨Display.

Листинг 5.36. Код обработчика события `ViewDisplayClick()`

```
void __fastcall TMainForm::ViewDisplayClick(TObject *Sender)
{
   if(ViewDisplay->Checked)
   {
      LCDPanel->Visible = false;
      ViewDisplay->Checked = false;
      if(!LCDPanel->Floating)
      {
         // Установка размера формы ...
         Height = Height - LCDPanel->Height;
      }
   }
   else
   {
      if(!LCDPanel->Floating)
      {
         // Установка размера формы ...
         Height = Height + LCDPanel->Height;
      }

      LCDPanel->Visible = true;
      ViewDisplay->Checked = true;

      if(LCDPanel->Floating)
      {
         SetFocus();
```

```
      }

      // Расположение строки состояния StatusBar внизу
      if(StatusBar1->Visible) StatusBar1->Align = alBottom;
   }
}
```

По сравнению с другими элементами управления изменить статус видимости для дисплея `LCDPanel` сложнее всего, потому что дисплей может быть откреплен от основной формы. Тем не менее сделать это достаточно просто.

Если дисплей `LCDPanel` видим (`ViewDisplay->Checked == true`), то его нужно скрыть (`LCDPanel->Visible = false`) и снять флажок команды меню View⇨Display (`ViewDisplay->Checked = false`). Затем, если дисплей `LCDPanel` закреплен в основной форме (`!LCDPanel->Floating`), нужно настроить высоту основной формы `MainForm` с учетом скрытия дисплея.

Если дисплей `LCDPanel` скрыт, то сначала нужно проверить, прикреплен ли он к основной форме (`LCDPanel->Floating == false`). Если это так, нужно настроить высоту основной формы `MainForm` таким образом, чтобы включить в нее дисплей. Затем дисплей следует сделать видимым и установить флажок команды меню View⇨Display. Если дисплей `LCDPanel` был откреплен от основной формы `MainForm`, то после его отображения на экране в него перейдет фокус ввода. В таком случае необходимо перевести фокус ввода в основную форму с помощью функции `SetFocus()`. Наконец, если строка состояния `StatusBar1` видима, ее нужно разместить в нижней части основной формы `MainForm`.

Изменить статус видимости строки состояния `StatusBar1` гораздо проще. Единственное, что нужно учесть, — изменение высоты основной формы `MainForm`. Как это делается, показано в листинге 5.37.

Листинг 5.37. Код обработчика события `ViewStatusBarClick()`

```
void __fastcall TMainForm::ViewStatusBarClick(TObject *Sender)
{
   if(ViewStatusBar->Checked)
   {
      StatusBar1->Visible = false;
      ViewStatusBar->Checked = false;
      Height = Height - StatusBar1->Height;
   }
   else
   {
      Height = Height + StatusBar1->Height;
      StatusBar1->Visible = true;
      ViewStatusBar->Checked = true;
   }
}
```

Наконец, рассмотрим как организовано изменение статуса видимости полосок внутри кнопочной панели `ButtonsControlBar`. Код этого действия одинаков для всех полосок, поэтому рассмотрим только одну полоску, которая используется для полоски `NumberButtonsPanel`. Соответствующий код показан в листинге 5.38.

Листинг 5.38. Код обработчика события `ViewNumberButtonsClick()`

```
void __fastcall TMainForm::ViewNumberButtonsClick(
                                          TObject *Sender)
{
   if(ViewNumberButtons->Checked)
   {
      NumberButtonsPanel->Visible = false;
      ViewNumberButtons->Checked = false;
      if(AutoFit->Checked) FitToControlBar(ButtonsControlBar);
   }
   else
   {
      NumberButtonsPanel->Visible = true;
      ViewNumberButtons->Checked = true;
      // Установка строки состояния StatusBar внизу
      if(StatusBar1->Visible) StatusBar1->Align = alBottom;
   }
}
```

При скрытии полоски `NumberButtonsPanel` снимается флажок связанной с ней команды меню, и для свойства `NumberButtonsPanel->Visible` задается значение `false`. Затем, если установлен флажок команды меню Tools⇨Options⇨AutoFit, вызывается вспомогательная функция `FitToControlBar()` для подгонки размера кнопочной панели `ButtonsControlBar` под размеры оставшихся в ней видимых элементов управления. Если в ней совсем не осталось видимых элементов управления, кнопочная панель `ButtonsControlBar` также будет скрыта за счет указания значения `0` для свойства `Height`.

Для отображения строки `NumberButtonsPanel` нужно снова установить флажок соответствующей команды меню и задать значение `true` для свойства `NumberButtonsPanel->Visible`. При этом не потребуется изменять размеры кнопочной панели `ButtonsControlBar` — это будет сделано автоматически для подгонки под размеры вновь видимого элемента управления. Наконец, если строка состояния `StatusBar1` стала видимой, ее нужно расположить в нижней части основной формы `MainForm`.

Настройка клиентской части родительской формы в MDI-интерфейсе

Создание для пользователя возможности настраивать фон родительской формы в MDI-интерфейсе (обычно за счет добавления в него рисунка) представляет собой не такую простую задачу, как это может показаться на первый взгляд. Поэтому она заслуживает особого внимания. Для этого необходимо создать подкласс клиентского окна родительской формы. (Более подробная информация о подклассах собрана в разделе, посвященном использованию невизуальных компонентов для отклика на сообщения, посланные другим компонентам, главы 11.) Дело в том, что клиентское окно родительской формы является фоном дочерних окон в MDI-интерфейсе. Фон придется создавать в клиентском окне, а не в самой форме. Более дополнительная информация по этому поводу собрана в интерактивной справке по Win32 SDK в разделе “Frame, Client, and Child Windows”. Для доступа к клиентскому окну нужно использовать свойство формы `ClientHandle`. Для создания рисунка в клиентском окне следует ис-

пользовать сообщение WM_ERASEBKGND. Проект MDIProject.bpr на прилагаемом к книге компакт-диске содержит код, необходимый для отображения рисунка в виде фона родительской формы в MDI-интерфейсе. Такой рисунок может быть расположен по центру, покрыть форму как бы мозаикой или растянут по размерам формы. При изучении этого кода обратите внимание на следующие две особенности. Во-первых, рисунок создается во внеэкранном битовом образе, а затем отображается на экране в клиентском окне с помощью WinAPI-функции BitBlt() или StretchBlt(). Это позволяет избежать мерцания. Во-вторых, метод Draw() используется для создания рисунка в канве Canvas внеэкранного битового образа. Этот способ предпочтительнее, чем использование функции BitBlt(), поскольку в рассматриваемом примере требуется организовать работу с рисунками в формате JPEG. Класс TJPEGImage является производным от класса TGraphic, потому он реализует метод Draw(), но сам по себе класс TJPEGImage не имеет канвы Canvas и не может использоваться вместе с функцией BitBlt().

Повышение практичности благодаря запоминанию предпочтений пользователя

Простейший способ запоминания предпочтений пользователя заключается в сохранении их в реестре. C++Builder позволяет легко сохранять их в реестре и извлекать их оттуда с помощью двух специализированных классов VCL-библиотеки: TRegistry и TRegIniFile. Класс TRegistry очень бегло рассматривается в руководстве пользователя C++Builder, поэтому здесь он будет рассмотрен в очень краткой форме. В большей степени нас будет интересовать класс TRegIniFile. Класс TRegIniFile действительно очень упрощает работу с реестром, а потому рекомендуется использовать именно его, а не TRegistry. Он является наследником класса TRegistry и предлагает большее количество функций на более высоком уровне абстракции. Для доступа к классу TRegistry или TRegIniFile необходимо включить в код модуля следующую директиву.

```
#include <Registry.hpp>
```

Наиболее часто используемые свойства и методы класса TRegIniFile перечислены в табл. 5.9.

Таблица 5.9. Свойства и методы класса TRegIniFile

Свойства и методы	Описание
FileName	Только для чтения. При создании объекта TRegIniFile возвращает путь как строку типа AnsiString
RootKey	Указывает корневой раздел (root key) объекта TRegIniFile. По умолчанию это HKEY_CURRENT_USER. При желании его можно изменить
ReadBool()	Считывает логическое значение заданного ключа (key). Обычно используется значение по умолчанию, которое применяется при отсутствии данного ключа
ReadInteger()	Считывает значение типа int заданного ключа (key). Обычно используется значение по умолчанию, которое применяется при отсутствии данного ключа
ReadString()	Считывает значение типа AnsiString заданного ключа (key). Обычно используется значение по умолчанию, которое применяется при отсутствии данного ключа
WriteBool()	Записывает логическое значение для заданного ключа (key). Если это значение отсутствует, оно создается

Окончание табл. 5.9

Свойства и методы	Описание
`WriteInteger()`	Записывает значение типа `int` для заданного ключа (key). Если это значение отсутствует, оно создается
`WriteString()`	Записывает значение типа `AnsiString` для заданного ключа (key). Если это значение отсутствует, оно создается

Важно, что при считывании/записи отсутствующих в реестре сведений не возникает ошибки, поскольку вместо отсутствующей записи используется принимаемое по умолчанию значение. Например, при записи нужного значения в реестре создается нужный, но отсутствующий прежде ключ. Таким образом, класс `TRegIniFile` существенно упрощает работу с реестром.

Класс `TRegIniFile` используется в программе MiniCalculator для хранения и чтения предпочтений пользователя. Ключи реестра создаются в двух случаях. Если пользователь выбирает команду меню Tools⇨Save Current Configuration, то сохраняются все текущие параметры программы MiniCalculator. А если пользователь установил флажок команды меню Tools⇨Option⇨Auto Save Configuration, то информация о конфигурации программы MiniCalculator записывается в реестр при закрытии программы MiniCalculator. Код реализации первой возможности приведен в листинге 5.39.

Листинг 5.39. Код обработчика события `SaveCurrentLayout1Click()`

```
void __fastcall TMainForm::SaveCurrentLayout1Click(
                                              TObject *Sender)
{
   std::auto_ptr<TRegIniFile>
                Registry(new
                   TRegIniFile("Software\\MiniCalculator"));

   // Создание ключей для хранения параметров в реестре
   Registry->WriteBool("Options", "AutoSaveLayout",
                       AutoSaveLayout->Checked);
   Registry->WriteBool("Options", "AutoFit", AutoFit->Checked);

   // Теперь сохранить все параметры
   WriteSettingsToRegistry(Registry);
}
```

Сначала создается новый объект `TRegIniFile`. Конструкция `auto_ptr<>` стандартной библиотеки C++ Standard Library используется в этом случае для его удаления при выходе `auto_ptr<>` за пределы области действия, что бывает при возникновении исключительной ситуации. В конструкторе содержится информация о подчиненном ключе, к которому нужно предоставить доступ в реестре. Корневой ключ автоматически инициализируется значением `HKEY_CURRENT_USER`, а потому полный путь имеет следующий вид.

```
HKEY_CURRENT_USER\Software\MiniCalculator
```

Все последующие операции чтения и записи будут выполняться только в этом разделе реестра.

Затем для записи параметров используется функция `WriteBool()`. Первый параметр этой функции указывает ключ, второй — имя данных, а последний — собственно значение данных, как показано ниже.

```
Registry->WriteBool("Options", "AutoFit", AutoFit->Checked);
```

В результате в реестре будет обновлен или создан новый раздел.

```
HKEY_CURRENT_USER\Software\MiniCalculator\Options: Name =
                                          AutoFit | Data = ?
```

Если значение свойства `AutoFit->Checked` равно `true`, значение данных будет равно 1, в противном случае одно будет равно 0.

Затем для записи в реестр всех параметров программы MiniCalculator используется функция `WriteSettingsToRegistry()`. Код этой функции показан в листинге 5.40.

Листинг 5.40. Код функции `WriteSettingsToRegistry()`

```
void __fastcall
TMainForm::WriteSettingsToRegistry(
               const std::auto_ptr<TRegIniFile>& Registry)
{
   // Дисплей LCDPanel
   // - Цвет
   Registry->WriteInteger("Settings\\Display\\Color",
                          "SurroundColor",
                          LCDPanel->Color);

   Registry->WriteInteger("Settings\\Display\\Color",
                          "BackgroundColor",
                          BackgroundPanel->Color);

   Registry->WriteInteger("Settings\\Display\\Color",
                          "ExponentColor",
                          ExponentEditColor);

   // - Закрепление
   Registry->WriteBool("Settings\\Display",
                       "Floating",
                       LCDPanel->Floating);

   Registry->WriteInteger("Settings\\Display",
                          "UndockWidth",
                          LCDPanel->UndockWidth);

   Registry->WriteInteger("Settings\\Display",
                          "UndockHeight",
                          LCDPanel->UndockHeight);

   if(LCDPanel->Floating)
   {
      TRect UndockedRect;
      if(GetWindowRect(LCDPanel->HostDockSite->Handle,
                       &UndockedRect))
      {
         Registry->WriteInteger("Settings\\Display",
```

```
                              "UndockLeft",
                              UndockedRect.Left);

      Registry->WriteInteger("Settings\\Display",
                              "UndockTop",
                              UndockedRect.Top);

      Registry->WriteInteger("Settings\\Display",
                              "UndockRight",
                              UndockedRect.Right);

      Registry->WriteInteger("Settings\\Display",
                              "UndockBottom",
                              UndockedRect.Bottom);
   }
}

// Основная форма Main Form
Registry->WriteInteger("Settings\\Position", "MainFormTop",
                       Top);
Registry->WriteInteger("Settings\\Position", "MainFormLeft",
                       Left);
Registry->WriteInteger("Settings\\Size", "MainFormHeight",
                       Height);
Registry->WriteInteger("Settings\\Size", "MainFormWidth",
                       Width);

// Строка состояния
Registry->WriteBool("Settings\\StatusBar", "Visible",
                    StatusBar1->Visible);
Registry->WriteBool("Settings", "EnableKeyboard",
                    EnableKeyboardInput);

// Панель кнопок
for(int i=0; i<ButtonsControlBar->ControlCount; ++i)
{
   AnsiString ControlPath = "Settings\\ControlBar\\";
   ControlPath += ButtonsControlBar->Controls[i]->Name;

   Registry->WriteInteger(
                ControlPath,
                "Left",
                ButtonsControlBar->Controls[i]->Left);

   Registry->WriteInteger(
                ControlPath,
                "Top",
                ButtonsControlBar->Controls[i]->Top);

   Registry->WriteInteger(
```

```
                    ControlPath,
                    "Height",
                    ButtonsControlBar->Controls[i]->Height+2);

      Registry->WriteBool(
                    ControlPath,
                    "Visible",
                    ButtonsControlBar->Controls[i]->Visible);
   }
}
```

Код записи значений в реестр очень прост и не нуждается в дополнительных комментариях. Труднее всего решить, какие параметры следует сохранять в реестре. Особо следует отметить координаты дисплея `LCDPanel` в том случае, когда он откреплен от основной формы (`Floating`). Если дисплей `LCDPanel` откреплен от основной формы, он будет содержаться в объекте типа `FloatingDockSiteClass`. По умолчанию этот тип предполагает создание границы и строки заголовка вокруг откепленного элемента управления. Для точного отображения дисплея в той же позиции необходимо иметь информацию о границе места закрепления. Для доступа к границе места закрепления нужно считать значение свойства `HostDockSite`. Для этого нужно передать в качестве параметра дескриптор окна места закрепления WinAPI-функции `GetWindowRect()` вместе с адресом структуры `TRect`, которая содержит параметры границы в следующем виде.

```
TRect UndockedRect;
if(GetWindowRect(LCDPanel->HostDockSite->Handle, &UndockedRect))
{
... // для простоты код здесь не показан
}
```

В кнопочной панели `ButtonsControlBar` последовательно перебираются все элементы управления и записываются соответствующие значения в ключе с именем каждого элемента управления. Это позволяет различать их при последующем считывании этих параметров из реестра.

Как видите, запись параметров в реестр на самом деле организовать не так просто. Еще одна важная часть работы заключается в правильной обработке считанных из реестра параметров.

При запуске программы MiniCalculator и вызове конструктора основной формы `MainForm` каждый раз вызывается функция `ReadAllValuesFromRegistry()`, которая считывает все сохраненные в реестре параметры программы. Во время первого запуска программы в реестре еще нет никаких параметров, но для данного примера это не важно. Вместо них используются принимаемые по умолчанию значения. В листинге 5.41 показан код этой функции `ReadAllValuesFromRegistry()`.

Листинг 5.41. Код функции `ReadAllValuesFromRegistry()`

```
void __fastcall TMainForm::ReadAllValuesFromRegistry()
{
   // Считывание параметров из реестра
   // Если выполняется при первом запуске программы,
   // в реестре никаких параметров этой программы еще нет

   std::auto_ptr<TRegIniFile>Registry(
                  new TRegIniFile("SOFTWARE\\MiniCalculator"));
```

```
   // Считывание параметров автоматического сохранения
   AutoSaveLayout->Checked
    = Registry->ReadBool("Options", "AutoSaveLayout",
                         AutoSaveLayout->Checked);

   AutoFit->Checked
    = Registry->ReadBool("Options", "AutoFit",
                         AutoFit->Checked);

   // Считывание других параметров
   ReadSettingsFromRegistry (Registry);
}
```

Функция `ReadAllValuesFromRegistry()` содержит очень простой код. Сначала считываются параметры автоматического сохранения, а затем значения других параметров с помощью функции `ReadSettingsFromRegistry()`. Обратите внимание, что текущее значение каждого параметра применяется как значение по умолчанию (третий аргумент). Это гарантирует стабильность работы программы при отсутствии нужного параметра в реестре. Далее параметру присваивается его текущее значение, как показано в листинге 5.42, в котором приведен код функции `ReadSettingsFromRegistry()`.

Листинг 5.42. Код функции `ReadSettingsFromRegistry()`

```
void __fastcall
TMainForm::ReadSettingsFromRegistry(
               const std::auto_ptr<TRegIniFile>& Registry)
{
  // Дисплей LCDPanel
  // - Цвет
   LCDPanel->Color = Registry->ReadInteger(
                              "Settings\\Display\\Color",
                              "SurroundColor",
                              LCDPanel->Color);

   BackgroundPanel->Color =
      Registry->ReadInteger("Settings\\Display\\Color",
                            "BackgroundColor",
                            BackgroundPanel->Color);
   ExponentViewColor = BackgroundPanel->Color;
   ExponentEditColor =
      Registry->ReadInteger("Settings\\Display\\Color",
                            "ExponentColor",
                            ExponentEditColor);

   // - Закрепление
   LCDPanel->UndockWidth =
      Registry->ReadInteger("Settings\\Display",
                            "UndockWidth",
                            LCDPanel->UndockWidth);
```

```
if(LCDPanel->UndockWidth > Screen->Width)
{
   LCDPanel->UndockWidth = Screen->Width;
}

LCDPanel->UndockHeight =
   Registry->ReadInteger("Settings\\Display",
                         "UndockHeight",
                         LCDPanel->UndockHeight);

bool Floating = Registry->ReadBool("Settings\\Display",
                                   "Floating",
                                   LCDPanel->Floating);

if(Floating)
{
   int UndockLeft =
       Registry->ReadInteger("Settings\\Display",
                             "UndockLeft",
                             LCDPanel->Left);
   int UndockTop = Registry->ReadInteger("Settings\\Display",
                                         "UndockTop",
                                         LCDPanel->Top);
   TRect UndockedRect(UndockLeft,
                      UndockTop,
                      UndockLeft + LCDPanel->UndockWidth,
                      UndockTop + LCDPanel->UndockHeight);

   int UndockRight =
       Registry->ReadInteger("Settings\\Display",
                             "UndockRight",
                             UndockedRect.Right);

   int UndockBottom =
       Registry->ReadInteger("Settings\\Display",
                             "UndockBottom",
                             UndockedRect.Bottom);

   if(UndockRight > Screen->Width)
   {
      int Offset = UndockRight - Screen->Width;
      UndockedRect.Right -= Offset;
      UndockedRect.Left -= Offset;
      if(UndockedRect.Left < 0) UndockedRect.Left = 0;
   }
   if(UndockBottom > Screen->Height)
   {
      int Offset = UndockBottom - Screen->Height;
      UndockedRect.Bottom -= Offset;
      UndockedRect.Top -= Offset;
      if(UndockedRect.Top < 0)
```

```
      {
         Offset = 0 - UndockedRect.Top;
         UndockedRect.Top = 0;
         UndockedRect.Bottom += Offset;
      }
   }

   LCDPanel->ManualFloat(UndockedRect);
}

// Основная форма
int top, left, height, width;

top = Registry->ReadInteger("Settings\\Position",
                            "MainFormTop", Top);
left = Registry->ReadInteger("Settings\\Position",
                             "MainFormLeft", Left);
height = Registry->ReadInteger("Settings\\Size",
                               "MainFormHeight", Height);
width = Registry->ReadInteger("Settings\\Size",
                              "MainFormWidth", Width);

if(width > Screen->Width) width = Screen->Width; // опасно!
if(left+width > Screen->Width)
{
   left -= (left+width) - Screen->Width;
}
if(height > Screen->Height) height = Screen->Height;//опасно!
if(top+height > Screen->Height)
{
   top -= (top+height) - Screen->Height;
}

Top = top;
Left = left;
Height = height;
Width = width;

// Строка состояния
StatusBar1->Visible =
   Registry->ReadBool("Settings\\StatusBar",
                      "Visible",
                      StatusBar1->Visible);

//if(!StatusBar1->Visible) Height -= StatusBar1->Height;
EnableKeyboardInput =
   Registry->ReadBool("Settings",
                      "EnableKeyboard",
                      EnableKeyboardInput);

// Панель кнопок
```

```
std::list<TControlBandInfo> BandList;

for(int i=0; i<ButtonsControlBar->ControlCount; ++i)
{
   AnsiString ControlPath = "Settings\\ControlBar\\";
   ControlPath += ButtonsControlBar->Controls[i]->Name;

   int ControlLeft
        = Registry->ReadInteger(
            ControlPath,
            "Left",
            ButtonsControlBar->Controls[i]->Left);
   int ControlTop
        = Registry->ReadInteger(
            ControlPath,
            "Top",
            ButtonsControlBar->Controls[i]->Top);
   int ControlHeight
        = Registry->ReadInteger(
            ControlPath,
            "Height",
            ButtonsControlBar->Controls[i]->Height+2);
   bool ControlVisible
         = Registry->ReadBool(
            ControlPath,
            "Visible",
            ButtonsControlBar->Controls[i]->Visible);

BandList.push_back(
   TControlBandInfo(ButtonsControlBar->Controls[i],
                    ControlLeft,
                    ControlTop,
                    ControlHeight,
                    ControlVisible));
}
BandList.sort();

std::list<TControlBandInfo>::iterator pos;

for(pos = BandList.begin(); pos != BandList.end(); ++pos)
{
   pos->Control->Visible = false;
}

for(pos = BandList.begin(); pos != BandList.end(); ++pos)
{
   pos->Control->Top = pos->Top;
   pos->Control->Left = pos->Left;
   pos->Control->Visible = pos->Visible;
}
```

```
   // Учесть подгонку размеров, выполненную ButtonsControlBar
   Top = top;
   Left = left;
   Height = height;
   Width = width;
}
```

Как уже было сказано, запись параметров в реестр выполняется достаточно просто. Так же просто выполняется и чтение параметров из реестра. Как видно из листинга 5.42, труднее решить, что делать с этими параметрами. Большая часть листинга 5.42 достаточно понятна, но отдельные замечания необходимо сделать в отношении обработки данных, касающихся дисплея `LCDPanel` и кнопочной панели `ButtonsControlBar`.

Если дисплей `LCDPanel` откреплялся, то для открепления следует применить метод `ManualFloat()`. Для этого нужно использовать переменную типа `TRect`, содержащую координаты области, в которой дисплей будет перерисован. Эта информация уже сохранена в реестре в параметрах `UndockLeft`, `UndockTop`, `UndockRight` и `UndockBottom`. Кроме того, в реестре хранится информация о ширине и высоте откреплённого дисплея `LCDPanel`. Можно было бы, вероятно, использовать четыре координаты `Undock`*`XXX`* для создания подходящей области `TRect`, а затем использовать ее в вызове функции `ManualFloat`. Однако это неверно. Дело в том, что код функции `ManualFloat` воспринимает переданные ей координаты области `TRect` не так, как этого следовало бы ожидать. Свойства `Left` и `Top` представляют координаты левой и верхней точки плавающего откреплённого узла (а *не* дисплея `LCDPanel`), а свойства `Right` и `Bottom` — нет. Они используются только для вычисления высоты и ширины дисплея `LCDPanel`. Следовательно, нужно создать собственную структуру области `TRect`, как показано ниже.

```
TRect UndockedRect(UndockLeft,
                   UndockTop,
                   UndockLeft + LCDPanel->UndockWidth,
                   UndockTop + LCDPanel->UndockHeight);
```

Остальная часть кода обработки параметров дисплея `LCDPanel` связана с перемещением плавающего дисплея, если он выходит за рамки экрана при низком разрешении.

Также проблематичной является интерпретация данных реестра для кнопочной панели `ButtonsControlBar`. Нужно отсортировать все элементы управления в порядке возрастания значений свойства `Top`, а затем отобразить их в кнопочной панели. Сначала нужно нарисовать самые верхние элементы управления, потому что в объекте `TControlBar` предусмотрено автоматическое выравнивание элементов управления по верху или расположенному выше элементу управления. Если сначала расположить самые низкие элементы управления, то они будут перемещены наверх, что нежелательно. Код, используемый для сортировки элементов управления, аналогичен коду, который используется в методе `ArrangeControlBarBands()` в листинге 5.32 и описан в разделе о выравнивании элементов управления внутри кнопочной панели `TControlBar` выше в этой главе. Создается список объектов кнопочной панели `TControlBandInfo`, список сортируется, все элементы управления скрываются (`Visible = false`), а затем отображаются по одному, начиная с самого верхнего. Класс `TControlBandInfo` показан в листинге 5.33 и описан в разделе о выравнивании элементов управления внутри кнопочной панели `TControlBar`.

Обратите внимание, что основная форма перемещается и ее размер изменяется, если разрешение экрана вызывает чрезмерное увеличение и перемещение формы. Это делается для демонстрации возможностей работы с программой при использовании экранов с разной разрешающей способностью. Более подробно эта тема рассматривается в следующем разделе.

Решение проблемы использования разных типов экрана

В этом разделе мы рассмотрим способы определения типов экрана для согласованного отображения интерфейса на компьютерах с разными параметрами экрана: разрешение экрана, размер шрифта и количество цветов. Наиболее важные среди них — разрешение экрана и размер шрифта. Ограниченное количество цветов обычно не является большой проблемой, если только в оформлении интерфейса не используется большое количество цветов палитры True Color.

Решение проблемы использования экранов с разным разрешением

Для определения текущего разрешения экрана нужно считать значения свойств `Width` и `Height` глобальной переменной `Screen`. Затем нужно использовать свойство `ScaleBy` для масштабирования каждого элемента управления, чтобы интерфейс имел одинаковые размеры для разных разрешений экрана. При этом важно понимать, что реализация этих действий сопряжена с ограничениями. При создании достаточно большого интерфейса нужно иметь в виду, что он может плохо масштабироваться для отображения на экране с низким разрешением, а потому его придется уменьшить. Для масштабирования приложения нужно подсчитать коэффициент масштабирования, деля разрешение экрана во время создания приложения на разрешение экрана во время запуска приложения. Именно этот коэффициент используется для свойства масштабирования `ScaleBy`. Этот метод вместе с другой более подробной информацией описывается в руководстве пользователя, который поставляется вместе с C++Builder.

Решение проблемы использования разных шрифтов

При создании приложения используется свойство формы `PixelsPerInch`, которое обозначает размер шрифта, используемого при создании приложения. По умолчанию для свойства `Scaled` формы `TForm` задается значение `true`. Это значит, что форма и элементы управления в ней будут автоматически подогнаны при изменении размера шрифта. Во многих случаях, например в программе MiniCalculator, это не нужно, и для предотвращения такого эффекта следует использовать динамическое масштабирование, т.е. задать значение `false` для свойства `Scaled`. Для определения размера шрифта во время выполнения приложения нужно просто считать значение свойства `PixelsPerInch` глобальной переменной `Screen`.

Решение проблемы использования разного количества цветов

Проблема использования разного количества цветов возникает в том случае, когда приложение было создано на компьютере с бо́льшим количеством цветов, чем количество цветов на компьютере, где запускается это приложение. Если приложение использует больше цветов, чем имеется в данный момент, то для его цветов будут применены максимально близкие к ним другие цвета. Иногда это приводит к тому, что прекрасный интерфейс становится отвратительным. Если с вашим приложением происходит такая ужасная метаморфоза, то нужно предпринять соответствующие действия для ее устранения.

Для этого можно применить следующие два подхода. Самый простой заключается в сокращении до минимума количества цветов интерфейса, которые используются при работе с приложением. Обычно 16-битовая палитра цветов подходит для большинства приложений. Другой способ заключается в определении используемого операционной системой количества цветов при выполнении приложения и использовании разных интерфейсов для различных палитр.

Для определения количества цветов, используемых при выполнении приложения, следует использовать WinAPI-функцию `GetDeviceCaps()`, передавая в качестве параметра дескриптор `Canvas->Handle` одного из визуальных элементов управления, например основной формы, с помощью следующего кода.

```
int ColorDepth = 0;

if(GetDeviceCaps(Canvas->Handle, RASTERCAPS) & RC_PALETTE)
{
   ColorDepth = GetDeviceCaps(Canvas->Handle, COLORRES);
}
else
{
   ColorDepth = GetDeviceCaps(Canvas->Handle, BITSPIXEL)
                * GetDeviceCaps(Canvas->Handle, PLANES);
}
```

В проекте `ScreenInfo.bpr`, который находится на прилагаемом к книге компакт-диске, приводится пример получения и отображения на экране информации о количестве цветов, размере шрифта и разрешении экрана.

Решение проблемы усложнения кода при создании интерфейса пользователя

Программирование интерфейса приложения представляет собой очень сложную задачу. Рассмотрим в качестве примера программу MiniCalculator. Можно согласиться, что она не выполняет ничего выдающегося. Действительно, код арифметических действий сложения, вычитания, умножения и деления очень мал, и составляет приблизительно 10–20 строк. Несколько сот строк используется для преобразования чисел к разным системам счисления и т.д., а остальные 2500 строк посвящены исключительно интерфейсу. Но даже такой интерфейс нельзя назвать сложным. Следовательно, при создании больших приложений очень пригодятся методы и инструменты, которые помогут справиться со сложным кодом интерфейса пользователя. Рассмотрим кратко два таких метода на примере приложения MiniCalculator.

Список действий

Единственное назначение списка действий заключается в упрощении программирования интерфейса. Эта цель достигается благодаря централизованному управлению кодом. Хотя теория списков действий и методы их использования подробно рассматриваются в руководстве пользователя (и интерактивной справке), которая поставляется с C++Builder 5, здесь все же не помешает рассмотреть один практический пример.

В программе MiniCalculator список действий `ActionList1` используется для операции копирования, выполняемой командой меню Copy основной формы, и операции копиро-

вания, выполняемой командой контекстного меню CopyPopupMenu. В программе MiniCalculator предусмотрено копирование текущего списка выполненных команд (CopyHistoryAction), текущего числа в дисплее (CopyNumberAction) или текущего числа в памяти калькулятора (CopyMemoryAction). Каждое действие имеет собственную комбинацию клавиш ускоренного доступа, которая указывается в свойстве действия ShortCut: для CopyHistoryAction используется комбинация клавиш <Ctrl+H>, для CopyNumberAction — <Ctrl+N>, а для CopyMemoryAction — <Ctrl+M>. Наконец, остается только создать код обработчика события OnExecute для всех этих действий так, как показано в листинге 5.43.

Листинг 5.43. Код обработчика события OnExecute для действий CopyHistoryAction, CopyNumberAction и CopyMemoryAction

```
//---------------------------------------------------------//
void __fastcall TMainForm::CopyHistoryActionExecute(
                                            TObject *Sender)
{
   AnsiString HistoryString = HistoryLabel->Caption;
   Clipboard()->AsText = HistoryString;
}
//---------------------------------------------------------//

void __fastcall TMainForm::CopyNumberActionExecute(
                                           TObject *Sender)
{
   AnsiString NumberString = LCDScreen->Caption;
   if(ExponentLabel->Caption != "E+0" &&
      ExponentLabel->Caption != "E-0")
   {
      NumberString += ExponentLabel->Caption;
   }
   Clipboard()->AsText = NumberString;
}
//---------------------------------------------------------//

void __fastcall TMainForm::CopyMemoryActionExecute(
                                           TObject *Sender)
{
   Clipboard()->AsText = MemoryString;
}
//---------------------------------------------------------//
```

Теперь, после определения этих действий, для сопоставления им команд меню следует присвоить свойству Action каждой команды меню соответствующее действие. Например, в контекстном меню CopyPopupMenu для свойства Action команды меню CopyHistory1 следует задать значение CopyHistoryAction. Затем в основной линейке команд меню основной формы MainMenu для свойства Action команды меню History1 (Copy) следует задать значение CopyHistoryAction. Аналогично задаются соответствующие свойства ShortCut,

ImageIndex и Caption. Кроме того, события OnClick для них задаются равными событию OnExecute действия CopyHistoryAction. Следовательно, это действие будет выполнено при щелчке на этой команде меню.

Представленный здесь пример обладает преимуществом простоты кодирования этих действий. При создании интерфейса с многими командами меню и панелями инструментов список действий может оказаться очень полезным.

В проекте MDIProject.bpr, который находится на прилагаемом к книге компакт-диске, приводится пример использования действий. Как можно заключить из названия проекта, это — приложение с многодокументным интерфейсом и, как большинство таких приложений, содержит команды меню Windows, которые позволяют пользователям разными способами организовать работу с дочерними окнами: расположить горизонтально, вертикально или каскадом. Для этого нужно добавить в список действий TActionList приложения такие стандартные действия, как TWindowTileHorizontal, TWindowTileVertical и TWindowCascade. Для этого не придется создавать дополнительного кода, а только присвоить свойствам Action соответствующих команд меню каждое стандартное действие в списке действий. При использовании компонента TImageList и отображении свойств Images на список действий изображение для каждого стандартного действия будет автоматически добавлено в список изображений при наличии этого действия в списке действий. Если свойство SubMenuImages команды меню Window1 включить в список изображений, то эти изображения автоматически появятся перед каждой соответствующей командой меню. В проекте MDIProject.bpr в качестве примера использованы не стандартные изображения, а изображения, созданные разработчиком этого приложения.

Совместное использование обработчиков событий

В разделе этой главы, посвященном использованию символов наряду с текстом, уже приводился пример совместного использования обработчиков событий. Рассмотрим теперь более подробно, как этот метод может использоваться для упрощения кода интерфейса.

При создании пользовательского интерфейса особое внимание следует уделить организации ввода данных. Обычно интерфейс создается на основе нескольких экземпляров элемента управления. Используемые таким образом элементы управления обычно выполняют очень похожие операции или по крайней мере аналогичным образом интерпретируют введенные данные. Например, при создании формы ввода данных на основе нескольких компонентов TEdit его событие OnExit можно использовать для размещения содержимого каждого элемента управления TEdit в структуре или массиве для последующего использования в программе. Если форма содержит 10 элементов управления TEdit, то для них обычно требуется создать 10 соответствующих им обработчиков события OnExit — по одному для каждого элемента управления. Так как каждый обработчик события выполняет практически одинаковую задачу, то при их создании следует избегать излишнего дублирования кода. В случае больших интерфейсов это может привести к созданию длинного и сложного кода, который трудно сопровождать. С целью предотвращения этой избыточности следует для аналогичных задач совместно использовать один обработчик события.

В программе MiniCalculator этот метод используется несколько раз. Один раз при создании кода обработчика события OnClick для кнопок с цифрами. Так как каждая кнопка отличается только представляемой ею цифрой, то для них вполне можно использовать один обработчик события. Обработчик события OnClick каждой кнопки связан с одним и тем же обработчиком NumberSpeedButtonClick() конструктора основной формы MainForm.

Эту связь можно (и нужно) создать во время создания приложения. Переместите объявление обработчика функции `NumberSpeedButtonClick()` в управляемый IDE-средой заголовочный файл основной формы `MainForm`. Это будет открытый раздел определения класса формы со следующим комментарием.

```
// Управляемые IDE-средой компоненты
```

Затем нужно выбрать обработчик события `OnClick`, активизируя раскрывающийся список для события `OnClick` каждой кнопки с цифрой в окне Object Inspector. Этот метод использовался для присвоения событий `OnClick` компонентам цветовой палитры, используемой для формы `SettingsForm`.

Чтобы использовать этот метод, следует найти способ для определения кнопки, на которой произведен щелчок мышью. Для этого нужно создать показанную ниже переменную `TCalculatorButton` перечислимого типа `enum`, значения которой представляют каждую кнопку свойства `Tag`.

```
enum TCalculatorButton { cb1 = '1',
                         cb2 = '2',
                         cb3 = '3',
                         cb4 = '4',
                         cb5 = '5',
                         cb6 = '6',
                         cb7 = '7',
                         cb8 = '8',
                         cb9 = '9',
                         cbA = 'A',
                         cbB = 'B',
                         cbC = 'C',
                         cbD = 'D',
                         cbE = 'E',
                         cbF = 'F',
                         cb0,
                         cbSign,
                         cbPoint,
                         cbExponent,
                         cbAdd,
                         cbSubtract,
                         cbMultiply,
                         cbDivide,
                         cbEquals,
                         cbBackspace,
                         cbClear,
                         cbAllClear,
                         cbMemoryAdd,
                         cbMemoryRecall };
```

Для удобства зададим для каждого значения кнопки с цифрой тот символ, который эта кнопка представляет. Это позволяет непосредственно использовать метку `Tag` каждой кнопки с цифрой для обновления строки типа `AnsiString`, которая представляет содержимое дисплея калькулятора. При этом важно учесть, что значение `enum` каждой кнопки калькулятора является уникальным. Соответствующее значение `enum` каждой

кнопки присваивается в конструкторе основной формы MainForm. Например, для кнопки с цифрой 1 следует записать такую строку.

```
SpeedButton1->Tag = cb1;
```

В листинге 5.44 приведен код обработчика события NumberSpeedButtonClick().

Листинг 5.44. Код обработчика события NumberSpeedButtonClick()

```
void __fastcall TMainForm::NumberSpeedButtonClick(
                                          TObject *Sender)
{
   TSpeedButton* SpeedButton =
      dynamic_cast<TSpeedButton*>(Sender);

   if(SpeedButton)
   {
      ButtonPressNumber(
         static_cast<TCalculatorButton>(SpeedButton->Tag));
      ButtonUp(
         static_cast<TCalculatorButton>(SpeedButton->Tag));
   }
}
```

Во-первых, в коде обработчика события NumberSpeedButtonClick() выполняется динамическое приведение типа TObject* переменной Sender к типу указателя на класс TSpeedButton. В случае успешного приведения типов кнопка TSpeedButton инициирует это событие, и работа может быть продолжена. На следующем этапе просто вызывается функция ButtonPressNumber(), обновляющая дисплей той цифрой, которая отображена на нажатой кнопке. Функция ButtonPressNumber() принимает параметр типа TCalculatorButton. Даже если тип TCalculatorButton присвоить метке Tag каждой кнопки, метка Tag имеет тип int по определению. Для удаления предупреждения со стороны компилятора нужно выполнить статическое приведение (static_cast) типа метки Tag к перечислимому типу TCalculatorButton. Наконец, в коде вызывается функция ButtonUp() для отпускания кнопки и приведения типа метки Tag к нужному типу TCalculatorButton.

Описанный метод очень прост и эффективен. Однако, присваивая цифры только свойству Tag, можно затруднить восприятие кода, поскольку цифры имеют небольшое значение или вовсе не имеют его. Для улучшения читабельности в программе MiniCalculator используется несколько перечислимых типов, которые позволяют точно определить, что хранится в свойстве Tag каждого элемента управления.

Еще один метод определения нажатой кнопки основан на сравнении указателя Sender с указателем каждой кнопки до достижения их совпадения. Это можно выполнить с помощью приведенного ниже кода.

```
if(Sender == SpeedButton1)      { /* здесь располагается код */}
else if(Sender == SpeedButton2) { /* здесь располагается код */}
else if(Sender == SpeedButton3) { /* здесь располагается код */}
else if(Sender == SpeedButton4) { /* здесь располагается код */}
// и т.д. ...
```

Это достаточно простой и эффективный подход. Однако в этом примере мы использовали метод на основе метки Tag, так как он обладает большей гибкостью, например

позволяет более широко использовать значение метки `Tag`, чем просто в роли идентификатора. Например, помимо выполнения роли идентификатора, метка может также представлять значение кнопки.

Резюме

В этой главе подробно рассматриваются способы проектирования и создания пользовательского интерфейса. В ней представлена философия проектирования пользовательского интерфейса: пользовательский интерфейс должен удовлетворять ожиданиям пользователей и быть интуитивно понятным.

На примере приложения MiniCalculator показано, как C++Builder может применяться для создания эффективного и профессионально выполненного пользовательского интерфейса, который предоставляет пользователям доступ ко всем функциям приложения. Рассмотрены также наиболее часто используемые элементы управления C++Builder: `TStatusBar`, `TSpeedButton`, `TLabel`, `TPanel` и `TControlBar`. Описаны методы работы с указателем мыши и подсказками, управление фокусом ввода, настраиваемые кнопки, элементы управления, команды меню, закрепление, изменение размеров, привязки и ограничения, запоминание предпочтений пользователя в системном реестре, решение проблемы разных параметров экрана, упрощение кода интерфейса. К сожалению, нам удалось только бегло познакомиться с этими понятиями и методами! Дело в том, что в этой главе нельзя было охватить или даже упомянуть другие многочисленные вопросы программирования интерфейса, особенно проблему локализации и интернализации приложения. Это отдельная и очень большая тема, которая заслуживает подробного обсуждения в отдельной главе. Вводное описание проблемы локализации и интернализации приложения приводится в главе 28.

Итак, создание эффективного интерфейса представляет собой очень сложную задачу. Но несмотря на это, создание интерфейса — благодарное занятие, которое позволяет получить конкретный и наглядный результат. Читателю предлагается самостоятельно поэкспериментировать и попробовать усовершенствовать программу MiniCalculator. Например, в нее можно включить обработку чисел с плавающей запятой, которая описывается в главе 30.

Изучение этой главы закладывает фундамент для создания пользовательского интерфейса приложения.

Глава

6

Компиляция и оптимизация приложения

Джарод Холингвэрт

В этой главе рассматриваются принципы работы компилятора и компоновщика. Вы познакомитесь с методами максимального ускорения компиляции, а также работы приложения и приемами оптимизации.

Компиляция и оптимизация являются взаимосвязанными понятиями. *Компиляция* — это процесс преобразования высокоуровневого кода и данных, с которыми работает программист, в низкоуровневый код и данные, с которыми работает центральный процессор. *Оптимизация* — это процесс улучшения какой-либо характеристики приложения, например повышение производительности или уменьшение размера. Как будет показано далее, оптимизация на очень низком уровне может включать переупорядочение инструкций машинного кода и выбор наилучшего набора инструкций для выполнения определенной работы. Частично эта работа выполняется автоматически оптимизирующим компилятором C++Builder.

Всем программистам необходимо знать основные принципы компиляции и оптимизации. Дело в том, что знания основ работы компилятора и понимание машинного кода, который генерируется компилятором, помогут программисту в программировании и отладке кода. Оптимизация должна быть одной из первоочередных целей проекта.

Далее в этой главе термин *компилятор* будет относиться к компилятору C++Builder, а термин *оптимизатор* — к процессу оптимизации, который выполняется этим компилятором. При этом следует учесть, что оптимизатор — это не какая-то отдельная утилита, а составная часть компилятора.

После прочтения этой главы читатель сможет работать с большой отдачей, станет лучше понимать принципы функционирования приложения, узнает способы повышения его эффективности. Эту главу также следует рассматривать как вступительную перед прочтением главы 7 об отладке приложения.

Принципы работы компилятора

Компилятор является наиболее часто используемым компонентом C++Builder. Именно этот компонент чаще всего хвалят и ругают программисты. Как бы то ни было, компилятор является составной частью C++Builder, без которого большинство других компонентов C++Builder были бы просто бесполезны.

Что происходит при выборе команд меню Make или Build? При отсутствии ошибок вы получите выполняемую версию приложения. Однако за кулисами этого процесса скрыто гораздо больше. Для создания приложения компилятор C++Builder обрабатывает каждый модуль, выполняя несколько разных фаз компиляции кода для создания выполняемого файла или DLL-файла. Ниже кратко описаны все эти фазы.

- **Фаза 1: препроцессор**. Вызываются директивы препроцессора, в исходный код вставляются макросы и заголовочные файлы.
- **Фаза 2: проверка меток и синтаксический разбор**. Анализируется выходной код препроцессора, включая проверку меток в исходном коде и создание синтаксической иерархической структуры для генерации кода.
- **Фаза 3: генерация кода**. Для каждой команды генерируются собственные машинные команды данного компьютера. Этот код оптимизируется с помощью спаривания инструкций и других операций, которые зависят от типа процессора. Полученный машинный код затем записывается на диск в виде объектного файла с расширением `.obj`. При необходимости в конце файла дописывается отладочная информация.
- **Фаза 4: компоновка**. Компоновщик считывает все определения сегментов из каждого объектного файла и создает глобальную таблицу символов. Список символов необхо-

дим для генерации выполняемого файла. Затем компоновщик комбинирует откомпилированный код из заданных модулей, ресурсы, а также файлы форм со статически связанными библиотеками и специальным объектным файлом запуска для создания итогового выполняемого файла или DLL-файла.

Компилятор C++Builder использует так называемую рекурсивную нисходящую модель с неограниченным предварительным просмотром (recursive descent model with infinite lookahead). Это означает, что предварительная обработка, проверка меток, синтаксический анализ и генерация кода выполняются одновременно и рекурсивно во время компиляции, а не за счет нескольких проходов, как в других компиляторах.

Одни типы оптимизации — такие, как свертывание подчиненных выражений (например, для вычисления математических констант), создание иерархической структуры выражений, вставка функций и замена интегральных постоянных, — характерны для высокоуровневой (внешней) компиляции. Другие типы оптимизации, например планирование и инвариантная оптимизация кода, присущи низкоуровневой (внутренней) компиляции. Оптимизация может выполняться только в границах функций.

Благодаря разным оптимизирующим уловкам компилятор C++Builder 5 способен создавать гораздо более быстрый код по сравнению с другими компиляторами. Несмотря на это работа по его усовершенствованию продолжается, и ее результаты, очевидно, будут представлены в C++Builder версии 6. Более подробную информацию о компиляторах и принципах их работы можно получить в перечисленных ниже книгах.

- *Introduction to Compiling Techniques*, by J.P. Bennett, Second Edition, McGraw-Hill 1996, ISBN 0-07-709221-X.
- *Compilers and Compiler Generators*. Эта отличная и насыщенная полезными сведениями книга доступна для интерактивного чтения, причем ее можно скопировать в формате PDF-файла по адресу: `http://www.scifac.ru.ac.za/compilers/`.

Компиляторам, принципам их работы и способам их создания посвящено очень много книг, которые читатель сможет найти в любой университетской библиотеке. Кроме того, в сети Internet на Web-сайтах нескольких университетов также предлагаются учебные планы изучения компиляторов.

Большинство параметров компилятора и компоновщика можно задать с помощью вкладок Compiler, Advanced Compiler, Linker и Advanced Linker диалогового окна Project Options. Однако только некоторые из них могут быть заданы в файле проекта или в командной строке.

Как уже говорилось в главе 2 о проектах и IDE-среде C++Builder, в C++Builder 5 используется новый формат файла проекта на основе языка Extensible Markup Language (XML). Для использования прежних параметров компилятора и компоновщика необходимо изменить файл проекта. Для этого файл проекта можно открыть в IDE-среде, выбрав команду меню Project⇨Edit Option Source. Параметры компилятора задаются в разделе `<OPTIONS>` файла проекта в строке `<CFLAG1 value="..."\>`, флаги компоновщика — в строке `<LFLAGS...\>`, параметры компиляции ресурсов — в строке `<RFLAGS...\>`, параметры компилятора Pascal — в строке `<PFLAGS...\>`, а параметры ассемблера — в строке `<AFLAGS...\>`.

В каталоге `Examples` основного каталога C++Builder находится прекрасная программа WinTools. Она содержит список параметров командной строки для компилятора и других инструментов командной строки, предусмотренных в C++Builder, а также содержит несколько прекрасных встроенных функций.

Полный перечень параметров командной строки для компилятора, компоновщика и других утилит командной строки можно найти в справке C++Builder 5.

Сокращение времени компиляции

Компилятор C++Builder работает очень быстро: он компилирует код на языке C++ почти в два раза быстрее компилятора GNU C++ и практически так же быстро, как компилятор Microsoft Visual C++. Однако читатель, который имеет опыт работы с компилятором Delphi, заметит, что компилятор C++Builder гораздо медленнее компилирует приложение. Относительно медленная компиляция программ на языке C++, по сравнению с компиляцией такой же программы Object Pascal в Delphi, объясняется следующими причинами.

- В C++ предусмотрено использование заголовочных (включаемых) файлов, а в Object Pascal — нет. Заголовочные файлы могут быть вложенными, а это существенно усложняет обработку кода. Простая программа из 10 строк может в итоге содержать сотни тысяч строк из-за вложенности заголовочных файлов, для обработки которых компилятору потребуется затратить много времени.
- В C++ используются макросы, а в Object Pascal — нет. Препроцессор должен выполнить синтаксический разбор и вставку макросов.
- В C++ применяются шаблоны, а в Object Pascal — нет. Шаблоны представляют собой достаточно сложные для анализа структуры.
- Семантика C++ должна удовлетворять стандарту ANSI. Причем "грамматика" C++ несколько сложнее "грамматики" Delphi, которая основана на языке Pascal, но соответствует стандарту Borland.

Вообще, C++ позволяет использовать более гибкие приемы программирования, чем Object Pascal. Однако такая гибкость достигается за счет увеличения времени компиляции и порой даже ухудшения читабельности кода. Для ускорения компиляции в C++Builder можно применить несколько простых способов. Наиболее существенное улучшение может быть достигнуто за счет использования предварительно откомпилированных заголовочных файлов. Этот и другие методы подробно описываются в следующих разделах.

Предварительно откомпилированные заголовочные файлы

Параметры предварительно откомпилированных заголовочных файлов представлены в группе Pre-compiled headers во вкладке Compiler диалогового окна Project Options. При выборе параметра Use pre-compiled headers (Использовать предварительно откомпилированные заголовочные файлы) или параметра Cache pre-compiled headers (Кэшировать предварительно откомпилированные заголовочные файлы) компилятор сохранит скомпилированные двоичные образы заголовочных файлов в отдельных модулях дискового файла (по умолчанию с расширением `.csm` в каталоге `lib` основного каталога C++Builder).

Последующее использование той же последовательности заголовочных файлов в другом модуле существенно ускоряет время компиляции этого модуля за счет использования уже скомпилированных заголовочных файлов. Выбор параметра Cache pre-compiled headers означает загрузку предварительно откомпилированных заголовочных файлов в оперативную память для еще большего ускорения процесса компиляции.

Применение директивы `#pragma hdrstop` указывает компилятору, что в данном месте модуля нужно прекратить генерацию предварительно компилированных заголовочных файлов. Тут очень важно отметить большое значение порядка следования заголовочных файлов перед директивой `#pragma hdrstop`. Изменяя этот порядок в двух разных модулях, можно изменить

код, полученный после компиляции этих заголовочных файлов в каждом модуле. Для этого необходимо компилировать и хранить оба списка заголовочных файлов в виде отдельных предварительно компилируемых групп заголовочных файлов.

Расположенные после директивы `#pragma hdrstop` заголовочные файлы будут компилироваться при каждой компиляции данного модуля. Обычно перед этой директивой следует располагать общие заголовочные файлы для двух или более модулей, чтобы избежать их повторной компиляции. А специальные заголовочные файлы, которые присутствуют только в данном модуле, следует помещать после этой директивы. Таким образом достигается наибольшее совпадение списков заголовочных файлов во всех модулях, что даст возможность получить преимущества от применения предварительно компилируемых групп заголовочных файлов.

В IDE-среде предусмотрена автоматическая вставка директивы `#pragma hdrstop` в новых модулях и размещение заголовочных файлов библиотеки VCL перед этой директивой, а специальных заголовочных файлов — после этой директивы. Пример группирования заголовочных файлов и их упорядочения показан ниже в верхней части модулей `LoadPage.cpp` и `ViewOptions.cpp` в листингах 6.1 и 6.2.

Листинг 6.1. Группа предварительно компилируемых заголовочных файлов в модуле `LoadPage.cpp`

```
//---------------------------------------
//LoadPage.cpp
#include <vcl.h>
#include <System.hpp>
#include <Windows.hpp>
#include "SearchMain.h"
#pragma hdrstop
#include "LoadPage.h"
#include "CacheClass.h"
//---------------------------------------
// Код...
```

Листинг 6.2. Группа предварительно компилируемых заголовочных файлов в модуле `ViewOptions.cpp`

```
//---------------------------------------
//ViewOptions.cpp
#include <vcl.h>
#include <System.hpp>
#include <Windows.hpp>
#include "SearchMain.h"
#pragma hdrstop
#include <Graphics.hpp>
#include "ViewOptions.h"
//---------------------------------------
// Код...
```

За счет эффективного группирования заголовочных файлов в каждом модуле и использования предварительно компилируемых заголовочных файлов часто почти в 10 раз можно ускорить компиляцию.

На заметку

Более подробную информацию о еще большем ускорении компиляции на основе использования предварительно компилируемых заголовочных файлов можно найти в полезных статьях в разделе `Articles` на Web-сайте разработчиков Borland по адресу `http://www.bcbdev.com/`.

Другие методы ускорения компиляции

Для ускорения компиляции можно использовать другие методы. Хотя они не столь эффективны, как оптимально сгруппированные предварительно компилируемые заголовочные файлы, но их все же стоит применять в тех случаях, когда время компиляции имеет очень большое значение, особенно при работе над крупными проектами.

Следует внимательно включать заголовочные файлы в модули, потому что на компиляцию ненужного кода может понапрасну расходоваться драгоценное время компилятора. Поэтому не рекомендуется включать в модули неиспользуемые заголовочные файлы. Если же такой неиспользуемый заголовочный файл уже входит в состав группы предварительно откомпилированных заголовочных файлов, то его не следует трогать. Кроме того, следует избегать очень частого изменения кода в заголовочных файлах, потому что при каждом таком изменении потребуется заново перекомпилировать ту группу предварительно откомпилированных заголовочных файлов, в состав которой он входит.

Используйте команду меню Make (Частичная сборка) вместо команды меню Build (Полная сборка). Дело в том, что при выборе команды меню Make компилятор попытается определить файлы, в которых был изменен код после предыдущей компиляции, и будет компилировать только их. А при использовании команды меню Build будет повторно выполнена компиляция всех файлов проекта. Очевидно, что для выполнения полной сборки (Build) потребуется гораздо больше времени, чем для частичной (Make), но выполнения полной сборки (Build) не всегда удается избежать.

Полную сборку (Build) рекомендуется выполнять после изменения параметров проекта, а также при использовании или обновлении файлов проекта другими разработчиками, что зафиксировано в системе контроля версий исходного кода. Кроме того, полная сборка (Build) необходима для компиляции готовой к распространению версии приложения. В результате такой компиляции может быть получена отладочная или бета-версия для тестирования либо готовая версия для продажи.

Для ускорения компиляции рекомендуется снять флажок параметра Don't generate state files (Не генерировать файлы состояния) во вкладке Linker диалогового окна Project Options. Это позволит ускорить последующие попытки компиляции проекта (особенно при первой компиляции при повторном открытии проекта и при работе с несколькими проектами в IDE-среде), так как компоновщик сохранит информацию о состоянии в файле.

Если работа с проектом пока ведется без отладки, отмените все параметры отладки с помощью кнопки Release во вкладке Compiler диалогового окна Project Options и снимите флажок Use debug libraries (Использовать отладочные библиотеки) во вкладке Linker. Если в данный момент компилируется не окончательная, а какая-то промежуточная версия приложения, то в таком случае рекомендуется выбрать параметр None в группе параметров Code optimization во вкладке Compiler диалогового окна Project Options и снять флажок Optimization в группе параметров Code generation во вкладке Pascal.

Очень важно внимательно изучить структуру приложения и рассмотреть возможность использования пакетов или DLL-файлов для отдельных модулей, особенно в больших проектах. При этом полная (команда меню Build) и частичная (команда меню Make) сборка будут выполняться гораздо быстрее.

Если в вашем приложении не используются операции с числами с плавающей запятой, то следует установить флажок параметра None в группе параметров Floating point во вкладке

Advanced compiler диалогового окна Project Options для ускорения времени компоновки, поскольку в этом случае не придется компоновать соответствующие библиотеки обработки чисел с плавающей запятой.

Таким образом, сократить время компиляции можно, изменяя параметры C++Builder и редактируя код. Однако при этом не следует забывать об используемом аппаратном обеспечении. Такие совершенные инструменты создания программного обеспечения, как C++Builder, предъявляют повышенные требования к процессору, оперативной памяти скорости доступа к данным на жестком диске. Улучшая эти характеристики, также можно добиться более высокой скорости компиляции. Кроме того, более медленные IDE-устройства (например, проигрыватель компакт-дисков старого типа) рекомендуется размещать отдельно от жесткого диска на другом IDE-контроллере. Дефрагментация жесткого диска также может несколько сократить время компиляции.

На многопроцессорных (SMP) компьютерах можно одновременно компилировать сразу несколько модулей на всех процессорах. Утилита MAKE фирмы Borland не поддерживает эту функцию, но можно создать сценарий для запуска одновременной сборки отдельных модулей. Можно также использовать свободно распространяемую утилиту GNU Make с параметром командной строки `-j [jobs]` для включения режима параллельной обработки. Утилиту GNU Make для Windows можно скопировать по адресу `http://sourceware.cygnus.com/cygwin/`.

Скопируйте инсталляционные файлы Cygwin или по крайней мере файлы `cygwin1.dll` и `make.exe`. Документацию по их инсталляции и использованию можно найти по адресу `http://www.gnu.org/software/make`. Для использования утилиты GNU Make в C++Builder 5 необходимо экспортировать файл сборки (makefile) либо с помощью команды меню Project⇨Export Makefile в IDE-среде, либо с помощью консольной утилиты экспорта `BPR2MAK.EXE`, так как файл проекта теперь имеет формат XML. Более подробную информацию об утилите `BPR2MAK.EXE` вы найдете в интерактивной справке C++Builder 5.

Наконец, вряд ли стоит напоминать, что во время компиляции следует закрыть все другие активные приложения, особенно те, которые интенсивно используют оперативную память или процессор. При отсутствии достаточного количества оперативной памяти компиляция, несомненно, будет выполняться значительно дольше. Замечено также, что создание приложений на операционных системах Windows NT и Windows 2000 выполняется более эффективно, чем в Windows 95/98 (к тому же они предоставляет более удобные условия для отладки приложения).

Еще один метод уменьшения времени компиляции рассматривается в разделе о фоновой компиляции ниже в этой главе.

Рассмотрим теперь усовершенствования компилятора и компоновщика в C++Builder 5.

Компилятор и компоновщик в C++Builder 5

В C++Builder 5 возможности компилятора и компоновщика существенно расширены. Вероятно, наиболее желанным усовершенствованием является возможность выполнения фоновой компиляции. В C++Builder 5 также включена поддержка совместимости с Microsoft Visual C++, новые инструменты для работы с файлами, усовершенствованные предупреждения и переключатели, расширенная информация об ошибках и многое другое.

Фоновая компиляция

Фоновая компиляция позволяет повысить производительность за счет того, что во время компиляции программист может продолжать работу в IDE-среде. Чем больше и сложнее проект, тем больше времени потребуется для выполнения компиляции и тем большую выгоду можно извлечь, применяя фоновую компиляция.

Во время фоновой компиляции можно продолжать редактирование файлов и форм в IDE-среде. При этом компилируемые файлы временно становятся доступными только для чтения. В результате редактируемый файл с кодом или форма не могут быть изменены в течение непродолжительного времени.

Во время фоновой компиляции в IDE-среде можно также выбирать другие файлы с кодом или формы. При этом следует учесть, что некоторые изменения могут быть включены в процесс компиляции, а другие — нет, в зависимости от того, был ли файл уже откомпилирован до внесения в него изменений. Кроме того, компилятор может учесть незавершенные изменения кода и воспринять их как ошибочные с выдачей предупреждений и сообщений об ошибках.

Фоновая компиляция имеет некоторые ограничения. Ее нельзя использовать при работе с пакетами, с параметром Make All Projects (Сборка всех проектов) или параметром Build All Projects (Полная сборка всех проектов). Кроме того, во время фоновой компиляции не функционируют некоторые компоненты Code Insight, а также становятся недоступными отдельные команды меню, например Project⇨Options.

Фоновая компиляция выполняется на 25% медленнее обычной. Дело в том, что она выполняется в отдельном потоке, а потому скорость компиляции уменьшают переключения потоков, синхронизованный доступ к буферам редактирования и синхронизация с основным потоком IDE-среды. При ожидании окончания полной или частичной сборки, например, для выполнения тестового запуска приложения, фоновую компиляцию следует отключить. Для этого выберите команду меню Tools⇨Environment Options, а затем на вкладке Preferences диалогового окна Environment Options снимите флажок параметра Background compilation группы параметров Compiling.

Наилучший способ использования фоновой компиляции заключается в периодическом выполнении компиляции (обычно с помощью команды меню Project⇨Make или комбинации клавиш <Ctrl+F9>) и продолжении работы. Если проект создается группой разработчиков, то фоновую компиляцию следует начинать после включения в вашу версию проекта всех изменений, сделанных вашими коллегами (обычно после операции обновления файлов, которая выполняется с помощью системы контроля версий). И только после этого можно продолжить работу с проектом.

При тестовом запуске программы компиляция, вероятно, будет выполнена очень быстро, если будут соблюдены все приведенные выше рекомендации. На последних этапах работы над проектом не рекомендуется изменять заголовочные файлы, так как после этого потребуется повторная компиляция всех модулей, в которых они используются. Именно на этом последнем этапе (перед тестовым запуском) разработчику не остается ничего другого, кроме как дожидаться окончания компиляции.

Дополнительные усовершенствования компилятора

Для совместимости с Microsoft Visual C++ в новую версию C++Builder включены два новых модификатора функций — `__msfastcall` и `__msreturn`. Они позволяют создавать функции в виде DLL-модулей, которые могут быть использованы в приложениях Microsoft Visual C++. Для указания использования модификатора функции `__msfastcall` можно использовать два ключа компилятора — `-VM` и `-pm`.

Кроме того, семь новых объявлений `__declspec` могут использоваться для определения выполняемых ролей. Объявление `__declspec (naked)` указывает компилятору, что не нужно генерировать код пролога и эпилога (установки и восстановления кода вызова функции) для отдельной функции. Объявление `__declspec (noreturn)` информирует компилятор о том, что функция не возвращает значение (например, это функция выхода, которая прекращает работу приложения). Прежде при использовании такой функции компилятор выдавал предупреждение.

Новым компонентом компилятора также является параметр Extended error information (Q) (Расширенная информация об ошибке) во вкладке Compiler диалогового окна Project Options. При установке флажка этого параметра становится доступной дополнительная информация о сообщениях компилятора, например о контексте ошибки или предупреждения (имя той функции, при выполнении синтаксического анализа которой возникла ошибка). Для просмотра расширенной информации щелкните на символе (+) возле предупреждения или сообщения об ошибке.

Полезным дополнением режима отладки является новый макрос препроцессора __FUNC__. Он может расширяться до строкового литерала с именем функции, в которой он содержится. Например, приведенный ниже код позволяет генерировать сообщения Enter(TForm1::Button1Click) и Exit(TForm1::Button1Click) в журнале регистрации отладочных сообщений (для просмотра этого журнала следует выбрать команду меню View⇨Debug Windows⇨Event Log).

```
#define DFUNC_ENTRY OutputDebugString("Enter(" __FUNC__")")
#define DFUNC_EXIT OutputDebugString("Exit(" __FUNC__")")

void __fastcall TForm1::Button1Click(TObject *Sender)
{
   DFUNC_ENTRY;
   ShowMessage("In the middle.");
   // Некоторый код.
   DFUNC_EXIT;
}
```

Макрос __FUNC__ также можно применять в методах, объявленных в определениях классов.

Усовершенствования компоновщика

Новая вкладка Advanced Linker диалогового окна Project Options содержит несколько новых и прежних параметров компоновщика, включая параметры задержки до загрузки указанных DLL-файлов. Это параметр позволяет загружать выбранные DLL-файлы только при вызове содержащихся в них функций, что ускоряет запуск приложения и снижает требования к оперативной памяти для редко используемых DLL-файлов. Это также удобно при создании модульных приложений, поскольку приложению не требуется придавать DLL-файлы, если оно используется в "сокращенном" варианте.

В C++Builder 5 имеется несколько новых ключей, которые используются в режиме командной строки. Причем вкладка Advanced Linker может использоваться для указания некоторых из них. Среди прочих следует отметить ключ -GC для указания строкового комментария для изображения, -GD — для генерации совместимого с Delphi файла ресурсов (DRC), а также ключ -ad для компоновки с 32-разрядным драйвером Windows. (Для этого еще потребуется изменить заголовочные файлы DDK и создать соответствующие библиотеки импорта.)

Кроме того, в C++Builder 5 можно более гибко управлять предупреждениями компоновщика. Более подробную информацию по поводу этих и других усовершенствованиях компоновщика можно найти в разделе "Linker Enhancements" в интерактивной справке C++Builder 5.

Введение в принципы оптимизации

Оптимизацией называется процесс улучшения отдельного аспекта приложения, например скорости выполнения, размера, требований к оперативной памяти, эффективного использования пропускной способности сети или скорости доступа к жесткому диску. Оптимизация

представляет собой одновременно науку и искусство, так как для ее осуществления требуется аналитический и творческий подход.

Закон Мура (Moore's Law) о том, что удельная плотность транзисторов на площади интегральной микросхемы удваивается каждые 18 месяцев, довольно точно выполнялся более трех десятилетий. Современный процессор Pentium III содержит около 28 миллионов транзисторов. Вместе с аналогичным увеличением тактовой частоты процессора это позволяет добиться практически таких же темпов роста производительности процессоров. Производительность и другие параметры остальных компонентов аппаратного обеспечения также постоянно растут. Появляются более емкие модули оперативной памяти и жесткие диски, более быстродействующие сети, видеоадаптеры и модемы. Так почему же пользователи не удовлетворены имеющимся у них аппаратным обеспечением? Только потому, что все эти аппаратные компоненты еще не достаточно быстродействующи и емки. Дело в том, что по мере возрастания параметров аппаратного обеспечения растут и требования, предъявляемые к ним со стороны программного обеспечения.

Компьютерные игры отчасти критикуются за непрекращающуюся гонку аппаратного и программного обеспечения. Современные игры, особенно насыщенные сложной 3D-графикой, до предела используют возможности аппаратного обеспечения. Кроме того, операционные системы и пользовательские интерфейсы становятся все сложнее, а в прикладных программах используется больше графических элементов, обрабатывается больше данных и выполняется больше функций. Самый свежий пример — использование нелинейных методов видеомонтажа, который стал относительно новым увлечением в среде непрофессиональных пользователей персональных компьютеров. Для этого требуется огромный объем оперативной памяти, быстродействующий процессор и чрезвычайно большой и быстродействующий жесткий диск.

Существует два наиболее общих способа повышения производительности: за счет улучшения аппаратного или программного обеспечения. Первый путь характеризуется хорошо известным пределом повышения производительности и обычно связан с неприемлемо высокими затратами. Но им не стоит пренебрегать, так как подчас улучшение аппаратного обеспечения является самым простым способом повышения производительности, особенно домашних приложений.

Однако описание методов оптимизации производительности за счет улучшения аппаратного обеспечения, например, на основе конфигурирования или замены его отдельных компонентов выходит за рамки этой книги. Речь будет идти только об оптимизации программного обеспечения.

Для многих приложений производительность не имеет большого значения, и, наоборот, есть некоторые типы приложений и функций, для которых требуется обеспечить максимальную производительность. В большинстве случаев это означает повышение их скорости, уменьшение размера или требований к оперативной памяти.

Оптимизацию нужно предпринимать только для уже готового рабочего приложения, которое прошло этап тестирования и отладки. Для этого следует четко сформулировать поддающиеся оценке цели такой оптимизации. Если приложение удовлетворяет предъявленным требованиям, его не следует оптимизировать. При этом всегда следует избегать чрезмерной оптимизации. Рекомендуется прекращать оптимизацию сразу же по достижении поставленных целей.

Цели оптимизации должны быть реальными. Бессмысленно задаваться целью создать приложение для сжатия часового полноэкранного видеоролика с коэффициентом сжатия 20:1 в течение одной минуты (это в принципе невозможно выполнить программными средствами на современном компьютере). Если это нельзя определить заранее, то рекомендуется посоветоваться с другими разработчиками и на основе их мнения принять правильное решение.

В большинстве случаев целью повышения производительности приложения является достижение уровня, который приемлем для пользователей, и не более. Вряд ли нужно уменьшать размер программы до 1 МБайт, если пользователь вполне доволен тем, что она помещается

на дискете емкостью 1,44 МБайт. При оптимизации по скорости вполне достаточно достичь уровня, когда, *по мнению пользователя*, приложение работает удовлетворительно.

Оптимизацию следует выполнять только для тех частей приложения, которые нуждаются в этом. Для оптимизации приложения необходимо понимать, как оно работает и где скрываются резервы для его оптимизации. Далее нужно сконцентрировать внимание и усилия и выработать такую стратегию оптимизации, благодаря которой можно было бы добиться достижения наилучших результатов.

При этом важно учитывать, что оптимизация часто усложняет код. Это может оказать отрицательное влияние на поддержку, тестирование и стабильность работы программы. Чем сложнее код, тем вероятнее появление ошибок при внесении изменений в это приложение в будущем. Кроме того, такие ошибки труднее найти и исправить. Более сложный код труднее понять, а потому в будущем для внесения в него изменений разработчикам потребуется затратить больше времени.

Каждый проект должен иметь набор четко сформулированных задач. Они могут существенно варьироваться от одного проекта к другому. Ниже перечислены приоритетные цели оптимизации.

- Скорость.
- Размер.
- Простота сопровождения.
- Простота тестирования.
- Возможность повторного использования.
- Стабильность.
- Масштабируемость.
- Переносимость.
- Практичность.
- Безопасность.

Макет приложения определяет уровень каждой цели так, чтобы все эти цели учитывались в самом начале создания приложения. Предполагая, что скорость и размер приложения имеют наивысший приоритет, необходимо на начальном этапе создания приложения на самом высоком уровне проектирования приложения внести соответствующие изменения.

Низкоуровневую оптимизацию следует применять на поздней стадии проектирования. Часто высокоуровневые изменения макета приложения могут привести к тому, что усилия по низкоуровневой оптимизации могут оказаться бесполезными. Это особенно важно для оптимизации на самом низком уровне, т.е. ручной подгонке кода на ассемблере (в настоящее время эти навыки стали почти эзотерическим знанием), которую придется, вероятно, повторить в случае изменения кода и применения высокоуровневых методов оптимизации, например использования другого алгоритма.

Любые связанные с оптимизацией изменения кода следует тщательно документировать. Это также нужно делать в тех случаях, когда имеются подозрения о том, что какие-то области кода функционируют не очень эффективно.

Оптимизация по скорости

Оптимизация по скорости является наиболее распространенной задачей оптимизации. Скорость работы приложения часто впечатляет пользователей. Кроме того, в связи с повышением сложности вычислений и структур в современных приложениях, а также с возрастанием объема обрабатываемых данных, скорость работы процессора все еще остается основ-

ным ограничивающим фактором. Оптимизирующий компилятор C++Builder создает наиболее быстрый выполняемый код, который не уступает в скорости коду, полученному с помощью компиляторов Delphi, Visual C++ и GNU C++.

На заметку

Во время создания этой книги Джон Якобсон (John Jacobson) провел сравнительный анализ скорости выполнения кода, полученного с помощью компиляторов C++Builder, Delphi и Visual C++. Он рассмотрел разные решения специальных задач и привел результаты их выполнения на Web-сайте Code Efficiency Challenge по адресу `http://home.xnet.com/~johnjac/`.

Каждый оптимизирующий компилятор имеет определенные преимущества и недостатки, а потому для их сравнительного анализа необходимо рассмотреть результаты выполнения множества различных тестов. Те тесты, которые показывают пятикратную или более разницу в скорости выполнения кода, генерированного разными компиляторами, демонстрируют специфический недостаток одного компилятора, и его вряд ли следует учитывать в общей характеристике компилятора.

Оптимизирующий компилятор C++Builder в большинстве случаев повышает производительность кода только от 20% до 150%. В некоторых приложениях потребуется добиться еще большего повышения эффективности. Как правило, это такие приложения, которые характеризуются следующими особенностями.

- Они должны выполнять большое количество сложных математических вычислений, например, для научного или инженерного моделирования реальных процессов, для фрактальных генераторов, а также в приложениях, где используется 3D-графика.
- Они должны обрабатывать большой объем данных, например выполнять операции сжатия, сортировки, поиска или шифрования.
- Они должны решать сложные задачи с многочисленными итерациями, сложным ветвлением или индексными выражениями, например, для имитации событий, наилучшей подгонки или решения головоломок.

Иногда основным фактором, лимитирующим производительность приложения, является нечто находящееся вне приложения, например низкоскоростной жесткий диск, перегруженная глобальная сеть или даже пользователь, который медленно реагирует на событие, блокирующее работу приложения в данный момент. Для оптимизации приложения по скорости разработано несколько методов, которые можно разбить на следующие категории.

- Параметры оптимизации компилятора.
- Изменения макета и алгоритмов.
- Низкоуровневое изменение кода.
- Изменение данных.
- Ручная настройка кода на ассемблере.
- Внешняя оптимизация.
- Скрытие скорости выполнения приложения.

Последний метод, строго говоря, хотя и не является методом оптимизации, но может быть очень полезен в некоторых ситуациях. Если пользователь может продолжить свою работу только после выполнения какой-либо задачи, то следует использовать индикатор ее выполнения или анимационный ролик, сообщающий пользователю о том, что приложение функционирует. Рекомендуется организовать продолжение обработки в фоновом потоке, чтобы пользователь мог продолжить свою работу.

Самая высокая производительность может быть достигнута благодаря изменениям макета и алгоритмов. Проще же всего выполнить оптимизацию с помощью параметров компилятора

и низкоуровневых изменений кода. Не все методы оптимизации могут пригодиться для повышения производительности приложения.

Следует отметить, что иногда применение методов оптимизации по скорости может привести к увеличению размера выполняемого файла, что противоречит оптимизации по размеру. В следующих разделах рассматриваются основные методы оптимизации по скорости на примере отдельно взятого приложения.

Кроссворд-головоломка Crozzle Solver

Рассмотрим в качестве примера приложение для решения кроссворда-головоломки. Это приложение относится к описанной выше третьей категории и содержит сложные ветвления и индексированные выражения. Нам будет полезно познакомиться с его описанием для последующего применения некоторых представленных ниже методов оптимизации.

Кроссворда-головоломка ("crozzle" от английских слов "crossword" + "puzzle") это вид кроссворда, в котором на исходной пустой решетке нужно разместить пересекающиеся слова из заданного списка слов. Основная цель заключается в достижении максимального количества пересечений слов.

За каждое использованное слово дается 10 очков, за буквы A–F — 2 очка, G–L — 4 очка, M–R — 8 очков, S–X — 16 очков, Y — 32 очка, и за Z — 64 очка. Слова могут быть использованы один раз и размещаться по горизонтали или вертикали в решетке с размерами 1510.

В решетке можно располагать только целые слова, а слова внутри слов не учитываются. Например, если в списке заданных слов находятся слова SCARE и CAR, и в решетке удалось разместить слово SCARE, то нельзя получить дополнительные 10 очков за вложенное слово CAR или символы C, A и R. Обычно в таком кроссворде используется список, содержащий от 110 до 120 слов длиной от 3 до 15 букв.

Этот пример выбран из-за его наглядности и забавности. Для работы этого приложения требуется достаточно большое время, к тому же на его примере можно продемонстрировать многие методы оптимизации и возникающие при этом трудности. Почти 12 лет назад автор создал грубую версию кода для такого кроссворда-головоломки на языке BASIC и позднее оптимизировал его на ассемблере, чтобы получить учрежденный журналом приз размером в $2000. К сожалению, он был недостаточно совершенен или недостаточно оптимизирован для компьютеров тех лет, чтобы предоставить хотя бы частичное решение за приемлемое время.

Благодаря значительно улучшенному алгоритму представленная здесь версия имеет более законченный вид с точки зрения ее функциональности и скорости выполнения, по сравнению с ранними версиями (хотя ее еще можно слегка улучшить). Код приложения содержит около 2900 строк кода, которые содержатся в 4 модулях и 3 заголовочных файлах. Это небольшой и вполне аккуратный проект. На рис. 6.1 показан внешний вид пользовательского интерфейса этого приложения Crozzle Solver. Он демонстрирует очень простой вариант кроссворда-головоломки.

Исходная версия этого приложения находится в подкаталоге `CrozzleInitial` на прилагаемом к книге компакт-диске. Окончательная версия, полученная после применения рассматриваемого ниже метода оптимизации, находится в подкаталоге `CrozzleFinal`. В них для удобства предлагаются версии кода для C++Builder 4 и 5, которые совершенно идентичны, за исключением файла проекта и списка неотложных задач.

Приложение Crozzle Solver можно использовать для решения кроссворда-головоломки с исходной совершенно пустой решетки или частично заполненной пользователем или компьютером решетки. Заданный список слов и их расположение на решетке хранятся вместе в текстовом файле с расширением `CRZ`. Их примеры приводятся в файлах `ExampleBlank.crz` и `ExampleToFinish.crz`. Этот файл достаточно просто отредактировать, вводя в него размеры решетки и полный список слов под параметрами решетки, по одному слову в строке.

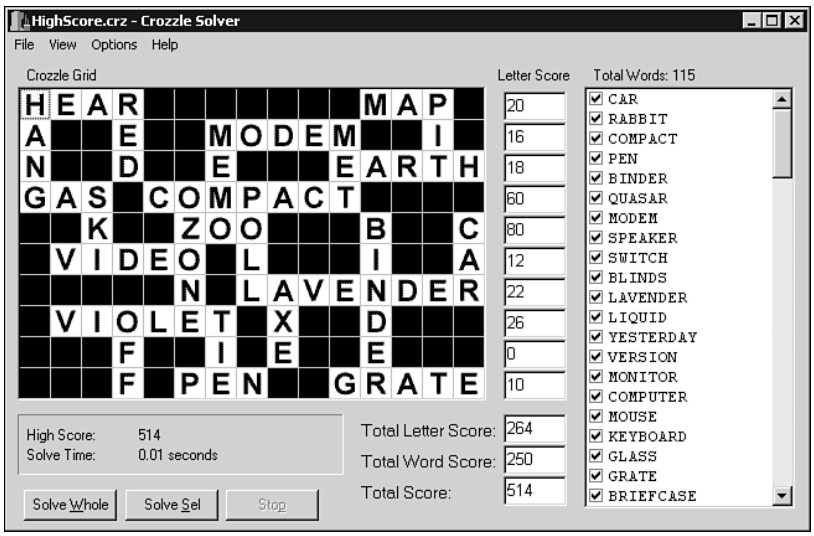


Рис. 6.1. Приложение Crozzle Solver с решением простого кроссворда-головоломки

Приложение кроссворда-головоломки может быть использовано для поиска расположения слов в целой решетке или в отдельном месте решетки. Для решения этой задачи для всей решетки следует использовать кнопку Solve Whole после загрузки программы. А для решения задачи в некоторой области укажите нужную область с помощью мыши, затем выберите команду меню Options⇨Selection Bounds Words или Options⇨Selection Bounds Interlocking Letters и используйте кнопку Solve Sel.

После этого механизм кроссворда-головоломки автоматически вычислит наилучшее текущее решение и запишет его в файл `HighScore.crz`, который впоследствии можно загрузить для просмотра в разных режимах. Поэкспериментируйте с ними, чтобы получить представление о способе работы приложения Crozzle Solver. Команды Save и Save As сейчас не очень полезны, поскольку трудно выполнить сохранение в процессе решения, к тому же решетка кроссворда-головоломки после получения решения принимает исходный вид.

Экспоненциальный рост времени вычислений

Благодаря оптимизации скорость решения кроссворда-головоломки постепенно возрастает. На каждом этапе оптимизации можно заметить постепенные улучшения. Они будут выражены в процентном соотношении по сравнению с предыдущим временем решения кроссворда, а также с представлением соотношения текущей и исходной скорости. Все измерения и тесты выполнялись в IDE-среде C++Builder на компьютере с процессором Pentium II 266 МГц, 128 МБайт оперативной памяти, под управлением операционной системы Windows 2000 Professional. Приведенные результаты получены в результате усреднения времен выполнения после трех прогонов.

На прилагаемом компакт-диске находится два примера конфигураций решетки и заданных слов — `RunComplete.crz` и `RunPartial.crz`. В файле `RunComplete.crz` находится список из 11 слов, для которых можно получить все возможные решения за сравнительно небольшой промежуток времени. Именно этот файл будет использован для всех измерений скорости в этой главе. Пример `RunPartial.crz` содержит список из 115 слов. В этом случае просто не-

возможно получить все возможные решения на современном аппаратном обеспечении, независимо от способа оптимизации программы. В обоих примерах используется исходная пустая решетка и предпринимается попытка поиска всех возможных комбинаций.

Самое лучшее решение, которое может получить обычный пользователь, содержит от 35 до 40 слов из всего списка слов, а конечный результат варьируется от 600 до 800 очков. Казалось бы, довольно легко создать программу, которая просто перебрала бы все комбинации слов для поиска самого лучшего результата. Не спешите! Количество всех возможных комбинаций чудовищно велико.

В табл. 6.1 показан график времени выполнения исходного приложения Crozzle Solver (откомпилированного без параметров оптимизации) для решения кроссворда-головоломки для первых 5–11 слов из списка, заданного в файле `RunComplete.crz`.

В этой таблице также показано количество рассмотренных комбинаций, количество найденных правильных решений и набольший полученный результат.

Таблица 6.1. Сводные данные для списков, содержащих от 5 до 11 слов

Число слов	Время, секунды	Число комбинаций	Число решений	Лучший результат
5	0,0318	3 577	1 492	90
6	0,288	31 257	9 892	102
7	2,96	317 477	101 604	120
8	47,1	4 765 661	1 192 436	146
9	458	44 057 533	11 123 772	164
10	6 294	554 577 981	152 343 008	190
11	79 560	>6 000 000 000	>1 900 000 000	208

Прежде всего отметим, что для списка из 11 слов (`RunComplete.crz`) на поиск решения уходит почти 22 часа! В среднем время поиска решения увеличивается почти на 1100% для каждого дополнительного слова. На рис. **6.2** показана диаграмма времени поиска решения из табл. 6.16. Эта таблица и диаграмма находятся в файле `CrozzleTimings.xls` электронной таблицы Microsoft Excel на прилагаемом к книге компакт-диске.

На основании этих табличных данных и диаграммы можно предположить, что время поиска решения увеличивается по экспоненциальному закону. Это становится очевидным после построения диаграммы с логарифмическими шкалами по оси абсцисс и ординат, на которой эти данные расположатся на прямой линии. Обе эти диаграммы также находятся на прилагаемом к книге компакт-диске.

Экстраполируя эти результаты, можно сделать вывод, что без оптимизации для решения кроссворда из 15 слов потребуется около 50 лет, для 17 слов — более 7 000 лет, а для 20 слов — более 10 миллионов лет! Вряд ли мы дождемся в скором времени появления процессора с тактовой частотой 1 ЭГц (экзагерц — это миллиард ГГц), но даже в случае его использования процесс поиска полного решения для 30 слов затянется на сотни миллионов лет.

Можно предположить, что время поиска решения и полученный максимальный результат будут расти не столь быстро для количества слов больше 40. Дело в том, что до этого момента в решетке остается достаточно места для слов, но теперь его практически не будет. Поэтому задача размещения дополнительных слов очень усложнится, а потому количество комбинаций не будет расти так же сильно, как для каждого отдельного слова, добавленного в список. В любом случае, ясно, что не существует способа для поиска полного решения для списка из 115 слов, по крайней мере за время существования вселенной.

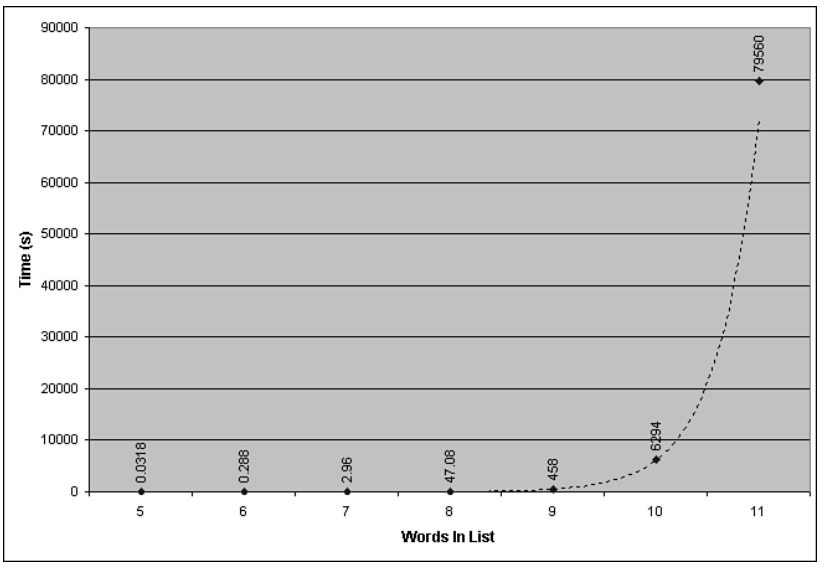


Рис. 6.2. Диаграмма времени поиска решения для 5–11 слов на основании данных из табл. 6.1

Итак, запомним контрольное время, полученное в данном примере кроссворда-головоломки с использованием исходного (основного) механизма поиска решения с отключением всех параметров оптимизации.

Исходное значение времени поиска полного решения: 79 560 секунд для 11 слов

А теперь приступим к улучшению этого результата с помощью оптимизации!

Параметры проекта для оптимизации по скорости

Для оптимизации скорости выполнения приложения в C++Builder предусмотрено несколько параметров проекта, которые располагаются в разных вкладках в диалоговом окне Project Options. Для максимального ускорения работы приложения рекомендуется установить следующие параметры проекта, предназначенные для оптимизации по скорости.

- Вкладка Compiler. Щелкните на кнопке Release. При этом большинство параметров оптимизации будет установлено автоматически. Выберите переключатель Speed в группе Code optimization, снимите флажки всех параметров в группе Debugging и флажок параметра Stack frames в группе Compiling.
- Вкладка Advanced Compiler. Выберите переключатель Pentium Pro в группе Instruction set, переключатель Quad word в группе Data alignment, переключатель Register в группе Calling convention, переключатель Automatic в группе Register variables, снимите флажок Fast и флажок параметра Correct Pentium FDIV flaw в группе Floating Point.
- Вкладка C++. Если это возможно в вашем приложении, отмените обработку исключительных ситуаций, снимая флажки параметров Enable RTTI и Enable exceptions в группе Exception handling. В противном случае установите флажок параметра Fast exception prologs. Затем в группе Virtual tables выберите переключатель Smart.
- Вкладка Pascal. В группе Code generation снимите флажки параметров Optimization и Aligned record fields, а потом снимите флажки параметров Stack frames и Pentium-safe FDIV. Затем снимите флажки всех параметров в группах Runtime errors и Debugging.

- Вкладка Linker. В группе Linking снимите флажки параметров Create debug information, Use dynamic RTL и Use debug libraries. В группе Map file выберите переключатель Off. При создании DLL-файлов, основание образа (image base) для всех DLL-файлов, используемых приложением, должно быть уникальным и иметь достаточное смещение для предотвращения перекрытия в памяти.
- Вкладка Packages. Снимите флажок Build with runtime packages.
- Вкладка Tasm. В группе Debug information установите флажок None.
- Вкладка CodeGuard. Снимите флажок параметра CodeGuard validation.

На заметку

Использование этих и других параметров оптимизации может затруднить отладку приложения. Оптимизатор может переупорядочить или даже удалить некоторые фрагменты кода и переменные. Вследствие этого могут быть остановлены контрольные точки и пошаговое выполнение программы в отладчике, т.е. измените способ их функционирования. Как уже говорилось выше, до выполнения оптимизации необходимо сначала полностью закончить отладку приложения.

В данном примере кроссворда-головоломки некоторые перечисленные выше параметры оптимизации не будут иметь никакого эффекта. Однако только за счет включения режима оптимизации кода по скорости уже можно получить значительное повышение скорости.

Текущее время выполнения: 51240 секунд. **Выигрыш**: 55% (в 1,55 раза)

При установке перечисленных выше параметров оптимизации следует проявлять осторожность. Например, параметр группы Instruction set во вкладке Advanced Compiler должен быть задан в соответствии с типом компьютера, на котором будет использоваться приложение, потому что процессоры не обладают совместимостью снизу вверх. Однако в данном примере приложения Crozzle Solver разница будет незначительной при использовании параметров 80386 и Pentium Pro из группы Instruction set во вкладке Advanced Compiler.

Использование параметра оптимизации Speed из группы Code optimization во вкладке Compiler и параметра Automatic (а не None) из группы Register variables во вкладке Advanced Compiler приведет к 40-процентному увеличению скорости для приложения Crozzle Solver.

Напомним, что в C++Builder 5 можно задавать разные параметры оптимизации для каждого модуля (параметры на уровне узлов). Это можно с успехом использовать, например, в случаях, когда весь критичный по отношению к скорости код сосредоточен в одном модуле. Тогда только для этого модуля можно задать максимально строгие параметры оптимизации по скорости.

Обнаружение уязвимых мест производительности приложения

Для применения методов оптимизации необходимо иметь четкое представление о принципах функционирования приложения и особенно тех его мест, в которых приложение затрачивает большую часть времени. В большинстве приложений 80% и 90% всего времени обработки приходится только на 10% и 20% кода, который обычно находится в нескольких циклах или нескольких часто выполняемых функциях.

Использование знаний об уязвимых местах приложения поможет добиться наилучших результатов с наименьшими усилиями. Для поиска уязвимых мест производительности приложения можно применять перечисленные ниже методы.

- Использование профайлера.
- Ручной мониторинг времени выполнения для некоторых разделов кода.
- Проверка макета и кода.

Профайлер

Простейший способ обнаружения уязвимых мест производительности приложения основан на использовании профайлера. *Профайлером* (*profiler*) называется инструмент, который отслеживает время выполнения приложения и предоставляет подробные статистические данные для некоторых частей приложения, которые используются чаще или дольше остальных. Большинство профайлеров отслеживает время выполнения фрагментов кода вплоть до уровня функций. Однако некоторые профайлеры могут отслеживать время выполнения фрагментов кода вплоть до уровня отдельных строк.

Обычно профайлеры функций могут отслеживать код приложения в том виде, в каком он есть, но для работы некоторых профайлеров в код иногда требуется внести некоторые изменения. Выбор профайлера функции или профайлера отдельных строк кода зависит от структуры приложения, но обычно достаточно использовать только профайлер функции. Профайлеры могут работать также с многопотоковыми приложениями.

В настоящее время существует несколько профайлеров, причем некоторые из них предназначены для работы с C++Builder. Например, профайлер Sleuth StopWatch фирмы TurboPower Software Company является частью пакета Sleuth QA Suite. Для создания профиля (т.е. мониторинга производительности) приложения Crozzle Solver далее в этой главе будет использоваться версия 1.0, в которой допускается создание профиля функций. Профайлер StopWatch также включает режим просмотра обратного ассемблирования кода, в котором код инструкций ассемблера располагается вместе с информацией, необходимой для низкоуровневой оптимизации кода на ассемблере.

Следует отметить, что во время создания этой книги уже была выпущена версия 2.0 пакета Sleuth QA Suite. Она существенно обновлена и включает новые компоненты, например анализ охвата (coverage analysis), создание профиля для отдельных строк, поддержка автоматизации, способность экспорта результатов просмотра в HTML-формате, настраиваемые пользователем представления результатов, а также анализ с использованием метода критического пути.

В пакет Sleuth QA Suite также входит утилита Sleuth CodeWatch, которая аналогична утилите CodeGuard. Она отслеживает утечку оперативной памяти и ресурсов, причем может фильтровать утечку в VCL-библиотеке. Она также способна перехватывать операции перезаписи в оперативной памяти, обнаруживать неверные параметры и возвращать значение для вызовов Win32 API-функций, регистрировать вызовы Win32 API-функций и много другое. Более подробную информацию и пробную версию пакета Sleuth QA Suite можно получить по адресу `http://www.turbopower.com/`.

Следует упомянуть и другие профайлеры.

- QTime фирмы Automated QA выпускается в упрощенной и стандартной версиях. Он может работать с C++Builder версии 3 и последующих версий. Стандартная версия содержит много полезных компонентов, включая охват кода и трассировку кода. Более подробную информацию и пробную версию этого профайлера можно получить по адресу `http://www.totalqa.com/`.
- Profiler фирмы RQ — не очень дорогой профайлер, но для его использования нужно добавить в код макросы-вызовы и скомпоновать его с DLL-файлами этого профайлера. Для выполнения этой задачи в нем предусмотрен редактор кода. Profiler может работать с C++Builder версий 1–5. Более подробную информацию и условно бесплатную пробную версию этого профайлера можно получить по адресу `http://ourworld.compuserve.com/homepages/rq/`.
- VTune Analyzer фирмы Intel содержит инструмент профилирования Code Coach с рекомендациями по оптимизации кода на языке C++, профиля вызовов функций, статического анализа кода на ассемблере для разных типов процессоров и многое другое. Однако он не интегрируется непосредственно с C++Builder и сложен в использовании.

Он имеет профайлер функций и отдельных строк. Более подробную информацию и условно бесплатную пробную версию этого профайлера можно получить по адресу `http://developer.intel.com/vtune/analyzer/`.

Пример создания профиля

При создании профиля приложения необходимо остановить все несущественные приложения, которые могут вмешаться и повлиять на результаты измерений времени выполнения приложения. Для получения точного представления о производительности приложения следует повторить профилирование несколько раз и вычислить среднее значение.

Общая проблема для всех профайлеров заключается в плохой обработке рекурсивных функций. При этом значения времени указываются для каждой функции, а не для каждого вызова этой функции, причем включается время, затраченное на работу подчиненных дочерних функций. Такую статистическую информацию довольно сложно интерпретировать. К сожалению, в приложении Crozzle Solver рекурсивные функции также используются для генерации решений кроссворда. Функция `SolveWord()` вызывается рекурсивно для создания каждого нового слова в решении. Профилирование этого примера приложения Crozzle Solver будет выполнено с помощью профайлера Sleuth StopWatch фирмы TurboPower. Для решения этой проблемы мы прибегнем к некоторым ухищрениям.

Нам потребуется несколько развернуть рекурсивные фрагменты кода. Это означает копирование и сцепление нескольких копий функций в рекурсивном цикле. Приложение Crozzle Solver имеет рекурсивный цикл из трех функций в ширину и одной функции в глубину.

Пусть функция A вызывает функции B, C и D, а функции B, C и D также вызывают функцию A. Это значит, что функция A расширяется до трех функций в ширину, а для возврата к ней придется углубиться на один уровень. При наличии одной функции A, которая вызывает сама себя, рекурсивный цикл будет иметь ширину в одну функцию и глубину 0.

В приложении Crozzle Solver роль функции A играет функция `SolveWord()`, а функций B, C и D — `SolveFirstWord()`, `SolveAdjacent()` и `SolveStandardWord()`. Функция `SolveWord()` размещает в текущем кроссворде-головоломке одно дополнительное слово с помощью одной из трех других специализированных функций. Функция `SolveFirstWord()` используется только для размещения в решетке всех комбинаций исходного слова. Функция `SolveStandardWord()` сцепляет другое слово из списка с текущим состоянием кроссворда.

Если помещаемое в решетке слово располагается рядом со словом в соседней строке или столбце, то может быть получено одно или несколько слов из двух букв. Для образования правильного слова функция `SolveAdjacent()` будет пытаться по-разному разместить слова в клетках решетки. За процессом размещения слов и разрешения возникающих комбинаций смежных букв можно проследить с помощью команды меню View⇨Thinking и View⇨Pause View. Новое слово либо будет размещено на соседних клетках, либо такое расположение слова будет отвергнуто.

К счастью, глубина рекурсии во время выполнения приложения Crozzle Solver, которая равна количеству циклов выполнения функции или размеру стека (выберите, что вам больше нравится), зависит от количества слов в списке, что означает возможность легкого управления этой рекурсией. Максимальная глубина рекурсии равна N+1, где N — количество слов в списке. Для использования списка из 7 слов нужно 8 раз развернуть рекурсивный цикл.

Сначала скопируем функции `SolveWord()`, `SolveFirstWord()`, `SolveAdjacent()` и `SolveStandardWord()` и вставим их семь раз. Затем изменим имена функций для семи копий каждой из них, добавляя номера 2, 3, 4 и т.д. в конце имени каждой функции. Таким образом мы получим семь идентичных копий каждой функции.

Затем сцепим их вместе, заменяя функцией `SolveWord2()` вызов функции `SolveWord()` в исходных функциях `SolveFirstWord()`, `SolveAdjacent()` и `SolveStandardWord()`. Далее

в функции `SolveWord2()` укажем вызов функций `SolveFirstWord2()`, `SolveAdjacent2()` и `SolveStandardWord2()`. Затем отредактируем код этих функций для вызова функции `SolveWord3()` и т.д. до тех пор, пока в конечном итоге не определим в функции `SolveWord8()` вызовы функций `SolveFirstWord8()`, `SolveAdjacent8()` и `SolveStandardWord8()`, которые, в свою очередь, вызывают исходную функцию `SolveWord()`, завершая таким образом цикл.

Если аккуратно выполнить все перечисленные выше действия, то эти операции не покажутся вам очень сложными. Так мы расширим цикл до ширины в восемь функций. Кроме того, необходимо создать прототипы всех только что созданных копий функций.

Внутри C++Builder запустим профайлер Sleuth StopWatch с помощью соответствующей команды меню Tools и укажем параметры профилирования этого проекта, отвечая на предложенные вопросы. В этом примере будет использован режим Trigger Mode для профилирования функции `SolveCrozzle()` и хронометража всех процедур кода и отдельных узлов. После указания параметров профилирования запустим приложение из профайлера StopWatch, загрузим файл со списком из восьми слов, найдем его полное решение, а затем закроем приложение.

После этого в окне профайлера StopWatch будут представлены статистические данные о результатах профилирования. На рис. 6.3 показан пример графического представления таких данных в окне профайлера StopWatch для приложения Crozzle Solver после получения полного решения для списка из семи слов.

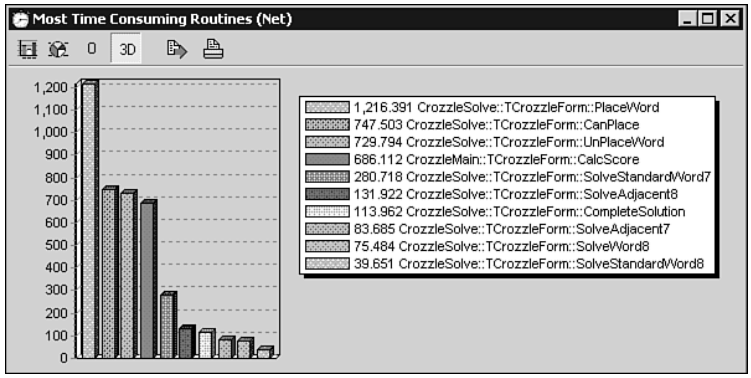


Рис. 6.3. Результаты профилирования в окне профайлера Sleuth StopWatch

Для получения истинных статистических результатов для рекурсивных функций необходимо сложить значения каждого показателя Times Called (Время вызова), Net Time (Чистое время) и Gross Time (Общее время) для всех восьми копий каждой из четырех функций. На основании этих значений будут получены значения Net % (Доля чистого времени), Net Average (Среднее чистое время) и Gross Average (Среднее общее значение). Окончательные результаты профилирования показаны в табл. 6.2. Следует отметить, что при использовании нерекурсивных функций весь процесс профилирования в целом будет выполняться гораздо проще.

Результаты профилирования показывают, что основные усилия необходимо приложить для улучшения функций `PlaceWord()`, `CanPlace()`, `UnPlaceWord()` и `CalcScore()`. Хотя для повышения производительности работы приложения Crozzle Solver необходимо повысить эффективность большого количества функций, но все же для более существенного улучшения работы приложения следует улучшить основной алгоритм.

Профилирование следует выполнять на различных этапах оптимизации для получения информации о достигнутом прогрессе после каждого изменения кода. При этом может получиться так, что изменения, внесенные для оптимизации приложения по скорости, на самом деле замедляют работу приложения!

Таблица 6.2. Окончательные результаты профилирования для приложения Crozzle Solver и списка из семи слов

Функция	Число вызовов	Чистое время (мс)	Доля чистого времени (%)	Среднее значение чистого времени (мс)	Общее время (мс)	Среднее общее время (мс)
PlaceWord	317462	1216,4	28,78	0,0038	1216,4	0,0038
CanPlace	544926	747,5	17,68	0,0014	747,5	0,0014
UnPlaceWord	317462	729,8	17,27	0,0023	729,8	0,0023
CalcScore	101604	686,1	16,23	0,0068	686,1	0,0068
SolveStandardWord	167474	356,6	8,43	0,002	24029,0	0,1435
SolveAdjacent	149988	253,95	6,01	0,0017	283,4	0,0019
SolveWord	317463	116,7	2,76	0,0004	28663,6	0,0903
CompleteSolution	101604	114,0	2,70	0,0011	803,6	0,0079
SolveFirstWord	1	0,00	0,00	0,00	4224,6	4224,6

Ручной хронометраж кода

Ручной хронометраж фрагментов кода основан на вставке специального хронометрирующего кода (stopwatch-type) в стратегически важных местах приложения. Если их расположение неизвестно, то следует начать сначала, как описано выше, раскрывая циклы и отдельные вызовы функций. Затем нужно продолжить движение в глубь кода до тех пор, пока не будут найдены области, которые потребляют большее количество времени. В листинге 6.3 продемонстрирован способ такого хронометража фрагмента кода.

Листинг 6.3. Хронометраж фрагмента кода

```
#include <time.h>

void TMyClass::SomeFunction()
{
    clock_t StartTime,
            StopTime;

    // Код.

    // Включение таймера.
    StartTime = clock();

    // Код таймера.
    // Остановка таймера.
    StopTime = clock();
    // Отчет о затраченном времени.
    ShowMessage("Elapsed time:" +
                FloatToStrF( (StopTime-StartTime)/CLK_TCK,
                             ffFixed, 7, 2) +
```

```
                " seconds");

    // Остальной код.
}
```

Хронометрируемый код располагается в центральной части этого фрагмента. Если он выполняется очень быстро (например, меньше секунды), то его можно окружить простым циклом, если только это не приведет к побочным эффектам, которые вызовут изменение поведения кода после первого цикла выполнения. После этого нужно измерить общее время выполнения заданных циклов и разделить его на количество циклов.

Если хронометрируемый фрагмент кода находится в функции, которая вызывается много раз, вряд ли можно воспользоваться функцией `ShowMessage()` для отображения затраченного времени. Вместо этого сведения следует записать в журнал учета событий отладки при работе в IDE-среде с помощью функции `OutputDebugString()` либо в указанный вами файл.

Исследование структуры

Метод исследования структуры может быть использован при наличии формальной документации по структуре приложения. Рассмотрите внимательно ядро структуры приложения, а затем найдите критические точки в потоке данных или плане исполнения. Проанализируйте псевдокод, блочные диаграммы, диаграммы потоков данных, а также разделы с описанием процессов в IPO-диаграммах, N-S-диаграммах и диаграммах Варнье-Орра (Warnier-Orr diagram).

Найдите процессы, которые вызываются во многих местах и циклах, особенно во вложенных циклах. Таким образом можно найти часто вызываемые фрагменты кода, в которых чаще всего находятся критические места для общей производительности приложения.

Исследование кода

Исследование кода означает изучение исходного кода на уровне циклов, сложных ветвлений, индексирования и арифметических действий с указателями, или математических вычислений. Объектно-ориентированный код обычно содержит много небольших методов и часто скрывает сложные арифметические действия с указателями.

Как при исследовании структуры, так и при исследовании кода, потребуется выполнить хронометраж кода, чтобы убедиться в том, что вы действительно имеете дело с тем кодом, на который приходится большая часть времени работы приложения. Для этого также следует использовать метод ручного хронометража, описанный выше в этой главе.

Используя эти методы, нужно не только найти уязвимые места, снижающие производительность, но и принять решение о том, какие из них имеют определяющее значение. Например, уязвимое место для производительности в фоновом потоке, который никак не влияет на работу пользователя, вряд ли стоит тщательно оптимизировать по скорости.

Оптимизация структуры и алгоритмов

Наибольшую выгоду можно получить при оптимизации структуры, выборе более удачных алгоритмов или оптимизации уже существующих алгоритмов. Для оптимизации структуры необходимо хорошо представлять, как работает приложение. С другой стороны, алгоритмы обычно реализуются на очень низком уровне, который изолирован от основной части приложения. Код алгоритма может быть очень сложным, но если он удачно спроектирован и написан, то его легко использовать.

Изменение структуры

Оптимизация структуры требует более широкого рассмотрения приложения: архитектуры приложения и технологий, которые используются для реализации его фундаментальных компонентов. Это в большей мере относится к большим приложениям, которые имеют модульную или распределенную структуру, или интерфейс для работы с другими приложениями. При создании приложений для Windows нужно также учесть множество других технических вопросов.

Вызов метода из внутреннего COM-объекта будет выполнен быстрее, чем такой же вызов из внешнего COM-объекта, и быстрее, чем такой же вызов из DCOM-объекта на удаленном компьютере. При масштабировании модульного приложения, в котором COM-объекты используются как каркас для использования DCOM-объектов, необходимо учесть эти особенности. Во многих случаях можно использовать альтернативные технологии, например CORBA или пользовательский интерфейс на основе сокетов протокола TCP/IP.

При работе с приложениями баз данных существует гораздо больше альтернативных вариантов. В C++Builder 5 уже содержится три метода доступа к базе данных: BDE, ADO Express и InterBase Express. Кроме того, скоро появится еще один вариант — DB Express. Следует отметить, что технология доступа к базе данных BDE реализована более чем 20 способами различными сторонними разработчиками. При этом всегда можно найти компромиссный вариант между требованиями к производительности, практичности и набором функциональных возможностей.

При выборе наиболее оптимальной технологии необходимо проделать большую работу по поиску и анализу преимуществ и недостатков каждого альтернативного варианта. Вероятно, наиболее полным источником данных являются списки рассылки новостей Borland. В них очень часто рассматриваются вопросы производительности и оптимального проектирования приложения. Поэтому при наличии конкретного вопроса с помощью этих списков рассылки можно всегда получить помощь со стороны более опытных пользователей, например членов группы TeamB. Полный список списков рассылки, связанных с C++Builder и другими продуктами Borland, приводится в приложении А.

Ниже приводятся наиболее общие принципы оптимизации структуры программы.

- Упростите структуру. Не разбивайте приложение на слишком большое количество модулей. Не следует также чрезмерно нормализовать базу данных.
- Упростите план выполнения. Избегайте цепочных вызовов распределенных объектов.
- Исключите медленно работающие технологии. Просто выберите наилучшую из них.
- Используйте эффективные инструменты сторонних разработчиков для создания отчета, сжатия и шифрования данных и других потребностей.
- Сократите до минимума использование графики и визуальных элементов управления, особенно при очень частом их обновлении. Рассмотрите возможность отказа от их использования или скрытия при обновлении и отображения по окончании обновления.
- Длительные задачи старайтесь выполнять в дополнительном потоке.
- Сократите до минимума сетевой трафик и операции доступа к жесткому диску. Рассмотрите возможность использования клиентских наборов данных, а также чтения и записи данных в нескольких больших фрагментах вместо большого числа малых фрагментов. В сети или других коммуникационных соединениях с большим временем задержки пересылайте сразу несколько запросов и ответов на них.
- Эффективно проектируйте приложения для многопроцессорных компьютеров. Распределяйте рабочую нагрузку равномерно между процессорами и сокращайте интенсивность взаимодействия процессов.

- Сокращайте использование оперативной памяти. Многопоточные приложения и приложения, которые выполняются одновременно с другими приложениями, должны аккуратно использовать оперативную память, чтобы ОС могла эффективно распределять ее.

Универсального метода для оптимизации макета приложения не существует. Если вы сомневаетесь в этом, спросите более опытных коллег. Группы пользователей в сети Internet часто объединяют профессионалов и любителей с разной подготовкой и опытом работы.

Приложение Crozzle Solver имеет очень простую структуру. Большинство перечисленных методов оптимизации не имеет никакого отношения к нему. Однако один компонент структуры, а именно решетка `TDrawGrid`, все-таки оптимизирована. По умолчанию текущее состояние кроссворда-головоломки не обновляется в решетке `TDrawGrid` во время поиска решения. Однако пользователь может его просматривать с помощью команд меню View.

Еще один способ оптимизации структуры заключается в небольшом улучшении плана выполнения за счет перестройки иерархии функций. В текущем состоянии, т.е. когда нажата кнопка Solve, вызывается функция `SolveCrozzle()`. Она, в свою очередь, вызывает функцию `SolveWord()` для генерации решений. `SolveWord()` — общая функция, которая вызывается для поиска решения, начиная с исходного или текущего состояния решетки. Она содержит следующий код.

```
if (WordsPlaced.NumPlaced == 0) {
    SolveFirstWord();
    return(true);
}
```

Вызов функции `SolveFirstWord()` необходим только на самом верхнем уровне рекурсии, который определяется по результатам выполнения условия `if` (т.е. в случае отсутствия слов в решетке кроссворда). Функция `SolveWord()` вызывается рекурсивно на всех уровнях, что означает выполнение оператора `if` на всех уровнях. Чтобы избежать этого, функцию `SolveFirstWord()` можно вызвать вне рекурсивного цикла и поместить ее в исходной вызывающей функции `SolveCrozzle()`. Теперь простой вызов функции `SolveWord()` из функции `SolveCrozzle()` будет выглядеть так, как показано ниже.

```
if (SolveFromBlank) {
    SolveFirstWord();
} else {
    SolveWord();
}
```

Теперь функция `SolveFirstWord()` вызывается только в том случае, если решетка кроссворда пуста, В противном случае вызывается функция `SolveWord()` для работы на основе исходного решения. Это позволяет избежать оценки условия `if` при каждом вызове функции `SolveWord()`, т.е. более 6 000 000 000 раз в примере с 11 словами!

Теперь мы получим следующие результаты.

Текущее время: 49525 секунд. **Выигрыш:** 3,5% (в общем скорость повышена в 1,61 раз)

Не очень существенное, но легко достижимое улучшение.

Выбор алгоритмов

Алгоритмы обычно позволяют более существенно повысить скорость работы приложения. Для каждой задачи существует множество способов получения решения, поэтому в каждом отдельном случае важно выбрать соответствующий алгоритм. Для решения стандартных задач обычно существует несколько достаточно эффективных и широко известных алгоритмов.

Алгоритм — это просто метод решения задачи, который имеет пять основных характеристик.

- Ограниченность. Он прекращает свою работу в некоторой точке.
- Корректность. Он находит правильное решение задачи.
- Предсказуемость. Он всегда выполняет одинаковые действия для тех же самых входных данных.
- Конечность. Он может быть описан с помощью конечного количества шагов.
- Определенность. Каждый шаг алгоритма имеет определенное значение.

Алгоритмы решения отдельных задач часто могут существенно повысить скорость работы приложения. Например, это имеет особо важное значение при создании игр, для которых характерна конкуренция между разными механизмами обработки 3D-графики. В некоторых случаях скорость работы приложения или избранный алгоритм может зависеть от входных данных.

Еще один пример разницы в скорости выполнения приложения связан с алгоритмами сортировки. Три наиболее известных алгоритма включают пузырьковую сортировку (bubble sort), сортировку методом выбора (selection sort) и быструю сортировку (quick sort). Познакомиться с их работой можно, компилируя и запуская приложение Threads в каталоге `Examples`, который находится в основном каталоге C++Builder.

Скорость выполнения этих трех алгоритмов сортировки зависит от количества сортируемых элементов и начального расположения элементов. При использовании набора из 115 элементов, которые генерирует приложение Threads, самую высокую скорость будет иметь быстрая сортировка, а самую низкую — пузырьковая сортировка. Однако при сортировке менее чем 10 элементов пузырьковая сортировка выполняется быстрее всего, а быстрая сортировка — медленнее всего. Если 115 элементов уже находятся в некотором порядке, то пузырьковая сортировка выполняется быстрее всех, а быстрая сортировка — медленнее всех. А при обратном порядке сортировки быстрее всех будет выборочная сортировка, а медленнее всех — пузырьковая сортировка.

Если скорость работы алгоритма в вашем приложении существенно меняется при использовании разных вводных данных, которые могут меняться в широких пределах, рассмотрите возможность включения двух или более алгоритмов в приложение для выполнения одной и той же задачи с помощью наиболее оптимального алгоритма.

Для описания работы алгоритмов часто используется математическое обозначение на основе символа O (асимптотическая сложность). Это обозначение можно рассматривать как сокращение при описании времени выполнения алгоритма по отношению к размеру задачи. Например, обозначение O(N) описывает линейную зависимость времени выполнения от размера задачи, т.е. для выполнения задачи с 1, 2 и 3 элементами, может потребоваться соответственно 1, 2 или 3 секунды. А при использовании алгоритма $O(N^2)$ время, необходимое для решения задачи, возрастает пропорционально квадрату размера задачи, т.е. для выполнения задачи с 1, 2 и 3 элементами, может потребоваться соответственно 1, 4 и 9 секунды. Алгоритм $O(2^N)$ характеризуется чрезвычайно большой зависимостью времени выполнения от размера задачи и обычно рассматривается как не имеющий полного решения.

Пузырьковая сортировка имеет сложность $O(N^2)$, а потому этот алгоритм следует использовать только для очень малых значений N. Для малых значений N существуют более эффективные алгоритмы. Алгоритм быстрой сортировки имеет сложность O(log N) и является одним из самых быстрых алгоритмов сортировки для больших N. Алгоритм, используемый в нашем примере приложения для решения кроссворда, обладает сложностью $O(2^N)$, т.е. он совсем неэффективен. Однако алгоритм $O(N^2)$ иногда можно заменить алгоритмом O(N), а алгоритм O(N) — алгоритмом O(log N).

Следует заметить, что два разных алгоритма необязательно приводят к получению одинакового результата. Например, более быстрый алгоритм может обладать меньшей точностью или повышенной частотой появления ошибок. Много лет назад автору удалось создать алго-

ритм рисования окружности, который выполнялся гораздо быстрее, чем стандартный системный алгоритм рисования малых и средних окружностей. В нем функция `sqrt()` используется для рисования точек по окружности вместо рисования N-стороннего многоугольника. Он позволяет получить более точно нарисованную окружность, но очень медленно рисует большие окружности.

Ниже приводится список нескольких очень полезных источников с описанием многочисленных алгоритмов.

- *Numerical Recipes in C* — книга в формате PDF, которую можно найти по адресу `http://www.ulib.org/webRoot/Books/Numerical_Recipes/`.
- *Stony Brook Algorithm Repository* содержит решения многих различных задач с исходным кодом по адресу `http://www.cs.sunysb.edu/~algorith/`.
- *Object-Oriented Numerics* (`http://oonumerics.org/`) содержит библиотеку Blitz++, которая включает классы C++, предназначенные для научных вычислений. Это высокоэффективная библиотека, в которой используются шаблоны, плотные массивы и векторы, генераторы случайных чисел, а также малые векторы и матрицы. Это открытый источник, который содержит множество ссылок на другие библиотеки.
- Алгоритмы для игр и графического программирования можно найти по адресам `http://www.gamedev.net/` и `http://www.magic-software.com/`, а также в группе новостей `comp.graphics.algorithms`.
- В журнале *Dr. Dobb's Journal* (`http://www.ddj.com/`) иногда публикуются статьи об алгоритмах. Например, апрельский номер 2000 года полностью посвящен алгоритмам. В сети Internet можно приобрести отдельные выпуски или компакт-диск со всеми выпусками за последние 11 лет по цене 80 долларов США.
- *Introduction to Algorithms by Cormen*, Leiserson & Rivest. ISBN 0262031418. 1990.
- *Algorithms in C++*, Sedgewick. ISBN 0201510596. 1992.
- *The Art of Computer Programming: Sorting and Searching*, D. Knuth. ISBN 0201896850. 1998.

Для сортировки, сжатия и хранения данных, работы с графикой создано огромное количество алгоритмов, которые вы легко можете использовать в приложении.

Улучшение алгоритмов

При использовании нестандартного и созданного вами алгоритма, вероятно, придется самостоятельно улучшать его. Однако иногда его можно улучшить на основе уже существующего алгоритма, особенно если исходный алгоритм имеет очень сложную структуру.

В библиотеке Standard Template Library (STL) можно найти шаблоны для оптимальной работы с массивами и выполнения сортировки специальных типов данных. Разные контейнеры могут обладать различной производительностью. В документации библиотеки STL и справке C++Builder можно найти несколько советов по оптимальному использованию библиотеки STL.

Прекрасный Web-сайт с информацией о времени выполнения и методах оптимизации компонентов и алгоритмов библиотеки STL можно найти по адресу `http://www.tantalon.com/pete/cppopt/main.htm`.

Общий метод ускорения алгоритма заключается в использовании справочных таблиц или просто справочных значений вместо математических или логических вычислений. Эти таблицы или отдельные значения иногда создаются до выполнения программы и закодированы в виде статического массива в исходном коде программы. Они также могут быть созданы во время выполнения приложения, но перед выполнением алгоритма или даже в самом алгоритме.

На заметку Для хранения предварительно вычисленных таблиц или значений необходимо выделить дополнительную память, причем иногда размер этих таблиц или значений может быть достаточно большим. Это следует иметь в виду при выборе компромиссного варианта между увеличением скорости и уменьшением размера.

При создании справочных таблиц или значений во время выполнения приложения необходимо учесть расходы времени на их создание и поддержание во время работы алгоритма. Самый известный пример использования таблицы — это алгоритм циклического избыточностного контроля (Cyclic Redundancy Check — CRC). CRC-контроль часто используется при передаче данных, чтобы проверить, получены ли они в том же корректном состоянии, в котором отправлялись. Тот, кто посылает данные, вычисляет контрольную сумму CRC и прикладывает ее к данным. Получатель также вычисляет контрольную сумму CRC и сравнивает ее с контрольной суммой CRC, которая послана с данными. При их совпадении полученные данные считаются корректными.

В медленном алгоритме контрольная сумма CRC вычисляется за счет итеративного перебора всех байтов данных и выполнения полиномиального деления по модулю два. Этот алгоритм также долго выполняется, как и формулируется. В *быстром табличном методе* (*faster table method*) используется таблица из 256 значений для каждого ASCII-символа, которая генерируется либо предварительно и инициализируется в исходном коде, либо — во время выполнения CRC-алгоритма. Затем подстановка значения из таблицы для каждого символа данных позволяет избежать многочисленных вычислений на каждом этапе. Конечно, некоторые вычисления придется выполнить, но они будут существенно сокращены. Это похоже на получение произведения сомножителей с помощью таблицы умножения, а не вычисления с помощью карандаша и бумаги.

Другими кандидатами для использования таблицы с подстановочными значениями являются большие выражения с командой `switch` или многочисленные выражения с командой `if else`. При выполнении только статических вычислений или вызове функций в разных выражениях `case` или `if` следует рассмотреть возможность сохранения предварительно вычисленных значений или указателей функций в таблице. Например, так, как показано ниже.

```
switch(NumSides){
    case 1: Circle(x, y, r); break;
    case 2: PolyError(x, y, r); break;
    case 3: Triangle(x, y,r); break;
    case 4: Square(x, y, r); break;
    case 5: Pentagon(x, y, r); break;
}
```

Приведенное выражение на основе оператора `switch` можно заменить следующей таблицей.

```
void (*Polyfunctions[5])(int, int, int) = {
    Circle, PolyError, Triangle, Square, Pentagon
};
    //Затем в некоторой функции.
    PolyFunctions[NumSides](x, y, r);
```

Выражение на основе оператора `if` также можно заменить простой подстановочной таблицей с помощью значений, которые представляют каждую комбинацию логических условий. Например, в выражении `if (A && B)`, `A` может быть равно `true` или `false`, `B` также может быть равно `true` или `false`. При этом существует всего четыре их комбинации. Используя побитовую математику, обозначим бинарным значением 00 ситуацию, когда `A` и `B` равны `false`, значением 01 — ситуацию, когда `B` равно `true`, значением 10 — ситуацию, когда `A`

равно true, и значением 11 — ситуацию, когда A и B равны true. Эти бинарные значения эквивалентны десятичным значениям 0, 1, 2 и 3, которые используются для индексирования подстановочной таблицы.

Основной алгоритм поиска решения алгоритма включает несколько методов оптимизации, в которых используются хранимые табличные значения, для повторного вычисления которых в обычных условиях потребуются значительные усилия. Некоторые из них представлены в следующих абзацах.

Для размещения слова в решетке механизм поиска решения должен перебрать все слова в заданном списке слов. При этом перебираются все буквы в каждом слове и определяется место в решетке, где эта буква появится (если появится вообще). Если она появится в решетке, то потребуется выполнить ряд тестов для определения ситуации пересечения с другим словом в этом месте, а также, тогда будет ли это слово расположено горизонтально или вертикально. Для этого применяются два метода оптимизации.

Во-первых, для хранения информации о местах расположения в решетке букв алфавита используется массив. При размещении слова в решетке, информация о расположении каждой его буквы заносится в этот массив. Это позволяет сэкономить огромное количество времени, поскольку не требуется сканировать 10 строк из 15 клеток каждый раз при поиске места пересечения с текущим словом.

Во-вторых, при размещении слова в решетке информация о его ориентации хранится в каждой клетке которую, занимает это слово, в виде двух логических значений — HorizWord и VertWord. В случае пересечения слов в этой клетке эти значения равны true. Это тоже экономит время, так как позволяет определить ориентацию слова (горизонтальная или вертикальная) в клетке. Визуально это можно определить довольно легко, но очень сложно с помощью программных средств, так как для этого потребуется выполнить несколько проверок содержимого клеток с буквами и границами решетки, за которые не должно выходить слово.

Еще один метод оптимизации на основе хранимых значений используется для оптимизации вычисления текущего количества призовых очков, т.е. счета. Согласно результатам профилирования функция CalcScore() содержит одно из таких уязвимых мест. Вместо сканирования решетки для поиска клеток с пересекающимися словами и сложения призовых очков для таких клеток с общим количеством призовых очков (т.е. общим счетом), счет для каждой клетки задается во время установления пересечения. Счет хранится вместе с другой информацией клетки в решетке кроссворда. Все, что нужно теперь для вычисления общего счета, — вычислить общий счет для всех клеток решетки (с учетом 4 ограничений слов в решетке), что позволяет избежать нескольких проверок значений в клетках. В итоге это ускоряет работу программы на 5%.

На основе результатов профилирования известно, что уязвимые места производительности находятся в разных функциях, а именно: в функции поиска места, проверки места, размещения слова и удаления слова. Позднее мы подробнее рассмотрим методы оптимизации их кода, но если удастся сократить общую нагрузку на эти функции, то это тоже позволит добиться существенного прогресса. Рассмотрим теперь другие методы оптимизации механизма решения кроссворда.

Одним из наиболее крупных недостатков текущего алгоритма решения кроссворда является то, что он генерирует повторяющиеся комбинации пересекающихся слов. Если в заданном списке имеются слова JAR и ROD, то расположенное горизонтально слово JAR пересечет расположенное вертикально слово ROD на букве R. В другом месте расположенное вертикально слово ROD пересекает расположенное горизонтально слово JAR на букве R с получением того же результата!

Рассмотрим способ поиска повторяющихся решений. Он имеет достаточно сложное описание, но основан на двух простых основных правилах, в зависимости от места его применения: на основе частичного решения или на основе исходной пустой решетки.

- Способ на основе частичного решения. Слово может быть размещено в решетке только в том случае, если оно пересекается с самым последним словом, которое расположено в решетке и находится ниже в списке слов, или если оно пересекается с любым словом, которое находится ниже в списке слов. Например, для списка слов `CAB`, `BIT`, `CAT` и частичного решения с уже размещенным словом `CAB` слово `CAT` может пересекаться со словом `CAB` на букве `C`. Однако при этом следует учитывать пересечение слова `BIT` только на букве `T` со словом `CAT`, а не на букве `B` со словом `CAB`, потому что эта комбинация ранее уже была решена.
- Способ на основе исходной пустой решетки. Здесь снова применяется предыдущее правило, а затем продолжается исключение повторяющихся решений. После размещения первого слова не разрешается размещение слов, которые располагаются выше него в заданном списке слов. В этом нет никакой необходимости, потому что ранее уже рассматривались все комбинации с этим словом. Например, для заданного списка слов (`CAB`, `BIT`, `CAT`) сначала в решетке размещается слово `CAB`, а затем выполняется поиск его пересечений со словами `BIT` и `CAT`. Позднее, при удалении слова `CAB` и размещении первым слова `BIT`, не потребуется размещать слово `CAB`, поскольку ранее были найдены все комбинации с этим словом.

Для реализации этого алгоритма необходимо отслеживать слово в заданном списке слов, которое находится выше всех в списке и уже было размещено в решетке, а также индексировать слова, с которыми оно пересекается. Если поиск решения проводится на основе пустой решетки, всякий раз при попытке размещения слова в решетке можно пропустить все слова в заданном списке слов, которые находятся в списке до первого размещенного в решетке слова. Первое размещаемое в решетке слово из заданного списка слов можно проиндексировать вне рекурсивного цикла в функциях `SolveCrozzle()` и `SolveFirstWord()`.

Задачи поиска и проверки повторяющихся решений имеют очень сложное решение. Для проверки правильности используемых алгоритмов требуется выполнить специальное тестирование, чтобы гарантировать правильность определения всех возможных комбинаций.

Этот новый метод исключения повторяющихся комбинаций обладает следующими характеристиками.

Текущее время выполнения программы: 24,33 секунды. **Выигрыш:** 203600% (общее ускорение 3270 раз)

Ого! Неужели это правда? Несомненно, так как на перебор повторяющихся комбинаций и рассмотрение всех вариантов расположения слов тратится очень много времени.

Еще одно небольшое улучшение алгоритма можно получить с помощью следующего метода. При размещении стандартного слова и определении наличия пересечения в клетке с буквой (т.е. при определении того, проходит ли через эту клетку решетки одновременно горизонтальное и вертикальное слово) алгоритм выполняет проверку логических значений `HorizWord` и `VertWord` в этой клетке. Это проще выполнить с помощью проверки счета в этой клетке.

Это позволяет добиться следующих результатов.

Текущее время выполнения программы: 23,68 секунд. **Выигрыш:** 2,7% (общее ускорение 3360 раз)

Заключительное замечание по поводу улучшения алгоритмов: следует рассмотреть все возможные алгоритмы решения данной задачи (см. список рекомендуемой литературы в разделе о выборе оптимального алгоритма выше в этой главе). Проанализируйте способы выполнения этой задачи разными алгоритмами, потому что иногда решение может иметь очень неожиданный вид.

Методы настройки кода

Существует очень много методов ускорения кода на языке C++. Эффективный, на первый взгляд, код на языке C++ не обязательно обозначает эффективный машинный код. Справедливо также то, что многие улучшения кода на языке C++ приведут к снижению его читабельности и усложнят его сопровождение, что следует учитывать при формулировании приоритетов во время создания приложения.

Современные процессоры

Современные процессоры имеют очень сложную структуру, а потому столь же сложную структуру имеет выполняемый на них машинный код. За исключением встроенного ассемблера, обычно не существует прямого метода воздействия на машинный код, который генерируется на основе кода на языке C++. Однако существует несколько других методов, которые позволяют повысить производительность кода. Эти методы непосредственно зависят от типа процессора.

В следующих разделах рассматриваются некоторые особенности устройства процессора Pentium фирмы Intel, в частности процессора Pentium II. Большинство этих особенностей устройства в равной степени относятся к процессорам K-серии фирмы AMD.

Прогнозирование ветвления

Наиболее важной чертой оптимизации процессора Pentium II является возможность статического и динамического *прогнозирования ветвления* (*branch prediction*). Существует три типа ветвления: безусловное (unconditional), прямое условное (forward conditional) и обратное условное (back-ward conditional). В языке C++ безусловное ветвление генерируется непосредственно на основе операторов `continue`, `break` и `goto`, а также косвенно на основе операторов `if`, `?:` и `switch`. Прямое условное ветвление генерируется на основе операторов `if`, `?:`, `switch` и `while`, а обратное ветвление — непосредственно на основе операторов `for`, `while` и `do`.

Рассмотрим, например, код (плохо организованный) из листинга 6.4, который вызывает другой алгоритм оптимизации рисования маленьких окружностей.

Листинг 6.4. Пример неудачного ветвления

```
if (Radius > MaxRadius) {
    return(false);
}

if (Radius < 50 && Filled) {
    CircleSmallAlgorithmFilled(X, Y, Radius);
}

if (Radius < 50 && !Filled) {
    CircleSmallAlgorithmOpen(X, Y, Radius);
}

if (Radius >= 50 && Filled) {
    CircleLargeAlgorithmFilled(X, Y, Radius);
}

if (Radius >= 50 && !Filled){
    CircleLargeAlgorithmOpen(X, Y, Radius);
}

return (true);
```

Машинный код, сгенерированный компилятором для первых двух операторов `if`, на самом деле аналогичен коду в листинге 6.5.

Листинг 6.5. Генерируемый компилятором машинный код для примера с неудачным ветвлением

```
if (Radius <= MaxRadius) {
    goto ifb;
}
return(false);

ifb:
if (Radius >= 50) {
    goto ifc;
}

if (!Filled){
    goto ifc;
}

CircleSmallAlgorithmFilled(X, Y, Radius);

ifc:

//И т.д. Остальные операторы if содержат аналогичный код.
```

Обратите внимание, что в сгенерированном компилятором коде (в листинге 6.5) операторы `if` генерируют противоположные логические значения. Каждое условие в операторе `if` представляет собой прямое условное ветвление. При использовании операторов `else` потребуется использовать безусловное ветвление в конце каждого блока `if` или `else`, за исключением последнего — для выхода из блока после выполнения всех операторов в нем и пропуска кода во всех следующих блоках `else`.

При динамическом предсказании ветвления процессор хранит информацию о последних четырех направлениях ветвления или продолжения работы для всех предыдущих 512 ветвлений. На основании этих данных процессор выбирает направление ветвления для следующего раза и с упреждением — нужные инструкции. При этом могут идентифицироваться конфигурации до четырех изменений направления, например ветвление с четными прыжками или ветвление с двойным прыжком, продолжением работы, еще одним прыжком, с повторением этой конфигурации.

Если тип ветвления не записан (как правило, потому, что его код не выполнялся до этого), то используется статическое прогнозирование ветвления. При этом безусловное и обратное условное ветвление прогнозируются (прыжки), а прямое условное ветвление не прогнозируется (продолжение работы).

Неправильное статическое и динамическое прогнозирование связано с большими затратами времени, особенно, если оно происходит в цикле. Неправильное прогнозирование означает, что предварительно выбранные инструкции должны быть удалены из конвейера выполнения, а вместо них выбраны правильные инструкции.

Для повышения эффективности ветвления в код C++ можно ввести следующие два основных улучшения. Во-первых, следует сократить количество ветвлений. Это позволит снизить вероятность неправильного прогнозирования и сократить объем хранимой информации о преды-

дущих ветвлениях. Во-вторых, следует использовать операторы `if else` (особенно вложенные операторы `if`) вместо нескольких операторов `if` на одном уровне расположения блоков.

В предыдущем примере кода для рисования окружностей, если значение радиуса `Radius` превышает максимальное значение `MaxRadius`, выполняется только одно прямое условное ветвление. В противном случае потребуется выполнить семь прямых условных ветвлений для перебора всех остальных вариантов. В листинге 6.6 показан способ изменения этого кода для снижения количества ветвлений.

Листинг 6.6. Улучшенная структура ветвлений

```
if (Radius > MaxRadius) {
    return(false);
} else if (Radius < 50) {
    if (Filled) {
        CircleSmallAlgorithmFilled(X, Y, Radius);
    } else {
        CircleSmallAlgorithmOpen(X, Y, Radius);
    }
} else {
    if (Filled) {
        CircleLargeAlgorithmFilled(X, Y, Radius);
    } else {
        CircleLargeAlgorithmOpen(X, Y, Radius);
    }
}

return (true);
```

В листинге 6.6 также выполняется только одно прямое условное ветвление, если значение радиуса `Radius` превышает максимальное значение `MaxRadius`. Но теперь для доступа к трем из четырех комбинаций потребуется выполнить только прямое условное и одно безусловное ветвление. Для доступа к последней комбинации, `Radius >= 50 && !Filled`, придется выполнить только три прямых условных ветвления. Безусловное ветвление теперь не потребуется, потому что план выполнения просто заменяется кодом выполнения единого блока `if else`. К сожалению, это управление нельзя применить для блоков `if else`, заключенных в операторе `switch`.

Еще один способ повышения эффективности ветвлений заключается в улучшении прогнозирования ветвления. При первом выполнении этого кода выполняется статическое прогнозирование ветвления. Первая часть генерированного компилятором кода содержит прямое условное ветвление.

Допустим, радиус 95% рисуемых окружностей не превышает максимального радиуса, в таком случае этот код не будет храниться в списке ветвлений из-за большого количества ветвлений, которые произошли со времени последнего его выполнения. В таких условиях прогнозирование ветвления будет неправильным. Неправильно будет спрогнозировано ветвление также в том случае, если радиус большинства окружностей больше или равен 50.

Код на языке C++ следует упорядочить таким образом, чтобы наиболее часто используемые условия располагались в первом блоке `if`, менее часто используемые условия — в блоке `else` и т.д. В листинге 6.7 показан наиболее эффективный вариант кода, при создании которого предполагается, что большинство окружностей имеет большой, но правильный радиус.

Листинг 6.7. Перестроенное ветвление для эффективного использования часто используемых условий

```
if (Radius <= MaxRadius) {
    if (Radius >= 50) {
        if (Filled) {
            CircleSmallAlgorithmFilled(X, Y, Radius);
        } else {
            CircleSmallAlgorithmOpen(X, Y, Radius);
        }
    } else {
        if (Filled) {
            CircleLargeAlgorithmFilled(X, Y, Radius);
        } else {
            CircleLargeAlgorithmOpen(X, Y, Radius);
        }
    }
} else {
    return(false);
}

return (true);
```

Еще один способ повышения эффективности вычисления логических выражений основан на использовании битовых операторов `|` и `&` вместо логических операторов `||` и `&&`. Например код `if ((A && B) || C)` можно заменить кодом `if ((A & B) | C)`. При использовании вместо логических битовых операторов все они преобразуются в одно битовое вычисление и для него требуется выполнить только одно прямое условное ветвление. В одних случаях битовое вычисление выполняется быстрее, чем несколько ветвлений с использованием логических операторов, а в других — медленнее. Для определения этого обязательно нужно выполнить профилирование или хронометраж соответствующего фрагмента кода.

Кэш-память

Процессор имеет кэш-память первого уровня, в которой 16 КБайт используется для хранения кода и 16 КБайт — для хранения данных. Кэш-память первого уровня (level-1 cache) — это область памяти процессора, которая используется для выполнения всех инструкций машинного кода и доступа к оперативной памяти. Для загрузки выполняемого кода или обрабатываемых данных в кэш-память необходимо потратить некоторое время, которое часто превышает время его выполнения.

При программировании цикла рекомендуется делать его максимально коротким, чтобы он мог целиком поместиться в кэш-памяти. Первая попытка выполнения этого цикла будет немного медленнее из-за загрузки в кэш-память, но последующие попытки будут выполняться максимально быстро, если этот код еще находится в кэш-памяти.

Настройка кода

Рассмотрим несколько методов настройки кода с помощью небольших фрагментов кода. В конце этого раздела они будут продемонстрированы на примере приложения Crozzle Solver.

Для этого используются следующие четыре основных принципа.

- Удалить избыточный код. В больших приложениях при реализации новой структуры и кода некоторые фрагменты кода могут устареть. Новая, более эффективная функция, обычно может заменить более старую и медленную.
- Не используйте ключевое слово `register`. Используя автоматическое размещение переменных в регистрах процессора, можно предоставить возможность компилятору принять наиболее эффективное решение о том, нужно ли размещать переменные в регистре и какие именно переменные.
- Используйте ключевое слово `const` для переменных, параметров функций и членов-функций, значение которых будет постоянным. Это также позволяет компилятору автоматически выполнить некоторые специальные процедуры оптимизации.
- Не нарушайте систему типов. Необязательное смешение разных типов данных, например присвоение значения типа `double` значению типа `long`, вызовет преобразование типов данных, которое замедлит работу программы и не позволит компилятору применить некоторые способы оптимизации.

Рассмотрим теперь несколько специальных методов тонкой настройки кода.

Встроенные функции

Стандартные вызовы функций включают дополнительные затраты времени и ресурсов на уровне машинного кода для передачи и возвращения параметров либо из-за вставки и выталкивания значений из стека, либо вследствие заполнения специальных регистров процессора и для вызова и возврата функции. Для очень малых функций, например функций доступа к членам класса, эти дополнительные затраты могут быть больше, чем затраты на выполнение самого кода функции.

На заметку

Более подробная информация о вызовах функций и указателях стека на уровне машинного кода приводится в главе 4 о вызовах процедур, прерываниях и исключительных ситуациях в томе 1 руководства разработчика Intel Architecture Software Developer's Manual. Его можно скопировать в формате PDF с Web-сайта Intel по адресу `http://developer.intel.com/design/ processor/`. Выберите тип процессора, например Pentium II, а затем войдите в раздел `Manuals`.

Для исключения или сокращения дополнительных затрат времени и ресурсов могут использоваться встроенные функции (inline functions). Встроенная функция с программной точки зрения действует точно так же, как и обычная функция. Однако после компиляции кода копия функции встраивается непосредственно в место вызова функции. Это также позволяет оптимизатору более эффективно выполнить свою работу.

Встроенные функции имеют два основных недостатка. Во-первых, они увеличивают размер кода, особенно, если встроенная функция очень велика и вызывается во многих местах. Во-вторых, если вызов функции располагается внутри цикла, то увеличенный размер кода снижает шансы того, что весь цикл целиком поместится в кэш-памяти (о чем уже подробно говорилось выше в этой главе).

Встроенные функции определяются явно с помощью указания ключевого слова `inline` для обычных функций и методов класса, определенных вне тела класса, а также неявно (автоматически) для методов класса, определенных в теле класса. Пример использования этих функций показан в листинге 6.8.

Листинг 6.8. Пример использования встроенных и обычных функций

```
class TMyClass
{
private:
    int FValue;
public:
    void SetValue(int NewValue) { FValue = NewValue; }
    int GetValue() const { return(FValue); }
    void DoubleIt();
    void HalveIt();
};

inline void TMyClass::DoubleIt()
{
    FValue *= 2;
}

void TMyClass::HalveIt()
{
    FValue /= 2;
}

inline int Negate(int InitValue)
{
    return(-InitValue);
}

void SomeFunction()
{
    TMyClass Abc;
    int NegNewVal;

    Abc.SetValue(10);
    Abc.DoubleIt();
    NegNewVal = Negate(Abc.GetValue());
}
```

В листинге 6.8 функции `TMyClass::SetValue()`, `TMyClass::GetValue()`, `TMyClass::DoubleIt()` и `Negate()` — встроенные, а функции `TMyClass::HalveIt()` и `SomeFunction()` — нет.

Экспериментируя со встроенными функциями, учтите, что по умолчанию встроенные функции отменены во время отладки и вызываются как обычные функции. Чтобы разобраться в принципе их работы, этот режим можно отменить в окне процессора CPU, сняв флажок параметра Disable Inline Expansions (Отменить расширение встроенных функций) во вкладке Compiler диалогового окна Project Options.

Для расширения встроенных функций они должны быть определены ранее функций, которые их вызывают. Если бы функция `Negate()` в листинге 6.8 была определена после функции `SomeFunction()`, то она не смогла бы быть расширена.

Не все функции могут быть встроенными, особенно при обработке исключительных ситуаций. Функция с заданной спецификацией исключительной ситуации, которая принимает по значению класс или возвращает по значению класс с деструктором, не может быть встроенной.

Чтобы найти оптимальные варианты использования встроенных функций, в приложении Crozzle Solver придется прибегнуть к методу проб и ошибок. В результате такой проверки можно убедиться, что функция `CanPlace()` позволяет получить наилучший результат.

Текущее время выполнения программы: 23,46 секунды. **Выигрыш**: 0,9% (общее ускорение 3391 раз)

Несмотря на присутствие большого количества вызовов функции, большое значение также имеют другие факторы: кэширование кода и то, что по умолчанию используется соглашение о вызовах для регистров (параметры компилятора). Еще раз подчеркнем, что встроенными лучше всего определять малые функции.

Исключение временных объектов и переменных

На создание и удаление временных объектов и переменных придется затратить некоторое время, особенно для объекта, который содержит сложный код конструктора и деструктора. По возможности их использование следует сократить или полностью исключить.

Такие объекты, строки и простые переменные, как `int`, `long` и `double`, следует объявлять в максимально малой области действия. В листинге 6.9 приведен пример создания трех временных объектов: `Obj`, `Name` и `Length`.

Листинг 6.9. Пример создания трех временных объектов

```
void SomeFunc(bool State)
{
    TComplexObj Obj;
    String Name;
    int Length;
    if (State) {
        Name = "This is a string.";
        Length = Name.Length();
        DoSomething(Name,Length);
    } else {
        Obj.Weight = 5.2;
        Obj.Speed = 0;
        Obj.Acceleration = 1.5;
        Obj.DisplayForce();
    }
}
```

Сократить время, затрачиваемое на создание временных переменных, можно с помощью локализации области действия или их полного исключения. В листинге 6.10 приведен пример использования таких мер.

Листинг 6.10. Пример локализации области действия временных переменных

```
void SomeFunc(bool State)
{
    if (State) {
    String Name;
    Name = "This is a string.";
    DoSomething(Name, Name.Length());
    } else {
        TComplexObj Obj;
        Obj.Weight = 5.2;
        Obj.Speed = 0;
        Obj.Acceleration = 1.5;
        Obj.DisplayForce();
    }
}
```

Для ускорения создания временных переменных следует инициализировать объекты во время их объявления. В листинге 6.11 показан способ улучшения кода из листинга 6.10 на основе этого принципа.

Листинг 6.11. Инициализация временных переменных во время их объявления

```
void SomeFunc(bool State)
{
    if (State){
        String Name("This is a string.");
        DoSomething(Name, Name.Length());
    } else {
        TComplexObj Obj(5.2, 0, 1.5);
        Obj.DisplayForce();
    }
}
```

При выполнении операция инкремента или декремента на основе постфиксных операторов ++ и -- временная переменная создается для сохранения возвращаемого значения так, как показано в листинге 6.12.

Листинг 6.12. Создание временной переменной для оператора ++ с целью возвращения значения

```
class TMyIntClass
{
private:
    int FValue;
public:
    TMyIntClass(int Init) { FValue = Init; }
    int GetValue() { return FValue; }
    // Постфиксный оператор ++.
    int operator++(int) {
        int Tmp = FValue;
```

```
        FValue = FValue + 1;
        return(Tmp);
    }
    // Префиксный оператор ++.
    int operator++() {
        FValue = FValue + 1;
        return(FValue);
    }
};

void MyFunc()
{
    TMyIntClass A(1);
    // Используется временная переменная.
    A++;

    // Возвращаемое значение для A++ не требуется.
    // Поэтому лучше использовать префиксный оператор.
    ++A;
}
```

Если для постфиксного оператора инкремента или декремента возвращаемое значение не требуется, используйте вместо них префиксные операторы инкремента или декремента. В таком случае, временное значение использоваться не будет. В листинге 6.12 временное значение исключается за счет использования оператора `++A` вместо оператора `A++`.

Когда объект передается функции по значению, временный объект для использования в функции создается с помощью конструктора копии. Для исключения создания временного объекта предпочтительнее передавать значение по ссылке-константе (`const`). Например, для исключения создания временного объекта для функции `MyFunc(TMyClass A)` ее следовало бы определить как `MyFunc(const TMyClass &A)`. При использовании ссылки-константы временная переменная также создается, но это переменная совсем другого типа.

Последний метод исключения временного объекта называется *оптимизацией возвращаемого значения* (*return value optimization*). Если объект или переменная должны возвращаться по значению и это значение создается только для возврата, лучше всего создать его непосредственно на месте, а не создавать для хранения временный объект и возвращать его значение. Применение этого метода показано в листинге 6.13.

Листинг 6.13. Оптимизация возвращаемого значения

```
TMyClass TmpReturnFunc(bool Something)
{
    TMyClass TmpObj(0, 0);
    if (какое-то условие) {
        TMyClass Obj1(1.5, 2.2),
                 Obj2(1.8, 6.1);
        TmpObj = Obj1 + Obj2;
    }
    return(TmpObj);
}
```

```
TMyClass FastReturnFunc(bool Something)
{
    if (какое-то условие) {
        TMyClass Obj1(1.5, 2.2),
                 Obj2(1.8, 6.1);
        return(Obj1 + Obj2);
    } else {
        return(TMyClass(0, 0));
    }
}

void MyFunc()
{
    TMyClass A;

    A = TmpReturnFunc(AddThem); // Создание временной переменной
                                // в функции TmpReturnFunc
    A = FastReturnFunc(AddThem);// Возвращаемый объект создается
                                // непосредственно в ходе присвоения
                                // значения для A.
}
```

В первой функции обязательно будут использованы конструктор и деструктор объекта `TmpObj`, а во второй функции возвращаемый объект будет создан непосредственно по адресу присваивающего объекта `A`.

Инвариантные вычисления

Инвариантные (т.е. неизменные) вычисления, которые появляются в цикле, следует вынести за пределы этого цикла. В листинге 6.14 инвариантами являются оператор `if` и оператор вычисления `x + 5` в первом разделе кода, а потому по втором разделе они вынесены за пределы цикла. Во втором варианте они будут оценены только один раз вместо повторяющихся оценок при каждой итерации цикла в первом варианте.

Листинг 6.14. Вынос инвариантных вычислений за пределы цикла

```
// Медленно выполняемые циклы.
for (int i = 0 ; i < 10 ; i++) {
    if (InitializeType == Clear) {
       a[i] == 0;
       b[i] == 0;
    } else {
        a[i] == y[i];
        b[i] == x + 5;
    }
}

// Быстро выполняемые циклы.
if (InitializeType == Clear) {
    for (int i = 0 ; i < 10 ; i++){
        a[i] == 0;
```

```
        b[i] == 0;
    }
} else {
    int Total = x + 5;
    for (int i = 0 ;i < 10 ; i++) {
        a[i] == y[i];
        b[i] == Total;
    }
}
```

Инвариантные фрагменты кода, например фиксированные вычисления, инвариантные индексирования массивов и вычисления указателей, всегда рекомендуется выносить за пределы циклов.

Индексирование массивов и вычисления указателей

Если в коде содержится выражение с использованием индексирования массивов и вычисления указателей, которое используется несколько раз, рекомендуется вычислить указатель на нужный элемент данных или объект и повторно использовать этот указатель. Рассмотрим приведенный в листинге 6.15 алгоритм поиска решения кроссворда. Длинные строки в нем разбиты на несколько строк меньшего размера.

Листинг 6.15. Сложное индексирование массивов (арифметика указателей)

```
// Первая часть сложного индексирования
// из функции SolveStandardWord().
k = CrozLetterPos[Words[CurrWordNum].Letters[
    CurrLetterIdx]].NumPositions - 1;
while (k >=0 &&
       CrozLetterPos[Words[CurrWordNum].Letters[
           CurrLetterIdx]].WordPlacedIdx[k] >>=
           PlacedIdxLimit) {
    CurrX = CrozLetterPos[Words[CurrWordNum].Letters[
                CurrLetterIdx]].Position[k].x;
    CurrY = CrozLetterPos[Words[CurrWordNum].Letters[
                CurrLetterIdx]].Position[k].y;
    // Другой код.
}

// Вторая часть сложного индексирования
// из функции PlaceWord().
CrozLetterPos[Words[WordNum].Letters[LetterIdx]].Position[
              CrozLetterPos[Words[WordNum].Letters[
                  LetterIdx]].NumPositions].x = CurrX;
CrozLetterPos[Words[WordNum].Letters[LetterIdx]].Position[
              CrozLetterPos[Words[WordNum].Letters[
                  LetterIdx]].NumPositions].y = StartY;
CrozLetterPos[Words[WordNum].Letters[LetterIdx]].WordPlacedIdx[
              CrozLetterPos[Words[WordNum].Letters[
                  LetterIdx]].NumPositions] =
                  CrozGrid[StartY][CurrX].HorizPlacedIdx;
CrozLetterPos[Words[WordNum].Letters[
                  LetterIdx]].NumPositions++;
```

В этих хитросплетениях указателей не так легко разобраться! В обеих частях кода было бы лучше использовать временный указатель на нужный элемент данных, т.е. на структуру TCrozLetterPos. В листинге 6.16 показано, как изменить код, чтобы использовать временные указатели.

Листинг 6.16. Упрощенное индексирование массива (арифметика указателей)

```
// Первая часть простого индексирования
// из функции SolveStandardWord().
TCrozLetterPos *TmpPos;

    TmpPos = &CrozLetterPos[Words[CurrWordNum].Letters[
                 CurrLetterIdx]];

    k = TmpPos->NumPositions-1;
    while (k >= 0 &&
           TmpPos->WordPlacedIdx[k] >= PlacedIdxLimit) {
        CurrX = TmpPos->Position[k].x;
        CurrY = TmpPos->Position[k].y;
        // Другой код.
}

// Вторая часть простого индексирования
// из функции PlaceWord().
TCrozLetterPos *TmpPos;

    TmpPos = &CrozLetterPos[Words[WordNum].Letters[LetterIdx]];
    TmpPos->Position[TmpPos->NumPositions].x = CurrX;
    TmpPos->Position[TmpPos->NumPositions].y = StartY;
    TmpPos->WordPlacedIdx[TmpPos->NumPositions] ==
        CrozGrid[StartY][CurrX].HorizPlacedIdx;
    TmpPos->NumPositions++;
```

Эти два фрагмента кода более просты для восприятия и обладают более высокой производительностью.

Текущее время выполнения программы: 22,28 секунды. **Выигрыш**: 5,3% (общее ускорение в 3571 раз)

Таким образом, эффективность повышена на 5% только благодаря оптимизации этих двух фрагментов кода.

Обработка чисел с плавающей запятой

Современные процессоры очень быстро выполняют обработку инструкций для чисел с плавающей запятой. Функции обработки чисел с плавающей запятой в C++Builder, например cos() и exp(), обычно также выполняются очень быстро. Однако для достижения максимальной скорости при обработке чисел с плавающей запятой в C++Builder 5 рекомендуется применять новые функции fastmath.

Для их использования в модуле в него нужно включить директиву #include <fastmath.h>. При этом многие стандартные математические процедуры будут автоматически связаны с процедурами fastmath, включая функции cos(), exp(), log(), sin(), atan() и sqrt(). Для отключения этой автоматической связи и отказа от использования процедур fastmath следует явно указать директиву #define _FM_NO_REMAP перед включением заголо-

вочных файлов и в соответствующих местах применять функции _fm_<имя_функции> вместо <имя_функции>. Например, функцию cos() следует заменить функцией _fm_cos(). Более подробные сведения об их использовании можно найти в справке C++Builder.

Процедуры fastmath не выполняют проверку большинства исключительных ситуаций. Поэтому их следует использовать только в тех случаях, когда скорость работы приложения имеет очень высокий приоритет и приложение тщательно проверено.

Один из эффективных методов настройки кода вычисления чисел с плавающей запятой основан на оптимизации операций деления в цикле. Для деления X = A/B, где B является константой, следует вычислить временную переменную T = 1/B до выполнения цикла, и исходную операцию деления заменить произведением X = A * T. Дело в том, что умножение числа с плавающей запятой выполняется быстрее, чем деление.

Другие методы настройки кода

Следует упомянуть также другие методы настройки кода, которые можно применить для специальных приложений.

- Отмените оперативное определение типа (Runtime Type Identification — RTTI), если это возможно. Для этого следует воздержаться от применения ключевых слов dynamic_cast и typeid.
- По возможности избегайте использования виртуальных функций. Каждый вызов такой функции приводит к поиску и подстановке функции из таблицы.
- При использовании графики сократите до минимума операции перерисовки. Для перерисовки только отдельных частей элементов управления применяйте функцию InvalidateRect(). Для рисования объектов используйте внеэкранную область памяти, а затем копируйте ее на экран с помощью одной операции. Или скройте, а затем покажите рисуемый объект на экране с помощью методов Hide() и Show().

В функции SolveWord() периодически вызывается функция Application->ProcessMessages() для работы с пользователями и отображения предполагаемых обновлений. Это необходимо для более интенсивной работы программы с игнорированием второстепенных событий. При удалении этой части кода, можно сократить время выполнения приложения на 1,5%. Она предназначена для вмешательства в процесс поиска решения и изменения режима просмотра найденных решений.

На этом описание методов настройки кода завершается. Мы рассмотрели несколько наиболее эффективных методов повышения производительности приложения. Рассмотрим теперь другие аспекты настройки производительности приложения.

Методы настройки данных

Кроме методов настройки кода, также существуют методы настройки данных. Как уже говорилось выше, процессор имеет кэш-память для хранения кода и данных. Хранение данных в кэш-памяти имеет такое же большое значение, как и хранение кода в ней.

Последовательный доступ к оперативной памяти выполняется гораздо быстрее, чем произвольный доступ, поскольку при каждой операции чтения в оперативной памяти в кэш-память передается блок данных. При этом все данные этого блока располагаются в кэш-памяти, а потому к ним можно получить очень быстрый доступ. Поэтому связанные списки, хэш-таблицы и деревья очень неэффективны в данном случае, поскольку данные в них хранятся не в последовательном порядке.

Для использования связанных списков применяйте пользовательский управляемый пул (managed-memory pool). Выделите место для хранения нескольких узлов в связанном списке и используйте эти узлы для создаваемых и удаляемых объектов этого списка. При удалении объекта не освобождайте память; вместо этого повторно используйте ее для следующего объекта. Это улучшает размещение данных и позволяет избежать дополнительных затрат времени и ресурсов, которые связаны с выделением и освобождением памяти для каждого объекта.

Использование типов данных меньшего размера повышает вероятность того, что нужные данные находятся в кэш-памяти, так как в ней может поместиться большее количество переменных или объектов. Это особенно важно при использовании структур и объектов. При использовании многомерных массивов рекомендуется создавать их как можно меньшими.

При проектировании структур данных очень важно в максимальной степени локализовать ссылки. Например при использовании двух массивов — массива `A[]` типа `X` и массива `B[]` типа `Y` — необходимо найти наилучший способ структурирования данных, в зависимости от способа доступа к ним. Если доступ к массиву `A[]` выполняется независимо от доступа к массиву `B[]`, то их лучше объявить отдельно. Если доступ к обоим массивам выполняется практически одновременно, особенно с тем же индексом, то в таком случае лучше сгруппировать элементы массива. В листинге 6.17 продемонстрировано использование обоих методов.

Листинг 6.17. Отдельный и согласованный доступ к данным с оптимальной локализацией ссылок на них

```
// Оптимальный код для выполнения раздельного доступа.
X A[100];
Y B[100];

void SepFunc()
{
    int Sum = 0;
    for (int i = 0 ; i < 100 ; i++) {
        Sum += A[i];
    }

    // Далее в коде.
    B[Index] == Q * R;
}

// Оптимальный код для выполнения согласованного доступа.
struct {
    X A;
    Y B;
} T[100];

void SimFunc()
{
    int AveDiff = 0;
    for (int i = 0 ; i < 100 ; i++) {
    AveDiff += T.B[i] - T.A[i];
    }
    AveDiff /= 100;
}
```

Определяя такие структуры для хранения элементов массива, можно оптимизировать работу приложения Crozzle Solver за счет улучшенной локализации ссылок на структуры TCrozLetterPos и TUnsolved. Однако при этом время выполнения приложения возрастет до 22,65 секунды! При оптимизации структуры TCrozLetterPos создается 12-байтовая структура TPlacedLetter. Это не позволяет процессору воспользоваться мощными инструментами обработки бинарных чисел при вычислении расположения элементов массива.

Дополняя структуру TPlacedLetter с помощью 4-байтного числа типа int для получения в целом 16 байтов (2 в степени 4), можно сократить время выполнения до 22,04 секунды. Более того, значения типа int в структуре TPos можно также заменить 2-байтовыми значениями типа short. Это позволяет получить структуру TPlacedLetter длиной всего в 8 байт (2 в степени 3), но теперь (для получения итоговой длины, кратной 2 в некоторой степени) нужно увеличить редко используемые элементы массива TAdjLetter до 8 байт, добавляя подстановочное значение типа short.

Текущее время выполнения программы: 21,79 секунды. **Выигрыш**: 2,2% (общее ускорение в 3651 раз)

Многомерный массив T[5][10] хранится в виде 5 множеств по 10 элементов типа T. Это значит, что элементы от T[0][0] до T[0][9] хранятся в памяти рядом, за ними располагаются элементы от T[1][0] до T[1][9] и т.д. вплоть до T[4][9]. С помощью такого упорядочения определяются медленный и быстрый способы доступа ко всем элементам массива во вложенном цикле, который показан в листинге 6.18.

Листинг 6.18. Доступ к данным в многомерном массиве

```
// Медленный способ очистки массива.
// Непоследовательное индексирование.
for (a = 0 ; a < 10 ; a++) {
    for (b = 0 ; b < 5 ; b++) {
        T[b][a] = 0;
    }
}

// Быстрый способ очистки массива.
// Последовательное индексирование.
for (b = 0 ; b < 5 ; b++) {
    for (a = 0 ; a < 10 ; a++) {
        T[b][a] = 0;
    }
}
```

В нашем примере кроссворда-головоломки для обработки двумерного массива решетки применяется наиболее оптимальный метод.

Еще один метод настройки производительности кода заключается в использовании счетчика ссылок (reference counting) для исключения работы конструктора и деструктора объекта, если он используется несколько раз. При динамическом создании неизменяемых (const) объектов рекомендуется консолидировать сразу несколько экземпляров. Создайте экземпляр объекта, используя new в первый раз его использования и задавая значение 1 для переменной-счетчика ссылок. Для последующих экземпляров нужно просто вернуть указатель на тот же объект и увеличить значение счетчика ссылок. Если объект больше не нужен и счетчик ссылок достигает значения 0, он может быть удален.

Наконец, избыточные данные также снижают производительность работы приложения, как и избыточный код. Найдите и удалите их.

Ручное ассемблирование кода

Создание кода на ассемблере в настоящее время можно назвать поистине черной магией. Для процессора Pentium II имеется более 200 инструкций для работы с целыми числами, почти 100 инструкций для работы с числами с плавающей запятой и около 30 системных инструкций. Кроме "обычных" инструкций, существует около 60 SIMD-инструкций и около 60 MMX-инструкций. Познакомьтесь с такими нюансами работы процессора, как кэш-память первого и второго уровня, многочисленные конвейеры, которые поддерживают параллельное выполнение, управление страницами памяти, прогнозирование ветвления, использование специальных регистров, динамический анализ потока данных, спекулятивное выполнение и другие особенности работы процессора, и вы убедитесь в их сложности.

Несмотря на сложность процесса создания эффективного кода на ассемблере, он все же обладает существенным преимуществом: максимальным контролем. На ассемблере можно создать код, который превзойдет эквивалентный набор команд на языке C++, потому что с помощью ассемблера можно использовать специфические особенности работы процессора и машинного кода для его оптимизации на самом низком уровне. Компилятор на языке C++ может генерировать и оптимизировать только код, созданный на языке C++. Одно из основных преимуществ высокоуровневого языка, являющееся недостатком в данном случае, заключается в том, что он скрывает базовый машинный код.

Изучение принципов работы и демонстрация возможностей машинного кода процессоров x86 (для семейства процессоров Pentium) выходит за рамки этой книги. Этой теме посвящено очень много других полезных книг. Ниже перечислены некоторые из них.

- *The Art of Assembly Language Programming* (`http://webster.cs.ucr.edu/`). Интерактивная книга в формате HTML с более чем 1500 страницами, которую можно скопировать в формате PDF.
- *Intel Architecture Software Developer's Manual* (`http://developer.intel.com/ design/processor/`). Трехтомник с подробным описанием принципов работы процессоров Pentium. Его можно скопировать в разделе `Manuals` на Web-сайте фирмы Intel.
- *Iczelion's Win32 Assembly HomePage* ((`http://win32asm.cjb.net`). Прекрасный Web-сайт, посвященный ассемблированию функций Win32 с прекрасной подборкой учебных материалов и ссылок.
- *Optimizing for Pentium Microprocessors* ((`http://www.agner.org/assem/`). Web-сайт А.Фога (Agner Fog) с подробной информацией об оптимизации ассемблерного кода для процессоров Pentium.
- Список рассылки *x86 Programming Newsgroup* (`comp.lang.asm.x86`).

Для более близкого знакомства с ассемблерным кодом следует использовать все параметры отладки и отменить параметры оптимизации. Затем нужно пошагово выполнять код на уровне машинного кода с просмотром действий процессора. Для этого нужно вставить в код контрольную точку, запустить приложение и по достижении контрольной точки проследить за действиями процессора. Пошаговое выполнение (и пропуск пошагового выполнения) машинного кода организовано так же, как и пошаговое выполнение (и его пропуск) кода на языке C++. Проследите за выполнением кода на языке C++ и эквивалентного кода на ассемблере.

Добавляя параметр `-B` в разделе `CFLAG1` файла проекта и устанавливая флажок параметра Generate Listing (Генерировать листинг) и параметра Expanded Listing (Расширенный листинг) во вкладке Tasm диалогового окна Project Options, можно генерировать ассемблерный код созданного вами на языке C++ кода. При этом во время компиляции проекта для каждого файла с расширением

CPP будет создан соответствующий ему файл с расширением ASM. Еще один вариант получения ассемблерного варианта кода основан на использовании команды BCC32.EXE -I<путь_к_каталогу_с_включаемыми_файлами> -S file.cpp в режиме командной строки. При этом C++ код будет представлен в комментариях, наряду с которыми будут представлены другие полезные комментарии по поводу использования регистров, содержащие переменные из кода на языке C++.

В предыдущих разделах уже описывались специфические для процессора компоненты, например кэш-память для хранения кода и кэш-память для хранения данных, а также прогнозирование ветвления. Эти рекомендации в равной степени применимы как для кода на языке C++, так и для кода на ассемблере.

Программирование на ассемблере гораздо сложнее, чем программирование на языке C++, причем в код на ассемблере гораздо легче проникают ошибки. Кроме того, такие ошибки сложно найти. При использовании встроенного кода на ассемблере следует учесть связанные с ним расходы на сопровождение и квалифицированных программистов, которые участвуют в работе над проектом.

Код на ассемблере можно непосредственно вставить в код на языке C++, указав ключевое слово asm для отдельных команд или блока, заключенного в скобки. В коде на языке C++ могут использоваться регистры с префиксами в виде символов подчеркивания, например _EDX, _ECX, и т.д. В листинге 6.19 представлен код на языке C++, который содержит встроенный блок кода на ассемблере для вычисления факториала целого числа.

Листинг 6.19. Пример использования встроенного блока кода на ассемблере для вычисления факториала

```
int Factorial(int Value)
{
   _EDX = Value;
    asm {
        push ebp
        mov ebp,esp
        push ecx
        push edx
        mov [ebp-0x04],edx
        mov eax,0x00000001
        cmp dword ptr [ebp-0x04],0x02
        jl end
top:
        imul dword ptr [ebp-0x04]
        dec dword ptr [ebp-0x04]
        cmp dword ptr [ebp-0x04],0x02
        jnl top
end:
        pop edx
        pop ecx
        pop ebp
    }
    return (_EAX);
}
```

Как видите, встроенный код на ассемблере гораздо сложнее, чем код на языке C++. Ниже приведены рекомендации по созданию встроенного кода на ассемблере.

- При создании кода на ассемблере наибольшую выгоду можно получить, программируя его самостоятельно и независимо от особенностей языка C++. При этом не рекомендуется улучшать код на ассемблере с помощью оптимизирующего компилятора.
- Разбейте код на несколько логических частей, используйте тщательно определенные интерфейсы и подробную документацию для этого кода.
- По возможности используйте спаривание инструкций. Часто для наилучшего использования параллельного способа выполнения требуется создавать код, который выполняет действия вне логического порядка.
- Сокращайте цепочки зависимостей. Разбейте выражения, которые основаны на значении предыдущего выражения для наилучшего использования спаривания инструкций.
- При выполнении длительных инструкций, например, с числами с плавающей запятой, убедитесь, что процессор может выполнять в это время и другую работу. Спаривайте их с другим кодом.
- Всегда устанавливайте соответствие между инструкцией возврата (`RET`) и инструкцией вызова (`CALL`) для наиболее эффективного использования буфера стека возврата (`RSB`).
- Выравнивайте данные для наилучшего использования пространства в памяти и скорости выполнения инструкций. В частности, выравнивайте данные по 32-байтовым границам для эффективного использования кэш-памяти.
- При использовании встроенного кода на ассемблере параметры оптимизации компилятора применяются к функции в целом. В отличие от других оптимизаторов, оптимизатор C++Builder оптимизирует функцию, содержащую встроенный код на ассемблере. При этом нужно выполнить хронометраж кода, чтобы определить целесообразность использования встроенного кода на ассемблере, по сравнению с утратой возможности оптимизации его с помощью компилятора.

Следует отметить, что для регистров существуют определенные ограничения на использование встроенного ассемблерного кода внутри кода на языке C++. При этом не гарантируется сохранение регистровых значений при работе с разными блоками `asm`. Регистр `ECS` необходимо сохранять для функций, в отношении которых используется соглашение о вызовах `__fastcall`. Кроме того, всегда следует сохранять значения регистров `ESP` и `EBP`. Более подробную информацию в отношении других ограничений можно найти в упомянутом выше списке рекомендованной литературы.

Автор желает удачи читателю в нелегком деле ручной настройки кода на ассемблере!

Внешняя оптимизация

Помимо внутренней оптимизации, иногда требуется выполнить внешнюю. Как мы уже видели ранее, скорость выполнения алгоритмов сортировки зависит от исходного упорядочения данных. Это также справедливо для других приложений.

В приложении Crozzle Solver порядок слов в исходном списке также влияет на время выполнения приложения. В этом примере слова следует расположить начиная с самого длинного и заканчивая самым коротким. Это позволяет получить следующий результат.

Текущее время выполнения программы: 17,94 секунды. **Выигрыш**: 21,5% (общее ускорение в 4435 раз).

Хотя в примере со списком из 115 слов в файле `RunPartial.crz` решение найти нельзя, такой порядок слов позволяет добиться не оптимизации приложения по скорости, а получе-

ния наибольшего результата за счет расположения самых длинных слов в начале списка. Оптимальный порядок слов определяется на основе компромисса между счетом для всех букв слова и длиной этого слова. Кроме того, для получения приемлемого результата для длинного списка слов пользователю следует применить более эффективный способ, т.е. найти частичные решения в небольших областях решетки и постепенно создать на их основе общее решение.

Заключительные замечания по поводу оптимизации по скорости

Здесь описаны методы оптимизации структуры, алгоритмов, кода и данных, а также обсуждается использование кода на ассемблере и влияние внешней оптимизации. В приложении Crozzle Solver также можно применить несколько других способов оптимизации, которые перечислены ниже.

- Использование постоянных (`const`) параметров функций в соответствующих местах.
- Слияние логических значений `IsValid`, `IsLetter` и `IsBlank` в одной конструкции перечислимого типа для экономии пространства.
- Поиск слов на основе клеток без пересечений в решетке, а не поиск клеток на основе перебора букв в каждом слове.
- Выполнение циклической перестановки слов для поиска решения, начиная с произвольно взятого слова, а не только самого первого в списке. Это вряд ли ускорит работу, но позволит получить более высокий результат за определенный промежуток времени.
- Учет клеток с пересечениями и использование их только при вычислении счета для текущей конфигурации кроссворда.

Читателю предлагается самостоятельно исследовать эти методы оптимизации и подумать над их улучшением.

Исходное время поиска решения было равно 79 560 секунд, или 22,1 часа. Итоговое время поиска решения равно 17,94 секунды, т.е. оно уменьшено в 4435 раз.

Для алгоритмов с асимптотической сложностью $O(2^N)$ иногда можно значительно повысить скорость выполнения, как в примере с кроссвордом, хотя это достаточно редкое событие для алгоритмов общего типа. Отдельной мерой скорости поиска решения для списка слов `RunPartial.crz` в течение пяти минут с исходной пустой решеткой является результат 402 очка. Нахождение полного решения для этого списка слов дает результат 484 очка.

Другие аспекты оптимизации приложения

Помимо оптимизации по скорости, возможна оптимизация приложений по размеру, используемой оперативной памяти, скорости доступа к жесткому диску и пропускной способности в сети. Для каждого из них используются собственные методы, которые перечислены в следующих двух разделах.

Оптимизация программы по размеру

Оптимизация программы по размеру означает уменьшение размера программы. Это обычно нужно для распространения программы на ограниченных по размеру носителях на гибких дисках и компакт-дисках, в сети Internet с ограниченной скоростью обмена данными,

а также в случае ограниченного объема используемой системной оперативной памяти, что особенно важно при работе нескольких программ на одном компьютере.

Параметры компилятора C++Builder содержат специальный параметр оптимизации по размеру Size Optimization. Он позволяет добиться уменьшения размера программы с немного меньшей скоростью выполнения, чем в случае применения оптимизации по скорости Speed Optimization. На размер программы также влияют параметры отладки, которые следует отключить, а для параметра Data Alignment задать значение Byte.

Если большое значение имеет размер дистрибутива, то в первой версии программного продукта его можно распространять вместе с библиотеками и пакетами времени. Это позволит контролировать размер последующих обновлений. При этом следует использовать модульную структуру и посылать клиентам только измененные модули либо использовать программы-заплаты для внесения небольших изменений в код программы.

Если скорость работы программы не имеет очень большого значения, то ее размер можно сократить, используя алгоритмы вместо статических таблиц данных либо создавая их динамически.

При большой насыщенности графическими и звуковыми данными для уменьшения размера программы можно иногда принести в жертву их качество. Для фотографий рекомендуется использовать формат JPEG, а для простой графики — формат GIF или PNG. При этом для графических данных следует применять больший коэффициент сжатия, использовать меньше цветов, рисунки меньшего размера. А для звуковых данных рекомендуется использовать меньшее количество битов и по возможности применять моно вместо стерео.

Не забывайте о существовании таких программ сжатия, как ZIP. Кроме того, большинство программ инсталляции также обладают возможностями сжатия инсталлируемых файлов. Здесь следует отметить, что использование программы разархивирования выполняемых модулей и DLL-модулей увеличивает время запуска программы, но позволяет создать более компактную программу.

Заключительные замечания

Для поиска более подробной и точной информации следует использовать списки рассылки новостей Borland и Web-узлы, посвященные программированию.

- **Доступ к диску**. Последовательные операции чтения и записи выполняются быстрее всего, потому что при этом сокращаются до минимума перемещения головки. Считывайте данные большими порциями, а не по одному символу.
- **Время запуска**. Указывайте уникальный и неперекрывающийся адрес загрузки для всех DLL-модулей. Сокращайте до минимума использование элементов управления ActiveX, видимых во время запуска. Для отвлечения внимания пользователя рекомендуется заставка с ежедневным выводом нового совета для них.
- **Задержка в сети**. Посылайте несколько пакетов одновременно и получайте в ответ данные одним блоком.
- **Доступ к базе данных с архитектурой клиент/сервер**. Используйте кэширование обновлений. Выполняйте потенциально длинные запросы в потоке (например, обновление). Поток вернет список ошибок, которые пользователь сможет исправить. Используйте утилиту SQL Monitor для просмотра текущего состояния программы. Для полей инкрементного поиска используйте задержку перед извлечением совпадающих наборов данных. Для таблицы или поля со списком загружайте сразу не более двух экранов элементов. Задавайте соответствующий размер полосы прокрутки. Используйте диапазоны вместо фильтров для ограничения возвращаемого набора записей.

Помните, что эти рекомендации носят общий характер, поэтому возможны случаи, когда требуются совершенно противоположные действия.

Резюме

В этой главе рассмотрены принципы работы компилятора, способы ускорения времени компилирования, а также новые возможности управления компиляцией в C++Builder 5. Здесь также подробно обсуждаются методы оптимизации с особым вниманием к оптимизации по скорости.

Компиляция и оптимизация представляют собой очень важные и сложные темы, изучение которых, несомненно, принесет большую пользу каждому программисту. Автор настоятельно рекомендует читателям продолжить изучение этих тем с помощью книг и справочных пособий, которые перечислялись здесь или могут быть найдены в других источниках.

Отладка приложения

Глава

7

Джарод Холингвэрт

Эта глава посвящена вопросам отладки приложений. Термин “приложения” используется здесь в очень широком смысле, поэтому многие излагаемые здесь принципы применимы для всех типов проектов.

Отладка — очень сложный и важный аспект программирования, который часто поверхностно рассматривается разработчиками. Эта глава предназначена для всех категорий читателей, независимо от имеющегося у них опыта.

Здесь не будут рассматриваться вопросы отладки в таких областях, как ISAPI, DirectX или MIDAS, хотя кратко будут рассмотрены DLL-модули и удаленная отладка, которые могут быть расширены до отладки серверов COM, DCOM, CORBA и ActiveX. Также будет рассмотрен инструмент ***CodeGuard***, несколько новых элементов C++Builder 5, а также приведены замечания по поводу тестирования.

Обзор принципов отладки

Отладкой называется процесс обнаружения и исправления ошибок в программном обеспечении. Адмирал Грэйс Хоппер (Grace Hopper) придумала термин “bug” (жучок) 9 сентября 1945 года, когда было обнаружено, что причиной сбоя компьютера-мэйнфрэйма Harvard Mark II был обыкновенный мотылек. Этот термин прижился в среде программистов для обозначения сбоя аппаратного обеспечения или ошибки программного обеспечения.

Почему возникают ошибки? Дело в том, что создание безошибочного кода представляет собой очень сложную задачу. Достаточно напомнить официальное заявление Microsoft об обнаружении тысяч ошибок в первой коммерческой версии операционной системы Windows 2000. Действительно, в больших и сложных системах количество ошибок может быть огромным.

Ошибки можно разбить на следующие основные категории.

- **Синтаксические ошибки**. Компилятор способен найти ошибки такого типа. Присутствие таких синтаксических ошибок означает, что в коде также могут быть другие, менее заметные ошибки.
- **Ошибки сборки**. При определенных условиях компиляция не может привести к созданию корректного выполняемого файла. Это может стать результатом применения команды выборочной сборки Make вместо полной сборки Build после внесения в код изменений из-за использования системы контроля версий, неправильного пути к каталогам с библиотеками и включаемыми файлами или из-за продолжающегося редактирования кода во время фоновой компиляции.
- **Семантические ошибки**. Они могут быть вызваны неинициализированными переменными, использованием оператора `&` вместо оператора `&&`, а также неправильным написанием имени переменной. Компилятор не может уловить ошибки такого рода.
- **Алгоритмические ошибки**. Ошибки, вызванные логическими ошибками, найти труднее всего. Простым примером такой ошибки является сортировка списка по размеру, когда первым элементом оказывается наибольший, а не наименьший.
- **Парные ошибки**. Некоторые задачи, обычно связанные с расходованием ресурсов, требуют, чтобы в ответ на первичное действие было выполнено некоторое вторичное действие. Примерами парной ошибки являются: выделение памяти без ее освобождения, открытие файла без его закрытия, проталкивание элемента в стек без выталкивания. Более тонкий пример: увеличение значения счетчика при каждом использовании объекта, но без уменьшения значения счетчика по окончании его использования.
- **Ошибки интерфейса**. Они вызваны несоответствием данных, посланных из одного приложения в другое, или некорректным определением интерфейса. Они могут также стать результатом применения неправильного протокола или версии интерфейса.

- **Побочные эффекты**. Они могут возникнуть, когда одна часть кода случайно приводит к некорректной работе другой части кода. В качестве примера следует назвать перезапись памяти, а также инициализацию переменной неверным значением.
- **Ошибки базовой функциональности**. Эти ошибки возникают при непредусмотренном способе работы компонента. Например, если часть кода этого компонента не используется или полученные результаты основаны на данных из неверного источника.

Большинство ошибок можно отнести к одной из этих категорий. Но могут быть и более специфические ошибки: например, ошибки интерфейса, которые усложняют работу с приложением, а также ошибки производительности, замедляющие работу приложения.

Первый принцип создания безошибочного приложения заключается в том, что при поиске ошибок нельзя целиком полагаться на компилятор. Нужно использовать и учитывать предупреждения компилятора, но только во вторую очередь. В первую очередь следует избавиться от ошибок, найденных компилятором. При выборе способа сборки (Make или Build) проекта следует убедиться в том, что код правильно откомпилирован и корректно работает.

Набор инструментов ***Code Insight*** среды C++Builder содержит несколько средств повышения производительности труда и сокращения вероятности появления ошибок. Среди них следует упомянуть средство автоматического дописывания (или выбора из списка) конструкций кода, средства отображения параметров и использования шаблонов, а также инструмент ***ToolTip Symbol Insight*** указанием места объявления объекта, над которым располагается мышь. Более подробную информацию об этих замечательных инструментах можно найти в справке среды C++Builder.

Рекомендации в отношении проекта в целом

Отладку следует рассматривать как последний этап процесса поиска ошибок в готовом программном продукте. Следует постараться сократить вероятность появления ошибок до минимума с самого начала и применять для этого имеющиеся средства автоматического обнаружения ошибок.

В разделе с вводными сведениями об оптимизации в главе 6, приведены следующие характеристики проекта.

- Скорость.
- Размер.
- Простота сопровождения.
- Тестируемость.
- Возможность повторного использования.
- Надежность.
- Масштабируемость.
- Переносимость.
- Практичность.
- Безопасность.

Если простота сопровождения, тестируемость, возможность повторного использования, надежность, практичность или безопасность имеют большое значение для данного проекта, то предотвращение, обнаружение и устранение ошибок имеют еще большее значение в процессе создания проекта.

Поговорка “к катящемуся камню мох не липнет” не справедлива, если в качестве камня представить эволюционирующее программное обеспечение, а в качестве мха — ошибки! Чем

дольше и сложнее процесс программирования, тем вероятнее появление ошибок. Не тратьте время на создание компонентов, которые вряд ли будут полезны, а создаются только для забавы. Во-первых, это время лучше потратить на усиление надежности программного продукта. Во-вторых, эти необязательные компоненты могут стать причиной необязательных ошибок.

Потратьте время на поиск стабильных компонентов и библиотек сторонних разработчиков, которые могут пригодиться в вашем приложении. Если не создаются самостоятельно, то вероятность появления ошибок в них будет существенно снижена.

При создании в приложении нового компонента сообщите об этом другим разработчикам проекта. Это позволит им избежать потенциальных конфликтов при создании интерфейса и воспользоваться возможностями повторного применения вновь созданных функций. Это сократит вероятность создания дубликатов одного и того же компонента, т.е. время создания приложения и количество потенциальных ошибок.

Рекомендации, касающиеся программирования

Использование методов профессионального программирования позволяет избежать появления ошибок и сократить время, необходимое для их обнаружения и исправления. В главе 3, где речь шла о методах программирования с помощью C++Builder, рассматривается несколько стилей кодирования и профессиональных методов программирования. Применяйте для этого оптимальные приемы объектно-ориентированного программирования, скрытия данных и принципы функциональности. Каждая функция или метод класса C++ должны иметь одну хорошо определенную задачу.

Очень важным фактором создания качественного программного обеспечения является читабельность и документирование кода. Независимо от того сколько человек участвует в создании приложения (один или несколько), код всегда следует создавать понятным даже для непосвященного. При этом особо тщательно следует комментировать самые трудные для понимания участки кода, например оптимизированный или тестовый код.

Во время создания приложения всегда следует помнить об используемых допущениях. Впоследствии, при частом пересечении границ между кодом и данными, их применение может вызвать проблемы. В этом случае необходимо убедиться в правильности сделанного допущения, особенно если оно основано на введенных пользователем данных или данных, полученных от внешних систем.

Если какие-либо функции или выходные данные очень критичны или влияют на безопасность, следует позаботиться об их корректности. В таком случае не помешает рассмотреть возможность использования совершенно другого алгоритма для проверки результатов работы первого.

Во время создания приложения следует мысленно отслеживать каждую строку кода и затем использовать отладчик для каждой небольшой части готового кода. Это — лучшая возможность понять принцип работы созданного кода.

Следует очень внимательно исправлять ошибки, чтобы не допустить при этом появления новых. Быстро исправив одну ошибку, вряд ли стоит надеяться на то, что код сразу заработает нужным образом и без какого-либо побочного отрицательного влияния на другие части кода. Придерживайтесь правила: “Если код удовлетворительно работает, его не нужно исправлять”! Образно говоря, не стоит чинить четвертое колесо разбирая на части три остальных исправных колеса.

Превентивные меры поиска и исправления ошибок в языке C++ включают проверку возвращаемых функциями значений, введение исключительных ситуаций, использование обработчиков исключительных ситуаций для перехвата предполагаемых и непредполагаемых исключительных ситуаций, а также проверку типов. Некоторые сведения об обработке исключительных ситуаций приводятся в главах 3 и 4. Кроме того, не забудьте также протестировать код обработчика исключительных ситуаций.

Цель отладки

Приступая к отладке, следует иметь в виду несколько основных принципов и концепций.

При обнаружении ошибки ее следует немедленно задокументировать и присвоить приоритет. Для этого можно использовать упомянутый в главе 2 новый компонент C++Builder 5, а именно список неотложных задач ***To-Do List***. При отсутствии специального отладочного программного обеспечения для этого также можно применить простой упорядоченный текстовый файл. После документирования ошибки следует немедленно приступить к ее исправлению. Не оставляйте решение этой задачи на потом, до окончания работы над проектом. Это одна из основных причин появления неконтролируемых списков ошибок и неправильной оценки времени создания приложения. Поддержание пустого списка ошибок может существенно облегчить жизнь разработчиков и менеджера проекта.

Список ошибок не всегда является злом, особенно когда большинство ошибок переходит в список "исправленных" ошибок. Демонстрация пользователям (как внутренним, так и внешним) списка замеченных ошибок и списка "исправленных" ошибок лично мне кажется убедительным аргументом в пользу более высокого качества продукта. Это позволяет пользователям убедиться в том, что поиск ошибок был выполнен и они действительно исправлены.

Перед исправлением ошибки всегда следует создать резервную копию исходного кода. Для этого можно использовать систему контроля версий или просто сохранить файл с исходным кодом под другим именем или в другой папке.

Прекрасным пособием для всех программистов является книга *Code Complete*, Стива Макконнелла (Steve McConnell), опубликованная издательством Microsoft Press (ISBN 1-55615-484-4, 1993). В ней очень подробно рассмотрены все аспекты создания программного обеспечения. Она является моей любимой книгой по программированию, и я считаю ее просто "библией программиста".

Наконец, никогда не допускайте повторения сделанных ранее ошибок. Изучите опыт своих и чужих ошибок и не совершайте их в будущем.

Основные методы отладки

В этом разделе мы рассмотрим основные методы обнаружения ошибок, а в следующих разделах будут рассмотрены методы их поиска с помощью интерактивного отладчика, поставляемого в составе C++Builder, а также проанализированы более сложные вопросы отладки.

Прежде чем приступить к довольно длительному процессу отладки приложения с использованием компонентов или библиотек стороннего разработчика, в которых подозревается наличие ошибки, следует проконсультироваться с ним на предмет наличия обновления или заплаты для исправления этой ошибки.

Ниже перечислены основные методы обнаружения ошибок.

- Ручная или программная проверка выходных данных приложения в ходе тестирования.
- Ручная или программная проверка плана выполнения и внутренних данных.
- Программный перехват неверных условий с использованием утверждений и других методов проверки условий.
- Программный перехват исключительных ситуаций с помощью реализации обработчика исключительных ситуаций.

Ручная проверка основана на пошаговом выполнении кода или выводе выходных данных в соответствующих местах, часто с использованием встроенного отладчика C++Builder. На основе знания входных данных и правильного порядка действий, которые должно предпринять приложение, можно определить правильность его функционирования.

Программный перехват неверных условий и исключительных ситуаций может точно указать причину возникновения ошибки. Проблема заключается в том, что место обнаружения ошибки может находиться очень далеко от места ее возникновения. Перечисленные выше методы обнаружения ошибок следует использовать как исходную точку в процессе поиска ошибок.

Ниже кратко описывается несколько методов, которые затем более подробно рассматриваются ниже в этой главе. Для отслеживания источника ошибки обычно используются три основных процесса, которые перечислены ниже.

- Мысленно проследите путь от места обнаружения ошибки обратно вдоль пути выполнения приложения, используя знания о состоянии переменных и объектов. Проверьте стек вызовов с целью получения ключей для поиска пути выполнения. Проследите за данными, которые могли вызвать появление ошибки. Используйте ручную или программную проверку или стратегически введенные утверждения для последующего запуска программы с целью сужения области поиска ошибки.
- Начиная с точки в пути выполнения приложения, до которой ошибка не может возникнуть, мысленно или с использованием интерактивного отладчика проследите за кодом вплоть до той точки, где была обнаружена ошибка. Используйте для этого те же методы, что и при обратном отслеживании кода.
- Третий метод под условным названием “разделяй и властвуй” основан на разновидности бинарного поиска или заключении в скобки кода для поиска ошибки. Для этого исследуемая область кода обычно разбивается на две части, как при бинарном поиске, с повторным изменением границ поиска и сужением области поиска ошибки. При этом следует использовать те же методы, что и при обратном отслеживании кода.

Как бы вы ни пытались искать ошибку, это должен быть системный подход на основе корректных и воспроизводимых выходных данных или отладочной информации, а не на основе хаотических или спорадических результатов. Если ошибка не может быть воспроизведена достаточно надежно, очень возможно, что ее появление вызвано неинициализированными данными или перезаписью памяти.

Если в течение относительно короткого промежутка времени ошибку найти не удалось, сделайте короткий перерыв и попытайтесь применить совершенно другой подход или проконсультируйтесь с более опытным разработчиком. Иногда описание возникшей проблемы другому программисту помогает понять причину возникновения ошибки. Особенно внимательно отнеситесь к недавно созданной части кода.

После обнаружения ошибки потратьте некоторое время на поиск истинной причины ее возникновения и поиска правильного решения для ее устранения. Не соблазняйтесь быстрым способом исправления ошибки, который, вероятно, не позволит утратить истинные причины возникновения ошибки или будет способствовать появлению других ошибок. Сразу после исправления ошибки рекомендуется тщательно протестировать исправленный код.

Просмотр результатов отладки

Основным методом отладки является просмотр информации из одного или нескольких мест приложения для обнаружения выполняемого кода и состояния отдельных объектов и переменных. Только сам разработчик может решить, какие именно сведения ему нужны, но в любом случае следует отобразить: входные данные для проверки их корректности; параметры, передаваемые ключевым функциям; а также информацию о плане выполнения для отображения решений, которые принимает приложение на ключевых этапах. В этом разделе рассматривается несколько методов получения этих сведений.

Очень полезным новым компонентом C++Builder 5 является директива _DEBUG, которая определяется в списке Conditional вкладки Directories/Conditionals в окне параметров проекта Project Options при нажатой кнопке Full Debug во вкладке Compiler. Она задается по умолчанию. Если нажата кнопка Release, директива _DEBUG удаляется из списка Conditional. Имя этого определения можно изменить, добавив соответствующую строку с именем DebugDefine в ключе реестра Windows HKEY_CURRENT_USERS\Software\Borland\C++Builder\5.0\Debugging.

Определение _DEBUG позволяет указать определенные разделы кода, которые следует компилировать в приложении только при использовании параметров полной компиляции Full Debug с помощью директив препроцессора #ifdef и #endif. Пользователи C++Builder версии 4 и более ранних версий должны вручную добавить и удалить директиву _DEBUG или аналогичное определение в списке Conditional defines или указать его в коде с помощью директивы #define _DEBUG. Далее в этом разделе будет использовано определение _DEBUG.

Существует несколько способов вывода отладочной информации об исследуемом коде. Простейший способ заключается в использовании оператора printf() или оператора cout << при отладке консольного приложения. Для использования оператора printf() нужно использовать заголовочный файл stdio.h, а при использовании оператора cout << — заголовочный файл iostream.h. В листинге 7.1 показаны способы использования этих методов для создания функции вывода отладочной информации общего типа, MyDebugOutput(), а также пример того, как она может использоваться в приложении. Текстовая строка Debug: отображается в начале каждого отладочного сообщения для визуального отделения отладочных сообщений от выходных данных приложения.

Листинг 7.1. Вывод отладочной информации в консольных приложениях

```
#include <stdio.h>
#include <iostream.h>

void MyDebugOutput(AnsiString OutputMessage)
{
   // Метод 1: Вывод отладочной информации с помощью
   //          функции printf().
   printf("Debug: %s\n", OutputMessage.c_str());

   // Метод 2: Вывод отладочной информации с помощью
   // функции cout <<.
   cout << "Debug: " << OutputMessage.c_str() << endl;
}

void NormalFunc(int MaxLines)
{
   // Код приложения.
   MyDebugOutput("Before loop, MaxLines=" + IntToStr(MaxLines));
   for (int i = 0 ; i < MaxLines ; i++) {
      MyDebugOutput("In loop, i=" + IntToStr(i));

      // Код приложения.
   }
}
```

В приложениях с GUI-интерфейсом можно использовать надписи, текстовые поля или окна формы либо диалоговые окна с отображением сообщений. Для этого можно также применить Win32 API-функцию `MessageBeep(0xFFFFFFFF)`, чтобы подать звуковой, а не визуальный сигнал. В листинге 7.2 показана функция `MyDebugOutput()` из листинга 7.1, которая изменена для применения в приложении с GUI-интерфейсом. Функция `MyDebugBeep()` в листинге 7.2 может быть вызвана в любом месте приложения, где вместо визуального сигнала удобно использовать звуковой.

Листинг 7.2. Вывод отладочной информации в приложениях с GUI-интерфейсом

```
void MyDebugOutput(AnsiString OutputMessage)
{
   // Метод 1: Вывод отладочного сообщения в надписи,
   // текстовом поле или окне основной формы. Любого из них
   // может оказаться вполне достаточно.
   MainForm->ErrorLabel->Caption = OutputMessage;
   MainForm->ErrorEdit->Text = OutputMessage;
   MainForm->ErrorMemo->Text = OutputMessage;

   // Метод 2: Вывод отладочного сообщения с помощью
   // функции ShowMessage().
   ShowMessage(OutputMessage);

   // Метод 3: Вывод отладочного сообщения с помощью
   // функции MessageDlg().
   MessageDlg(OutputMessage,
              mtInformation, TMsgDlgButtons() << mbOK, 0);
}

void MyDebugBeep()
{
   // Использование звукового сигнала,
   // заданного по умолчанию в операционной системе Windows.
   MessageBeep(0xFFFFFFFF);
}
```

В многопотоковом приложении часто бывает полезно выводить информацию об идентификаторе каждого потока с помощью Win32 API-функции `GetCurrentThreadId()`.

Описанные выше методы отладки могут пригодиться только в самых простых случаях. Одна из проблем использования диалоговых окон заключается в том, что их трудно использовать при отладке метода `Paint()` графических компонентов. Отображение такого диалогового окна на экране скрывает часть элемента управления, что может снова инициировать сообщение `WM_PAINT` для повторного вызова метода `Paint()`.

Альтернативным вариантом отладки в IDE-среде является использование Win32 API-функции `OutputDebugString()`. Эта функция возвращает строку, которая может быть использована отладчиком. В нашем случае это сообщение передается интегрированному отладчику C++Builder и пересылается в журнал учета событий. А если приложение запущено вне IDE-среды (вне области действия отладчика), то никакие выходные данные не фиксируются.

Журнал учета событий можно просмотреть, выбрав команду View⇨Debug Windows⇨Event Log или нажав комбинацию клавиш <Ctrl+Alt+V>. Закрепите окно этого журнала Event Log под окном редактора кода Code Editor или просмотра классов Class Explorer. Функция `OutputDebugString()` принимает один параметр, т.е. определенное пользователем сообщение для вывода его в переменной типа `const char *`, что позволяет полностью контролировать содержимое этого сообщения. Все результаты, помещаемые в журнал учета событий, имеют префикс `ODS:` и суффикс `Process <имя_проекта>.exe`.

Прекрасные возможности отладки предоставляют макросы `TRACE` и `WARN`. Они используют функцию `OutputDebugString()` для генерации сообщений, но при этом автоматически предоставляют информацию об имени файла и номере строки, а также поддерживают работу с операторами `iostream` для создания определенного пользователем сообщения. Для этого необходимо использовать директивы `#define __TRACE` и `#define __WARN`, а после них — директиву `#include <checks.h>`. После этого макросы можно применять в коде приложения. Макрос `TRACE` имеет один аргумент, а именно строковый поток, который содержит заданное пользователем сообщение. Макрос `WARN` имеет два аргумента: тестовое выражение и заданное пользователем сообщение. Это сообщение будет выведено только в том случае, если тестовое выражение будет оценено как истинное.

В листинге 7.3 приведен пример использования директивы с определением режима отладки `_DEBUG` и макросами `TRACE` и `WARN` для условной генерации отладочной информации. Хотя их можно оставить нетронутыми даже в готовом для реализации коммерческом приложении, так как сообщения функции `OutputDebugString()` не будут использованы вне IDE-среды, такое приложение будет иметь очень низкую производительность. Макрос `TRACEF` в листинге 7.3 аналогичен макросу `TRACE`, но в его отладочное сообщение добавлено имя функции. Этот макрос также демонстрирует способ применения функции `OutputDebugString()`.

Листинг 7.3. Использование макросов `TRACE`, `WARN` и функции `OutputDebugString` при компиляции в режиме отладки

```
#ifdef _DEBUG
#define __TRACE
#define __WARN
#endif
#include <checks.h>
#pragma option -w-ccc

#define TRACEF(s)TRACE(__FUNC__ ": " << s)

void MyFunc(AnsiString Title, int *MyArray, int Max)
{
   int i, Sum = 0;

   TRACE("Simple trace");
   TRACEF("Includes function name");

   for (i = 0 ; i < 10 ; i++) {
      Sum += MyArray[i];
   }

   TRACE("The sum is " << Sum);
```

```
   WARN(Sum > Max,
        "The Sum is too big! The maximum is " << Max);
   OutputDebugString(Title.c_str());
}
```

Обратите внимание, что в листинге 7.3 параметр компилятора `-w-ccc` использован в директиве препроцессора `#pragma option`. Дело в том, что макросы `TRACE` и `WARN` фактически реализованы в стиле цикла `do {} while (0)`, который приводит к появлению предупреждения `Condition is always true` (Условие всегда выполняется) при каждом их использовании. А эта директива всегда отменяет вывод такого предупреждения. Другими словами, этот параметр компилятора можно указать в файле проекта или отменить вывод предупреждений, задав для параметра Warnings значение Selected во вкладке Compiler в окне параметров проекта Project Options, нажав кнопку Warnings и сняв флажок Condition is always true... с префиксом `W8008` в C++Builder 5.

Часто при выводе отладочной информации требуется отобразить содержимое переменных вместе с их именами, например `переменная=значение`, или отобразить выражение вместе с результатом его вычисления, например `A+B=13`. Это действие может очень сложно формулироваться и редактироваться в отладочной строке. Директива макроса `#macro` может избавить программиста от этих трудностей, преобразуя аргументы макроса в строку. Макрос `DEBUGEXP` в листинге 7.4 демонстрирует применение этого способа.

Листинг 7.4. Макрос отладки простого выражения

```
#ifdef _DEBUG
#define __TRACE
#define __WARN
#endif
#include <checks.h>
#pragma option -w-ccc

#define DEBUGEXP(x) TRACE(__FUNC__ ": " #x "=" << (x))

void MyFunc()
{
   int A =6,
       B =7;

   DEBUGEXP(A);
   DEBUGEXP(A+B);
}
```

Нужно всегда учитывать вероятность возникновения побочных эффектов при передаче аргументов описанным выше макросам, например вызовов функций, инициализации переменных или выполнении операций инкремента `++` или декремента `--`. Например, применение директивы `DEBUGEXP(++a)` приведет к инкременту `a`, но если `_DEBUG` не определено, то это выражение не будет вычислено, а переменная `a` не будет изменена.

Вообще, рекомендуется их в виде встраиваемых функций. Однако в предыдущем пример, макрос придется использовать для применения макроса `__FUNC__` и директивы `#` (преобразование в строку).

Последний пример использования макросов для трассировки выполнения кода представлен в листинге 7.5.

Листинг 7.5. Трассировка выполнения кода

```
#ifdef _DEBUG
#define __TRACE
#define __WARN
#endif
#include <checks.h>
#pragma option -w-ccc

#define DEBUGTRCN(s) TRACE("CMD: " #s); s
#define DEBUGTRCC(s) s; TRACE("CMD: " #s)

void MyFunc()
{
   int i,
       Sum;

   DEBUGTRCN(Sum = 0);
   DEBUGTRCN(for (int i = 0 ; i < 10 ; i++) {);
   DEBUGTRCN(Sum += i);
   DEBUGTRCC(});

   DEBUGTRCN(if (Sum > 20) {);
   DEBUGTRCN(ShowMessage("Sum is large"));
   DEBUGTRCC(}else {);
   DEBUGTRCN(ShowMessage("Sum is small"));
   DEBUGTRCC(});
}
```

Макросы `DEBUGTRCN` и `DEBUGTRCC` в листинге 7.5 приводят к созданию двух выражений: одно из них отображает, а другое — фактически выполняет заданное выражение. Макрос `DEBUGTRCN` используется для большинства выражений, но для выражений с закрывающей скобкой (}), как показано в этом примере, необходимо использовать макрос `DEBUGTRCC`. Так как эти макросы развертываются в два выражения, они не могут использоваться в сингулярных выражениях (например, после `if` или `for`) без использования скобок для охвата этого выражения. Однако использование этих макросов не вызывает никаких побочных эффектов.

Применение утверждений

Утверждение представляет собой способ проверки сделанных в коде предположений. Оно утверждает, что выражение является истинным. Если выражение ложно, то утверждение приведет к отображению сообщения об ошибке и прекращению выполнения программы либо инициированию исключительной ситуации. Макрос `WARN` в предыдущем разделе является примером базового утверждения, за исключением использования обратной логики и отображения только некоторой отладочной информации без прекращения выполнения программы. Некоторые программисты предпочитают реализовывать обработку исключительных ситуаций, а не использовать утверждения. Обработка исключительных ситуаций будет кратко рассмотрена в следующем разделе.

Создание соответствующих утверждений в коде может представлять собой достаточно длительный процесс. Поэтому их лучше вводить при создании кода, когда принятые допущения еще свежи в памяти. Утверждения обычно создаются в начале вызова функции или метода класса, но их в равной степени можно располагать в любом месте кода.

Утверждения обычно включатся только в отладочную или тестовую версию приложения, но очень редко — в готовую коммерческую версию. Созданные ранее утверждения следует оставить нетронутыми в готовой версии исходного кода и компилировать их с использованием директив препроцессора `#ifdef`. Это позволяет применять их для последующего тестирования, например, после добавления в приложение новых компонентов. Большинство утверждений имеет встроенные конструкции для отказа от компиляции.

Для использования встроенных утверждений в среде C++Builder необходимо с помощью директивы `#include <assert.h>` включить в код заголовочный файл `assert.h`. Затем в нужное место кода с помощью команды `assert(выражение)` вставить утверждение. Напомним, что, в отличие от макроса `WARN`, это выражение должно быть истинным при нормальном функционировании (т.е. в случае справедливости сделанного допущения) и ложным при ошибочном допущении.

В листинге 7.6 приводится пример использования утверждений для проверки диапазона параметров функции и общих условий.

Листинг 7.6. Пример использования утверждений

```
// Выше располагаются стандартные заголовочные файлы.
#include <assert.h>

// Здесь располагаются функции.
void MyFunc(int Width)
{
   int Length;

   // Ширина не может быть нулевой.
   assert(Width != 0);

   Length =Area/Width;
   // Длина Length должна быть меньше MaxLength
   assert(Length < MaxLength);

   // Если все в порядке, то продолжить работу.
}

void DisplayNode(TNode *Node)
{
   // Убедиться в существовании узла.
   assert(Node != (TNode *)NULL);

   // Если все в порядке, то продолжить работу.
}
```

Если в консольном приложении утверждение будет неверным, то в стандартный поток `stderr` будет передано сообщение `Assertion failed:`, затем выражение утверждения, имя файла модуля, а также номер строки этого выражения утверждения. В приложении с GUI-интерфейсом на экране будет отображено диалоговое окно, которое содержит выражение ут-

верждения, имя файла модуля, а также номер строки с выражением утверждения. На рис. 7.1 показан пример сообщения об ошибке, которое появляется в приложении с GUI-интерфейсом, если в предыдущем примере аргумент функции `MyFunc()` получит значение `0`.

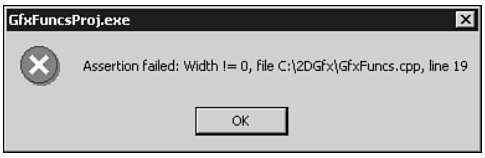


Рис. 7.1. Это диалоговое окно появится, если утверждение окажется ложным

Утверждения могут быть удалены из кода во время компиляции с помощью директивы `NDEBUG`. Это можно сделать, использовав список Conditional defines во вкладке Directories/Conditionals окна параметров проекта Project Options или директиву `#define NDEBUG` перед строкой `#include <assert.h>`, если определение `_DEBUG` не задано для связи всего отладочного кода в одном определении. Как уже говорилось выше, выражения с утверждениями можно оставить нетронутыми в готовом коде приложения.

Так как компиляцию утверждений можно впоследствии отключить, то не рекомендуется применять в них операторы присваивания, инкремента или декремента переменных или вызовы функций, которые могут стать причиной побочных эффектов. Нарушение этого правила может привести к возникновению противоречий между отладочной и готовой версиями приложения. Напомним о необходимости документирования допущений, которые не совсем очевидны при чтении кода.

Создание глобального обработчика исключительных ситуаций

Альтернативным или дополнительным вариантом для использования утверждений является применение профессионального обработчика исключительных ситуаций. В частности, создание глобального обработчика исключительных ситуаций для перехвата всех необработанных исключительных ситуаций.

Глобальный обработчик исключительных ситуаций может сообщать о типе исключительной ситуации, месте ее возникновения, а также о текущем состоянии приложения. Более сложные типы обработчиков исключительных ситуаций, которые способны регистрировать состояние приложения в файле, подробно рассматриваются в главе 4.

Простым альтернативным вариантом метода, описанного в главе 4, является компиляция с выводом информации о месте возникновения исключительной ситуации с помощью установки флажков параметров Enable Exceptions и Location Information во вкладке C++ в диалоговом окне параметров проекта Project Options или с помощью параметра командной строки компилятора `-xp`. С помощью директивы `#include <except.h>` можно получить доступ к следующим трем функциям.

- Функция `__ThrowExceptionName()` возвращает указатель `char *` на имя исключительной ситуации.
- Функция `__ThrowFileName()`возвращает указатель `char *` на имя файла, в котором возникла исключительная ситуация.
- Функция `__ThrowLineNumber()`возвращает значение типа `unsigned int`, содержащее номер строки из исходного файла, в которой возникла исключительная ситуация.

Пример простой исключительной ситуации приводится в показанном ниже коде.

```
#include <except.h>

   // В основной функции приложения.
   try {
      // Выполнение основной работы (вызов функций, и т.д.)
   }
   catch (Exception &e ) {
      ShowMessage(e.Message + "\nType: " +
                  __ThrowExceptionName() + "\nFile: " +
                  __ThrowFileName() + "\nLine: " +
                  AnsiString(__ThrowLineNumber()));
   }
```

Конечно, всегда можно создать более эффективный обработчик, например, способный записывать отладочную информацию в файл, который для устранения неисправности можно переслать по электронной почте соответствующему специалисту-консультанту службы технической поддержки.

Другие вопросы базовой отладки

В языке C++ есть несколько особенностей, которые усложняют поиск перечисленных ниже ошибок при визуальном контроле кода.

- Помните, что числа, которые начинаются с нуля `0`, записаны в восьмеричной системе счисления. Например, число `010` является числом `10` в восьмеричной системе счисления (т.е. числом `8` в десятичной системе счисления).
- Не путайте число `0` (нуль) с буквой `O` при задании входных данных и сравнении строковых и символьных выражений.
- Внимательно отнеситесь к выражениям, в которых ошибочно могут использоваться битовые операторы `&`, `|` или `~` вместо логических операторов `&&`, `||` или `!`, если только это не сделано преднамеренно в целях оптимизации.
- Заключайте макросы в скобки. Макрос `MAX(a,b)` определенный в виде `a>=b?a:b` не будет работать в выражении `Z = Y + MAX(A,B)`. Для этого нужно применить скобки `(a>=b?a:b)`.
- Помните о побочных эффектах при использовании макросов. Если макрос `MAX(a,b)` определен как `(a>=b?a:b)`, то выражение `Z = MAX(A++,B)` приведет к двойному инкременту переменной `A`, но значение `Z` будет вычислено так, как если бы состоялась одна операция инкремента переменной `A`.
- Помните, что макрос — не функция, поэтому по возможности используйте встраиваемые функции вместо макросов.
- Избегайте использования таких причудливых функций, как, например, `strncpy(s1,s2,length)`, которая может (или не сможет) добавить концевой символ `NULL` в строку `s1` при копировании строки `s2`, в зависимости от величины аргумента `length` и длины строки `s2`.

В следующих разделах рассматриваются более сложные методы отладки.

Интерактивный отладчик C++Builder

Интерактивный отладчик C++Builder содержит много сложных элементов с такими возможностями, как оценка выражений, установка данных, инспекция объектов, сложные точки останова, просмотр дизассемблированного машинного кода, просмотр регистров FPU и MMX, отладка межпроцессового взаимодействия, удаленная отладка, присоединение к выполняемому процессу, слежение за результатами выполнения выражения, просмотр стека вызовов, пошаговое выполнение кода и многое другое. При создании приложения программисту придется затратить немало времени на работу с ними (по крайней мере это настоятельно рекомендуется сделать).

Отладчик предназначен не только для поиска ошибок, но и для изучения принципов работы кода на низком уровне.

Для эффективного использования отладчика прежде всего следует снять все параметры оптимизации компилятора. При их использовании оптимизатор любой ценой попытается ускорить работу приложения или сократить размер кода, удаляя, перемещая или группируя отдельные фрагменты. Это затрудняет пошаговое выполнение кода и установление соответствия между исходным и машинным кодом при просмотре состояния процессора. При указании контрольной точки в некоторой строке и непопадании в нее помните, что это может быть вызвано применением параметров оптимизации.

Отладка может существенно затрудняться при работе с экраном, избыточно перегруженным окнами просмотра отладочной информации. Для создания оптимальной и удобной среды программирования и отдельной от нее среды отладки в C++Builder 5 можно применять многочисленные параметры настройки рабочей среды.

Типичная рабочая среда отладки показана на рис. 7.2. Она включает: (а) окна стека вызовов Call Stack, просмотра переменных Watches и локальных переменных Local Variables, закрепленных вертикально и слева от экрана; (б) окно редактора кода Code Editor в правом верхнем углу, которое занимает большую часть экрана; (в) окно просмотра состояния процессора CPU под окном редактора кода Code Editor в нижнем правом углу экрана. Иногда во время отладки автор использует окно инспектора отладки Debug Inspector, которое располагается над окнами (а) и закрепляет окно событий Event Log под окном редактора кода Code Editor, если окно CPU не используется. Для применения такой среды отладки рекомендуется использовать большой экран и разрешение 1280×1024.

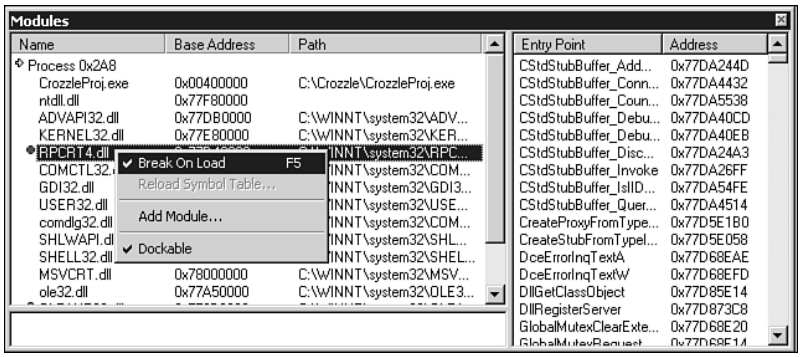


Рис. 7.2. Окна отладки

В остальной части этого раздела предполагается, что читателю знакомы основные принципы работы отладчика, например использование контрольных точек в коде с условиями и счетчиками, перешагивание и вход в отдельные фрагменты кода, а также оценка выражений с помощью контекстной подсказки (при размещении указателя мыши над выражением во время паузы в работе приложения в процессе отладки). Рассмотрим несколько наиболее сложных компонентов отладки.

Сложные контрольные точки

Помимо стандартных контрольных точек, которые просто временно прекращают выполнение приложения при достижении выбранного фрагмента кода или строки кода ассемблера, можно также применить более сложные контрольные точки. В следующем разделе мы рассмотрим несколько новых возможностей контрольных точек в C++Builder 5.

Контрольные точки модулей особенно полезны при отладке DLL-модулей и пакетов. Выполнение приложения можно приостановить при загрузке выбранного модуля, что идеально позволяет достичь точки входа в DLL-модуль или пакет для их отладки. Контрольную точку модуля можно указать двумя способами.

Во-первых, при выполнении приложения в IDE-среде. Для этого сначала откройте окно Modules, выбрав команду меню View⇨Debug Windows⇨Modules. В списке модулей в верхней левой части окна Modules найдите модуль, в котором требуется разместить контрольную точку. Если нужный модуль отсутствует в этом списке, значит, он не загружен приложением. В этом случае в список модулей нужно добавить нужный модуль, выбрав команду Add Module в контекстном меню, затем ввести имя модуля или найти его и щелкнуть на кнопке OK. Наконец, выберите модуль в списке модулей, а затем — команду меню Break On Load из контекстного меню (рис. 7.3). Если нужный модуль уже был загружен приложением, то контрольная точка сработает только при его повторной загрузке, динамической отгрузке либо после перезапуска.

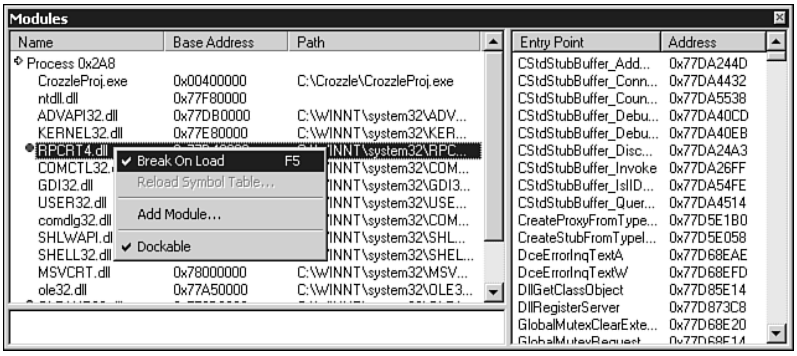


Рис. 7.3. Установка контрольной точки для модуля с помощью окна Modules

Во-вторых, если приложение не выполняется в IDE-среде, выберите команду меню Run⇨Add Breakpoint⇨Module Load Breakpoint, затем введите имя модуля или найдите его и щелкните на кнопке OK. Наконец, запустите приложение.

Контрольные точки на основе адреса и значений данных также предоставляют возможность приостановить выполнение приложения по достижении заданного адреса или при изменении данных по заданному адресу. Они могут быть добавлены только при условии выполнения приложения или его останова.

Контрольная точка по адресу (address breakpoint) работает аналогично обычной контрольной точке, хотя они создаются на основе значения адреса отдельной инструкции в оперативной памяти, а не строки исходного кода. Контрольная точка по адресу срабатывает при выполнении машинного кода. Если контрольная точка задана для инструкции, связанной со строкой кода, то она задается как обычная контрольная точка в этой строке исходного кода. Контрольная точка по адресу обычно используется при отладке внешних модулей на низком уровне с помощью окна просмотра состояния процессора CPU. Окно просмотра состояния процессора CPU более подробно рассматривается ниже в этой главе.

Рассмотрим теперь, как задается контрольная точка по адресу. Для этого мы используем проект BreakpointProj.bpr, который находится в папке Breakpoints компакт-диска, прилагаемого к книге. Загрузите его в среду C++Builder, а затем скомпилируйте и запустите приложение, выбрав команду меню Run⇨Run. После отображения формы остановите приложение, выбрав команду меню Run⇨Program Pause. При этом в окне CPU будет отображено текущее состояние процессора.

Выберите команду меню View⇨Units, затем — модуль BreakpointForm в списке модулей и щелкните на кнопке OK. Теперь код этого модуля будет отображен в окне редактора кода Code Editor. Прокрутите его до функции AddressBreakpointButtonClick(). Щелкните правой кнопкой мыши на строке Label2->Caption = "New Caption" в этой функции и выберите из контекстного меню команду Debug⇨View CPU. Окно CPU снова отобразится на экране, но на этот раз с указанием адреса в оперативной памяти, по которому располагается машинный код данного выражения C++.

В верхней левой части окна CPU обратите внимание на шестнадцатеричное число слева от строки, содержащей машинный код lea eax,[ebp-0x04]. У автора этот адрес равен 004016ED, но у читателя он может быть совершенно другим. Это именно тот адрес, по которому будет задана контрольная точка.

Для вставки контрольной точки по заданному адресу выберите команду меню Run⇨Add Breakpoint⇨Address Breakpoint, а потом в поле Address введите ранее найденный адрес. Шестнадцатеричные числа, с помощью, которых отображаются адреса в окне CPU, должны начинаться с символов 0x. Например, в данном случае следует ввести число 0x004016ED.

Для проверки работоспособности контрольной точки следует продолжить выполнение программы, выбрав команду меню Run⇨Run. Затем выберите в панели задач Windows это приложение и щелкните на кнопке Address Breakpoint. Заданная контрольная точка приостановит выполнение приложения. В окне CPU будет показано текущее состояние пути выполнения с зеленой стрелкой, указывающей на строку кода с контрольной точкой. Продолжить выполнение приложения можно, снова выбрав команду меню Run⇨Run.

Если в окне CPU отобразить строку машинного кода по адресу, для которого необходимо задать контрольную точку, то ее можно задать, щелкнув кнопкой мыши в межстолбцовом промежутке строки машинного кода, как это делается при создании обычной контрольной точки в исходном коде. Это можно было бы сделать в предыдущем примере, но автору хотелось продемонстрировать общий принцип указания фактического адреса.

Контрольные точки по адресу очень полезны для отслеживания ошибок за счет обнаружения в коде отдельной переменной или адреса памяти. В качестве примера рассмотрим проект BreakpointProj.bpr. Запустим и приостановим его выбирая команду меню Run⇨Add Breakpoint⇨Data Breakpoint. В поле Address введите Form1->FClickCount и щелкните на кнопке OK. Этот закрытый член с данными формы играет роль счетчика нажатий кнопки DataBreakpointButton. Указание этой контрольной точки приведет к останову приложения при каждом изменении счетчика.

Как видите, для контрольной точки по адресу может использоваться любое истинное выражением с адресом данных, а не только адресом в оперативной памяти. Альтернативный вариант заключается в получении адреса, выборе команды меню Run⇨Inspect, вводе Form1->FClickCount, получении адреса в верхней части окна Debug Inspector. Теперь этот шестнадцатеричный адрес можно ввести (с префиксом 0x) в поле Address.

Для проверки работоспособности этой контрольной точки продолжите выполнение программы, выбрав команду меню Run⇨Run. Выберите это приложение в окне задач Windows и щелкните на кнопке Data Breakpoint. Заданная контрольная точка приведет к остановке приложения в месте изменения этих данных. Соответствующая строка кода будет показана в окне редактора кода Code Editor либо в окне CPU. Продолжить выполнение приложения можно, снова выбрав команду меню Run⇨Run.

Несколько сложнее выглядит процесс создания контрольной точки по значению данных для такого свойства, как текст надписи `Caption` или текстового поля `Text`. Эти свойства не имеют прямого соответствия с адресами в оперативной памяти, по которым записываются их изменения. Вместо этого для изменения своих значений эти свойства используют функции `Set`. Для останова в момент изменения этих свойств проще всего создать контрольную точку по адресу для функции `Set`, а не выполнять поиск в памяти адреса, по которому фактически хранятся эти данные и создавать контрольную точку по данным. Поясним этот метод на примере свойства `Caption` надписи `ClickCountLabel` формы из предыдущего примера.

После остановки проекта выберите команду Run⇨Inspect. В поле Expression введите `Form1->ClickCountLabel`. Выберите вкладку Properties в окне Debug Inspector и найдите в нем свойство `Caption`. Метод записи значения этого свойства называется `SetText`. Щелкните на вкладке Methods и найдите его. При этом справа будет отображен адрес метода. Выберите команду меню Run⇨Add Breakpoint⇨Address Breakpoint и введите адрес метода `SetText` с префиксом `0x` без точки с запятой, а затем щелкните на кнопке OK. Продолжить выполнение приложения можно, снова выбрав команду меню Run⇨Run. Теперь при любом изменении текста этой надписи контрольная точка приостановит выполнение приложения.

Для слежения за изменением значения стандартной переменной `AnsiString`, которая не имеет метода `Set`, например закрытого члена данных `FSomeString` формы в данном примере, можно использовать контрольную точку по переменной `.Data` класса `AnsiString`, который содержит строковые данные. В рассматриваемом примере такая контрольная точка может быть задана для строки `Form1->FSomeString.Data`.

При создании контрольной точки по значению данных в поле Length окна Add Data Breakpoint для сложных типов данных, например структур или массивов, следует указывать их длину. Контрольная точка приостановит приложение при изменении адреса в рамках этой длины. Контрольные точки по значению данных можно также создавать, выбирая команду меню Break when Changed из контекстного меню в диалоговом окне Watches, отображаемом на экране с помощью команды меню View⇨Debug Windows⇨Watches.

На заметку

Контрольные точки по адресу и по значению данных могут корректно работать только для текущего выполняемого приложения. Для каждого нового запуска приложения их необходимо задавать заново, поэтому адреса инструкций машинного кода и данных меняются при каждом новом запуске приложения.

Новые возможности работы с контрольными точками в C++Builder 5

Контрольные точки можно разбить на группы и сопоставлять их с некоторыми действиями. Благодаря контрольным точкам можно выполнять или прекращать какие-либо действия, выполнять или прекращать обработку исключительных ситуаций, записывать в журнале учета событий сообщения и результаты вычислений.

Благодаря новым возможностям можно задавать достаточно сложные условия приостановки работы программы в специально заданных случаях. Например, целый набор контрольных точек можно активизировать только при выполнении определенного фрагмента кода.

Задавая или отменяя обработку исключительных ситуаций, можно управлять обработкой ошибок в участках кода, в которых могут появляться ошибки. Регистрация сообщений помогает автоматизировать просмотр значений переменных и трассировку их выполнения.

Новые возможности по группированию отладочной информации в контрольных точках и связь контрольных точек с некоторыми действиями также могут использоваться в подсказках ToolTip для контрольных точек в окне редактора кода Code Editor и в окне списка контрольных точек.

Окна просмотра отладочной информации в C++Builder

Отладчик может использоваться для отображения информации, полезной для отладки приложения, например локальных переменных, списка всех контрольных точек, стека вызовов, списка загруженных модулей, состояния потоков, машинного кода, состояния данных и регистров, регистрации событий приложений и т.п.

Новинкой в C++Builder 5 является окно FPU (Floating-Point Unit), которое показывает состояние модуля обработки чисел с плавающей запятой и регистров MMX. Окна просмотра отладочной информации можно открыть с помощью команд меню View⇨Debug Windows или соответствующей комбинации клавиш быстрого доступа. В следующих разделах мы рассмотрим наиболее сложные окна просмотра и способы их использования для отладки приложения.

Окно CPU

В окне CPU можно просматривать приложение на уровне машинного кода. Машинный и ассемблерный код приложения предлагается в нем вместе со сведениями о состоянии регистров и флагов процессора, стека компьютера, а также данных (получаемых в результате) дампа оперативной памяти. Как показано на рис. 7.4, окно CPU содержит пять панелей.

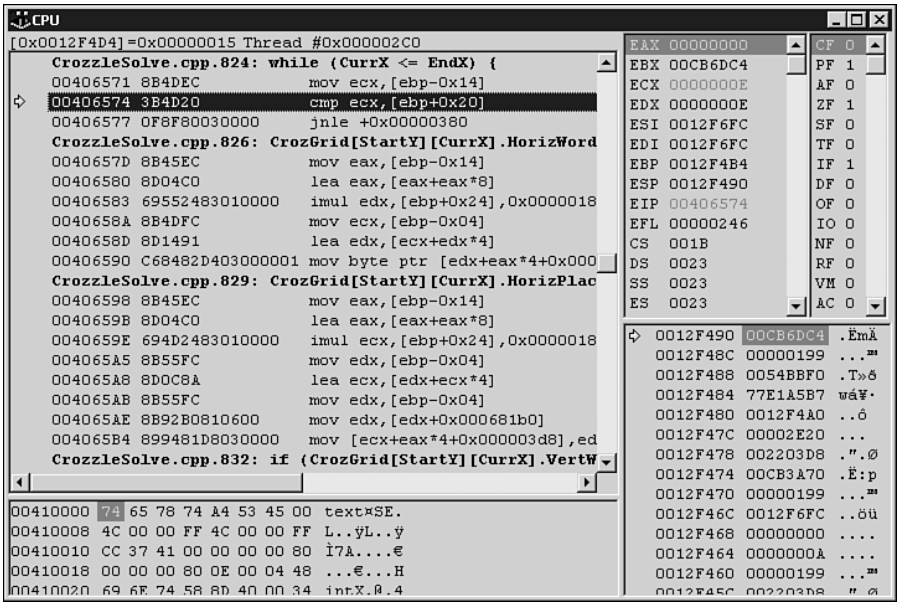


Рис. 7.4. Окно просмотра отладочной информации CPU

Большая панель слева содержит дизассемблированные инструкции машинного кода (или ассемблерного кода) приложения. В левом столбце этой панели располагается адрес инструкции, за ним — столбец данных машинного кода и затем эквивалентный ассемблерный код. Над этой панелью располагается фактический адрес выражения для текущей выделенной строки машинного кода, хранимое по этому адресу значение, а также идентификатор потока. На рис. 7.4 показано, что фактическим адресом инструкции `[ebp+0x20]` является значение `0x0012F4D4`, по которому хранится значение `0x00000015`, а идентификатором потока является `0x000002C0`.

Если перед компиляцией приложения установить флажок параметра Debug Information во вкладке Compiler в окне параметров проекта Project Options то в панели диссамблированного кода будут приведены строки кода на языке C++ над соответствующими инструкциями ассемблерного кода. На рис. 7.4 также показаны строки кода на языке C++.

В панели диссамблированного кода можно следить за пошаговым выполнением инструкций, так же, как при пошаговой трассировке исходного кода в окне редактора кода. Зеленая стрелка при этом показывает текущую выполняемую инструкцию. В этом случае можно использовать контрольные точки и другие инструменты, которые используются для отладки исходного кода в окне редактора кода. С помощью команд контекстного меню можно менять потоки, выполнять поиск данных в памяти, а также изменять текущую точку выполнения.

Панель состояния регистров процессора располагается справа от панели диссамблированного кода. В ней показано текущее значение каждого регистра процессора. При изменении значения регистра он окрашивается в красный цвет. Изменить значение регистра можно с помощью команд контекстного меню.

В верхнем правом углу располагается панель флагов процессора. По сути это расширенное представление регистра `EFL` (32-разрядного флага) в отдельной панели флагов процессора. Изменить значение флага также можно с помощью команд контекстного меню. Описание каждого флага можно найти в интерактивной справке.

Под панелью диссамблированного кода располагается панель дампа памяти. Она применяется для отображения содержимого памяти в адресном пространстве приложения. Слева располагается адрес, а за ним показан дамп памяти по этому адресу в шестнадцатеричном формате и формате ASCII. С помощью команд контекстного меню можно изменить способ отображения этих данных, а также перейти к заданному адресу для поиска нужных данных.

Последняя панель стека компьютера расположена в нижнем правом углу окна CPU. В ней отображается содержимое текущего состояния стека приложения, указанное в регистре `ESP` (указатель стека) процессора. Эта панель аналогична панели дампа памяти с теми же командами контекстного меню.

Окно CPU представляет собой прекрасный инструмент для изучения работы приложения на самом низком уровне. Анализируя приложение на этом уровне, можно получить более полное представление об указателях и массивах, а также скорости выполнения (при оптимизации приложения). С помощью панелей этого окна упрощается процесс отладки приложения, т.к. программист может проанализировать процессы, происходящие на самом низком уровне.

Самым лучшим справочным пособием по инструкциям процессоров x86 и Pentium является руководство разработчика *Intel Architecture Software Developer's Manual*. Этот трехтомник содержит все необходимые сведения о процессорах Pentium. Его можно скопировать из раздела `Manuals` для соответствующего процессора с Web-узла фирмы Intel по адресу `http://developer.intel.com/design/processor/`. Другая рекомендованная литература и некоторые сведения по ассемблеру также приводятся в главе 6.

Программирование на ассемблере в настоящее время стало эзотерическим знанием, так как этот язык очень сложен и используется чрезвычайно редко только для создания небольших, но очень эффективных и критичных по скорости фрагментов кода.

Окно Call Stack

Стеком вызовов (call stack) называется последовательность вызовов, которые приводят к текущему состоянию выполнения. Функции, которые были вызваны и возвращены прежде, не представлены в стеке вызовов.

Окно Call Stack отображает стек вызовов, в котором наиболее свежие вызовы функций располагаются в верхней части списка. Вместе с контрольными точками по условиям окно Call Stack предоставляет очень полезную информацию о функции, содержащей текущую контрольную точку. Этот особенно удобно, если функция вызывается в нескольких местах приложения.

Для отображения функции, содержащейся в окне Call Stack, в окне редактора кода Code Editor нужно дважды щелкнуть мышью на этой функции. Если код функции отсутствует (например, если она располагается во внешнем модуле), то ее дизассемблированный машинный код будет представлен в окне CPU. Как бы то ни было, в окне стека вызовов будет показано следующее выражение или инструкция, которые должны быть выполнены на этом уровне.

Новинкой в C++Builder 5 является возможность просматривать в окне Local Variables локальные переменные отдельной функции в стеке вызовов с помощью команды View Locals контекстного меню.

Окно Threads

Отладка приложений с многими процессами и потоками может быть очень трудным и длительным занятием. Дело в том, что потоки обычно выполняются в асинхронном режиме. Часто потоки приложения взаимодействуют друг с другом с помощью Win32 API-функции `PostThreadMessage()` или используют мьютексы для получения доступа к совместно используемым ресурсам.

При отладке многопотокового приложения можно приостанавливать отдельный поток. Один поток может быть остановлен с помощью контрольной точки, а другой — нет. Проблемы могут возникнуть, если другой поток еще выполняется и основан на взаимодействии потоков или если приостановленный поток имеет открытый мьютекс, закрытия которого ожидает другой поток.

Даже замедление работы приложения в режиме отладки может привести к возникновению проблем с хронометражом (timing) многопоточного приложения. Вообще, не рекомендуется строго хронометрировать приложение, потому что нельзя управлять средой, в которой выполняется приложение.

Окно Threads позволяет устранить некоторые из этих трудностей за счет создания снимка о текущем состоянии процессов и потоков приложения. Каждый процесс имеет основной поток и может иметь несколько дополнительных потоков. Окно Threads отображает потоки в иерархическом виде с группированием потоков по процессам. Первыми в ней указаны первый процесс и основной поток. Для каждого процесса показаны его имя и идентификатор, идентификатор потока, а также его состояние, статус и расположение. На рис. 7.5 показан пример окна Threads.

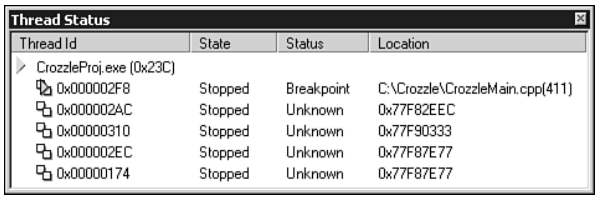


Рис. 7.5. Окно Threads с отображением процесса с четырьмя потоками

Состояние вторичных процессов может быть производным (Spawned), присоединенным (Attached) или присоединенным межпроцессовым (Cross-Process Attach). Процесс может находиться в состоянии выполнения (Runnable), приостановленном состоянии (Stopped), блокированном состоянии (Blocked), или в состоянии None. Адрес потока содержит сведения о текущем расположении этого потока в исходном коде. Если исходный код недоступен, то будет показан текущий адрес.

При отладке приложений с большим количеством процессов или потоков всегда существует один текущий поток. Текущим называется процесс, к которому относится данный поток. Текущий процесс и текущий поток обозначены в окне Threads зеленой стрелкой, как показано на рис. 7.5. Большинство действий и представлений отладчика относятся к текущему потоку. Текущий процесс и текущий поток могут быть изменены с помощью выбора другого процесса или потока в окне Threads и выбора команды Make Current из контекстного меню, с помощью которого можно также прекратить выполнение процесса. Более подробные сведения о дополнительных параметрах и установках окна Threads можно найти в интерактивной справке C++Builder, выполняя поиск по ключевым словам Thread status box.

Окно Modules

В окне Modules перечислены все DLL-модули и пакеты, загруженные вместе с текущим приложением, или модули, в которых была задана контрольная точка по загрузке модуля до выполнения приложения. Как уже говорилось, это очень удобный инструмент для отладки DLL-модулей и пакетов. На рис. 7.6 показано типичное содержимое окна Modules.

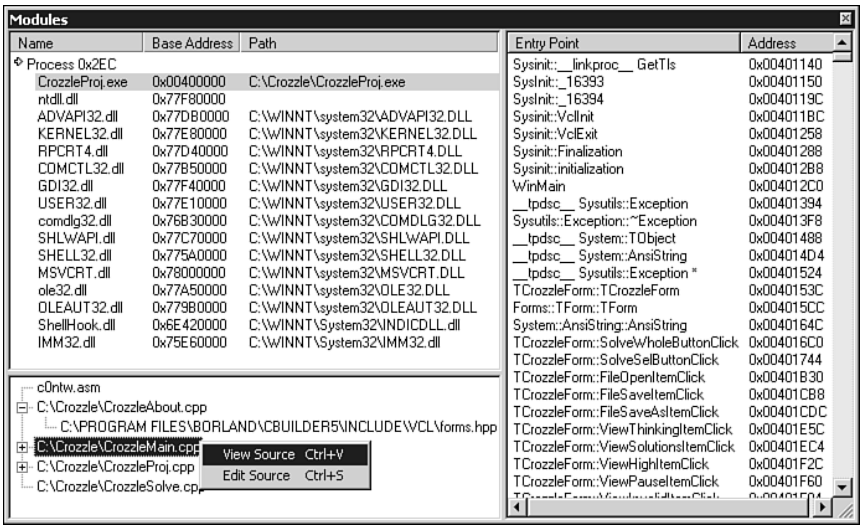


Рис. 7.6. Окно Modules с перечислением модулей, файлов с исходным кодом точек входа

Окно Modules имеет три панели. Верхняя левая панель содержит список модулей, их базовые адреса, а также полные пути к ним. Базовым считается адрес — это адрес, по которому фактически загружается модуль, причем базовый адрес не обязательно указывается во вкладке Linker окна параметров проекта Project Options при создании этого модуля. Выбрав модуль, можно установить контрольную точку загрузки модуля с помощью соответствующей команды контекстного меню.

В нижней левой панели располагается иерархическое представление структуры файлов с исходным кодом, которые используются для создания модуля. Здесь можно выбрать нужный файл и просмотреть его код в окне редактора кода, выбрав команду View Source из контекстного меню.

В правой панели перечислены точки входа модуля и их адреса. С помощью контекстного меню можно перейти к нужной точке входа. При наличии для точки входа исходного кода он будет показан в окне редактора кода. Если исходного кода нет, то точка входа будет отображена в окне CPU.

Окно FPU

Новинкой в C++Builder 5 является окно FPU (Floating-Point Unit — FPU). Оно позволяет просматривать состояние модуля обработки чисел с плавающей запятой или информацию об инструкциях MMX при отладке приложения. На рис. 7.7 показан пример окна FPU с отображением состояния модуля обработки чисел с плавающей запятой.

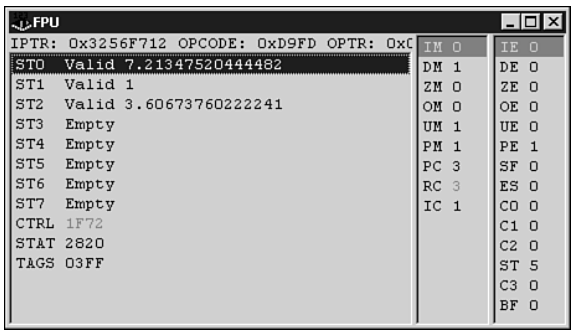


Рис. 7.7. Новое окно FPU в C++Builder 5 с отображением состояния модуля обработки чисел с плавающей запятой

Окно FPU имеет три панели. В левой — показано состояние стека регистров ST0–ST7, слова управления, слова статуса, а также слова-дескриптора. Состояние регистра и его значение показаны в ней для всего стека регистров модуля обработки чисел с плавающей запятой. Статус может иметь значения Empty (Пусто), Zero (Нуль), Valid (Истинно) или Spec. (Специальный), в зависимости от содержимого стека регистров.

Если стек регистров не является пустым (Empty), то в панели будет показано его значение. Формат представления значения (long double или word) можно менять с помощью соответствующей команды Display As контекстного меню. Кроме того, с помощью контекстного меню можно задать нулевое, пустое или конкретное значение для отдельного регистра стека, а также нуль или конкретное значение для слова управления, слова статуса и слова дескриптора.

Средняя панель содержит одноразрядные и многоразрядные флаги управления, которые изменяются при выполнении инструкций обработки чисел с плавающей запятой. Формат их значений можно менять, а сами значения просматривать с помощью соответствующих команд контекстного меню.

Правая панель содержит флаги статуса модуля для вычислений чисел с плавающей запятой. Это расширенное представление слова статуса из панели регистров модуля для обработки чисел с плавающей запятой, с перечислением всех флагов статуса. Формат их значений можно менять, а сами значения просматривать с помощью соответствующих команд контекстного меню.

При изменении значения оно будет окрашено красным цветом во всех панелях. Этот эффект лучше всего заметен при пошаговом выполнении инструкций обработки чисел с плавающей запятой в окне CPU.

Просмотр выражений, их оценка и изменение

Просмотр (watch) выражения представляет собой способ наблюдения за содержимым выражения в процессе отладки. В качестве такого выражения может применяться имя переменной или сложное выражение, которое включает указатели, массивы, функции, значения и переменные. Эти выражения и их значения приводятся в списке просмотра значений окна Watches, которое отображается на экране с помощью команды меню View⇨Debug Windows⇨Watches. Окно Watches может автоматически отображаться на экране при создании нового просматриваемого выражения.

Выражения могут добавляться в список просмотра с помощью трех методов. Первый метод — использование команды Add Watch контекстного меню окна Watches. Второй метод — выбор команды меню Run⇨Add Watch. Третий — щелчок правой кнопкой мыши на соответствующем выражении в окне редактора кода и выбор команды Debug⇨Add Watch at Cursor контекстного меню. При выборе последнего метода выражение будет введено автоматически.

Способы просмотра выражения можно редактировать, задавать и отменять с помощью команд контекстного меню. Их можно удалить, выбрав соответствующее выражение и нажав клавишу <Delete> или выбрав команду Delete Watch контекстного меню. Если выражение не может быть оценено из-за того, что одна или несколько его частей выходят за пределы области просмотра, то вместо результата оценки будет отображено сообщение `undefined symbol` (неопределенное значение).

Возможность оценки и изменения выражений позволяет гибко управлять выражениями, просматривать результат, а также изменять переменные во время выполнения. С помощью команд Evaluate и Modify можно эффективно выполнять подробное интерактивное тестирование, которое достаточно трудно выполняется другими средствами.

Для применения команд Evaluate и Modify нужно приостановить работу приложения одним из двух способов. Первый способ заключается в выборе команды Run⇨Evaluate/Modify и вводе оцениваемого выражения. Но проще это сделать, вызвав команду Evaluate/Modify в окне редактора кода.

После остановки приложения в режиме отладки заданные выражения можно оценить в исходном коде, располагая над ними указатель мыши. Команда Evaluate/Modify позволяет нужным образом изменить выражение. Ее можно вызвать, щелкнув кнопкой мыши на нужном выражении и выбрав команду Debug⇨Evaluate/Modify контекстного меню. После этого в окне Evaluate/Modify будет показано выражение и его результат. Поле Modify позволяет изменять значение выражения для простых типов данных. Для изменения значения структуры или массива нужно по отдельности изменить значение каждого составного элемента.

В выражения можно включить вызовы функций. Учтите, что оценка выражений дает такой же результат, как и оценка этого выражения во время выполнения приложения. Если выражение приводит к появлению побочных эффектов, они будут отражены в состоянии выполнения приложения в ходе дальнейшего пошагового или обычного его выполнения.

В отличие от окна Watches, в диалоговом окне Evaluate/Modify результат выражения не обновляется автоматически при пошаговом выполнении кода. Для этого нужно щелкнуть на кнопке Evaluate. Результат выполнения выражения можно отформатировать, указав специальный модификатор в конце строки с выражением. Более подробную информацию об этом можно найти в интерактивной справке.

Окно Evaluate/Modify обычно используется для проверки ошибочных условий и отслеживания причин появления ошибок. Для проверки ошибочного условия нужно установить контрольную точку в месте возникновения ошибки или непосредственно перед ним либо подойти к этому месту с помощью пошагового выполнения кода, а затем принудительно вызвать появление ошибки, задав соответствующее ошибочное значение с помощью команды Modify. Для проверки правильности обработки ошибки используйте команду Single Step или Run.

Если у вас есть подозрения, что некоторый фрагмент кода содержит ошибку и задает неверные данные, установите контрольную точку сразу после этого фрагмента кода, исправьте данные вручную, а затем продолжите выполнение программы. Так вы сможете убедиться, что эти неверные данные приводят к появлению ошибки. Проследите за всеми предыдущими строками кода вплоть до обнаружения ошибки или используйте контрольную точку по значению данных для поиска того места, где происходит ошибочное изменение данных.

Инспектор отладки *Debug Inspector*

Инспектор отладки Debug Inspector работает так же, как инспектор объектов времени выполнения. Он может использоваться для отображения данных, методов, свойств классов, структур, массивов, функций, а также простых типов данных во время выполнения программы, предоставляя удобный универсальный инструмент для просмотра/изменения значений выражения.

Запустить инспектор отладки Debug Inspector можно после приостановки приложения либо с помощью команды меню Run⇨Inspect и ввода проверяемого выражения, либо с помощью щелчка правой кнопкой мыши на нем в окне редактора кода и выбора команды Debug⇨Inspect контекстного меню. В втором случае проверяемое выражение будет автоматически введено в окно Debug Inspector.

На рис. 7.8 показаны членов класса формы в окне Debug Inspector.

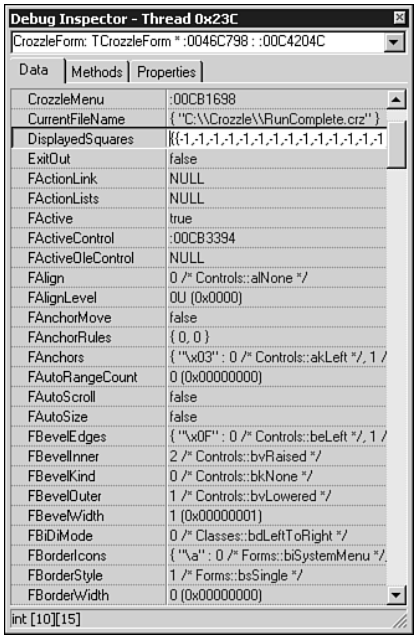


Рис. 7.8. Члены класса формы в окне Debug Inspector

В заголовке окна Debug Inspector приводится идентификатор потока. В верхней части окна содержится имя, тип и адрес рассматриваемого элемента. В зависимости от типа элемента, в этом окне будет представлено до трех вкладок с именем, содержимым или адресом каждого элемента данных, метода или свойства. Вкладка Property показана только для классов, производных от классов библиотеки VCL. В нижней части окна показан тип текущего выбранного элемента.

Значения данных простых типов можно изменять; в таком случае возле элемента располагается многоточие. Для изменения значения щелкните на многоточии и введите новое значение.

Окно Debug Inspector можно применять для трассировки структуры данных и классов. Для проверки одного из членов данных, методов или свойств в текущем окне Object Inspector нужно выбрать его, а потом выбрать команду Inspect из контекстного меню. При этом можно также скрывать или отображать элементы-наследники.

Помимо имеющегося параметра Inspectors Stay On Top (Показать окно Object Inspector поверх других окон), в C++Builder 5 добавлены три новых параметра инспектора объекта: Show Inherited (Показать элементы-наследники), Sort By Name (Сортировать по имени) и Show Fully Qualified Names (Показать имена полностью), которые могут быть заданы с помощью команды меню Tools⇨Debugger Options. Параметр Show Inherited переключает режим просмотра данных во вкладках Data, Methods и Properties: в одном показаны все внутренние и унаследованные члены-данные или свойства класса, в другом — только те, которые объявлены в этом классе. Па-

раметр Sort By Name регулирует способ сортировки элементов по имени или в порядке их объявления. Параметр Show Fully Qualified Names позволяет привести для унаследованных членов их полное имя и отобразить его, задавая параметр Show Inherited. Все три новых параметра могут быть заданы с помощью команд контекстного меню окна инспектора отладки Debug Inspector.

Окно Debug Inspector широко используется для отладки, потому что в нем можно одновременно отобразить сразу несколько элементов. Оно также позволяет последовательно просматривать иерархическую структуру класса и данных.

Инструмент *CodeGuard*

Инструмент ***CodeGuard*** является новинкой профессиональной (Professional) и корпоративной (Enterprise) версий C++Builder 5. Он уже использовался в Borland C++ и теперь входит в состав C++Builder. Инструмент ***CodeGuard*** контролирует расходование оперативной памяти и других ресурсов, а также проверяет вызовы функций при выполнении приложения.

Инструмент ***CodeGuard*** может обнаружить следующие типы ошибок во время выполнения приложения:

- неверное освобождение памяти,
- неверное использование файлов и потоков,
- неверные указатели,
- использование освобожденной памяти,
- утечка памяти,
- использование выделенной памяти, которая не была освобождена,
- неправильно переданные аргументы функциям Borland и Win32 API-функциям
- функции Borland и Win32 API-функции, которые возвращают ошибочное значение,
- неверные дескрипторы ресурсов, которые переданы функциям Borland и Win32 API-функциям.

Примером неверного использования памяти является попытка приложения несколько раз освободить какой-либо ресурс или осуществить доступ к области памяти, которая была освобождена. В следующих разделах рассматриваются причины возникновения таких ошибок.

Инструмент ***CodeGuard*** выводит сообщения о найденных ошибках в файл журнала, содержимое которого можно просматривать в IDE-среде C++Builder. Инструмент CodeGuard также обладает возможностями непосредственного доступа к строке кода, которая привела к возникновению ошибки.

Применение и конфигурирование инструмента *CodeGuard*

Для применения инструмента ***CodeGuard*** необходимо включить его в состав откомпилированного кода приложения. Для этого нужно установить флажок параметра CodeGuard Validation во вкладке CodeGuard в окне параметров проекта Project Options. Для идентификации номера строки, в которой произошла ошибка, необходимо также установить флажки параметров Debug Information и Line Number Information во вкладке Compiler в окне параметров проекта Project Options. Перекомпилируйте проект заново с помощью команд Build или Build All Projects.

Во вкладке CodeGuard также имеются три других параметра, которые показаны на рис. 7.9. Первый параметр Validate global and stack accesses используется для обнаруже-

ния неверных указателей на локальные, глобальные и статичные переменные и нарушения границ данных. Второй параметр Validate the 'this' pointer on member function only используется для обнаружения вызовов на неверные или удаленные объекты. Третий параметр Validate pointer accesses позволяет инструменту ***CodeGuard*** проверить правильность встраиваемых указателей и существенно замедляет скорость выполнения программы. Обычно приходится использовать все три параметра. При изменении какого-либо из них придется также перекомпилировать код приложения с помощью команды Build или Build All Projects.

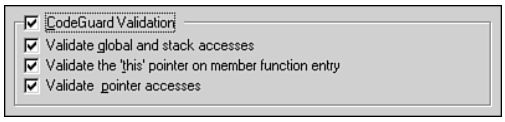


Рис. 7.9. Параметры инструмента CodeGuard

Многие параметры инструмента CodeGuard могут быть заданы в окне CodeGuard Configuration, которое вызывается с помощью команды меню Tools⇨CodeGuard Configuration, или в режиме командной строки с помощью утилиты `CGCONFIG.EXE`. Все параметры конфигурирования собраны в четырех вкладках.

Во вкладке Preferences, которая показана на рис. 7.10, задаются общие параметры инструмента CodeGuard. Параметр Enable активирует инструмент CodeGuard без перекомпиляции кода приложения. Флажок этого параметра следует установить для включения режима проверки наличия ошибок с помощью ***CodeGuard*** во время выполнения приложения. Для полного отключения инструмента ***CodeGuard*** нужно снять флажок параметра CodeGuard Validation во вкладке CodeGuard окна параметров проекта и заново перекомпилировать код приложения с помощью команд Build или Build All Projects. Параметр Stack fill frequency регулирует соотношение между тщательностью проверки наличия ошибок и скоростью выполнения при неверном доступе к стеку во время выполнения приложения. Для него рекомендуется задать значение 2.

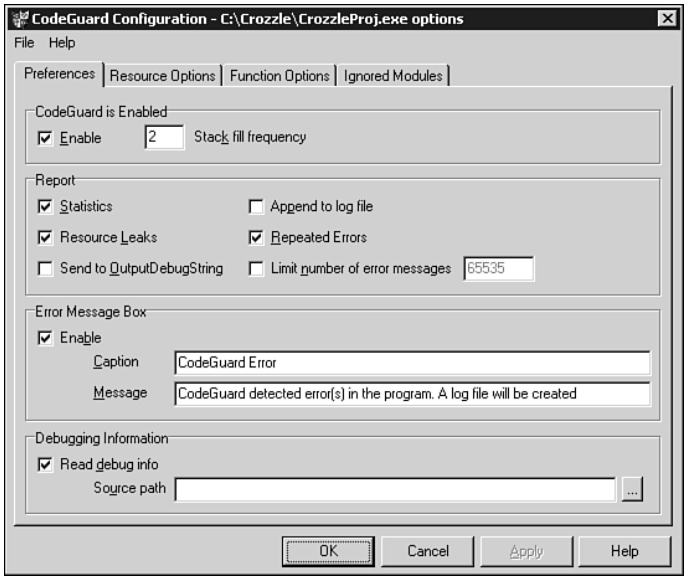


Рис. 7.10. Вкладка Preferences окна CodeGuard Configuration

Группы параметров Report и Error Message Box определяют способ представления отчетов о найденных ошибках. В группе параметров Report параметр Statistics регулирует вывод статистических данных о выделении и освобождении оперативной памяти, вызовах избранных Win32 API-функций, использовании ресурсов, а также списка модулей в конце файла журнала. Параметр Resource Leaks регламентирует вывод в отчете сведений об обнаруженной утечке ресурсов сразу после завершения работы приложения. Группа параметров Error Message Box предназначена для создания диалогового окна с сообщением об ошибке, обнаруженной с помощью инструмента CodeGuard, в приложении, которое выполняется вне IDE-среды. Более подробные сведения о параметрах вкладки Preferences можно найти в интерактивной справке C++Builder.

Вкладки Resource Options и Function Options окна CodeGuard Configuration позволяют задать параметры трассировки ресурсов, дескрипторов файлов, а также вызовов функций. Без крайней необходимости не стоит изменять предлагаемые по умолчанию значения этих параметров. Здесь же стоит упомянуть об одном из наиболее полезных параметров вкладки Function Options, который предназначен для регистрации каждого вызова особых функций.

Во вкладке Ignored Modules (Игнорируемые модули) можно сообщить инструменту ***CodeGuard*** о тех DLL-модулях или пакетах, которые можно пропустить при поиске ошибок во время выполнения приложения. Этот параметр следует использовать только в случае крайней необходимости.

Применение инструмента *CodeGuard*

Работа с инструментом ***CodeGuard*** начинается с его включения и конфигурирования (как описано в предыдущем разделе), и последующего запуска приложения. Инструмент ***CodeGuard*** будет отслеживать все заданные аспекты работы приложения как при работе внутри IDE-среды C++Builder, так и вне ее. Он также будет регистрировать замеченные ошибки в файле журнала учета ошибок под именем `<имя_проекта>.cgl`. Он представляет собой текстовый файл, который можно просматривать и редактировать с помощью любого стандартного текстового редактора. Но все же лучше всего для этого использовать IDE-среду C++Builder.

Для просмотра журнала учета ошибок инструмента ***CodeGuard*** в IDE-среде нужно выбрать команду меню View⇨Debug Windows⇨CodeGuard Log или нажать комбинацию клавиш <Ctrl+Alt+O>. C++Builder интерпретирует информацию из журнала учета ошибок инструмента ***CodeGuard*** и отображает ее в более удобном для пользователя иерархическом виде. Все ошибки сгруппированы по типам, и их можно просматривать, раскрывая списки каждого типа. Эта информация включает сведения о месте использования, выделения и освобождения ресурса; о состоянии стека вызовов во время возникновения ошибки; а также об указателях на стро

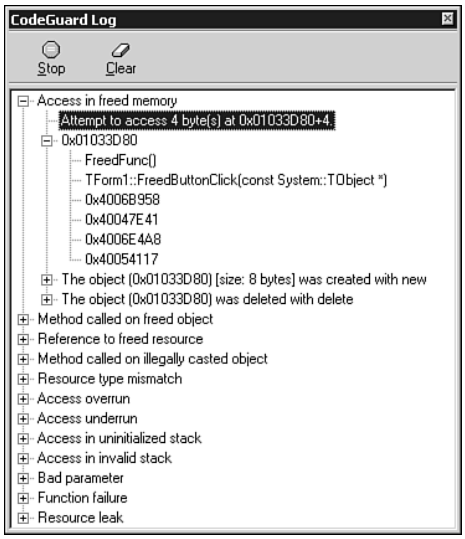


Рис. 7.11. Окно просмотра отладочной информации CodeGuard Log с сообщением об ошибке при доступе к уже освобожденной памяти Access In Freed Memory

ки исходного кода, в которых она произошла. На рис. 7.11 показан пример использования окна CodeGuard Log для просмотра информации об ошибке при доступе к уже освобожденной памяти Access In Freed Memory.

В окне CodeGuard Log можно заметить две кнопки: Stop и Clear. Если кнопка Stop нажата, инструмент ***CodeGuard*** прекратит выполнение приложения в случае обнаружения ошибки. А если — нет, то работа приложения будет продолжена и все последующие ошибки зарегистрированы. Если кнопка Clear нажата, инструмент ***CodeGuard*** будет очищать содержимое журнала учета ошибок перед каждой попыткой запуска приложения.

Для перехода к строке исходного кода, которая соответствует данной ошибке, нужно дважды щелкнуть на узле с информацией об ошибке в окне CodeGuard Log или на адресе памяти в окне CPU, если исходный код недоступен. Для этого также можно использовать команду контекстного меню View Source. Например, после двойного щелчка на строке `Attempt to Access 4 Byte(s)`, показанной на рис. 7.11, будет совершен переход к строке кода, в которой ошибочно был осуществлен доступ к уже освобожденной области памяти. Кроме того, дважды щелкнув на первой функции в строке кода с информацией об ошибке, в которой приведено содержимое стека вызовов, также можно совершить переход к соответствующей строке кода.

После обнаружения ошибки с помощью инструмента ***CodeGuard*** и перехода к соответствующей строке кода остается только исправить эту ошибку, используя правильный адрес или функцию или отслеживая происхождение ресурса или аргументов функции до обнаружения причин происхождения ошибки. Для упрощения этого процесса следует использовать контрольные точки по значению данных и средства просмотра выражений.

Анализ ошибок и причин их возникновения с помощью *CodeGuard*

Инструмент ***CodeGuard*** способен обнаружить множество ошибок времени выполнения. Ниже кратко рассматриваются сообщения об ошибках инструмента CodeGuard и их примеры. Найденная с помощью инструмента CodeGuard ошибка очевидна, соответствующую ей строку исходного кода легко найти, кроме того, такую ошибку легко исправить.

Все приведенные ниже ошибки, найденные с помощью инструмента ***CodeGuard***, демонстрируются на примере приложения, которое находится в папке `CodeGuard` на прилагаемом к книге компакт-диске. Это приложение представляет собой простую форму с кнопками, которые вызывают появление разных ошибок времени выполнения. Перед началом работы с этим приложением рекомендуется отобразить на экране окно журнала учета ошибок CodeGuard Log с ненажатой кнопкой Stop.

Для большинства ошибок инструмент ***CodeGuard*** отобразит состояние стека вызовов и укажет подходящую функцию в верхней части списка, место возникновения ошибки, а также место выделения и освобождения ресурса, если таковой имеется. Для этого ресурса также будет приведено количество запрашиваемых и выделенных байт, если это имело место.

Доступ к уже освобожденной области памяти

Ошибка типа `Access In Freed Memory` происходит при попытке доступа к уже освобожденной области памяти. Обычно память выделяется с помощью оператора `new` или `malloc`, а освобождается с помощью оператора `delete` или `free`. Приведенный ниже код демонстрирует пример доступа к уже освобожденной области памяти.

```
#include <stdio.h>
#include <dir.h>

class TSomeClass {
```

```
    int FNumber;
public:
    int GetNumber() { return FNumber; }
    void SetNumber(int NewNumber) { FNumber = NewNumber; }
    int Double(int Val) { return Val*2; }
    int PubVal;
};

void MyFunc()
{
    TSomeClass *MyClass = new TSomeClass;
    delete MyClass;
    MyClass->PubVal = 10; // Область памяти для объекта MyClass
                          // уже освобождена.
}
```

В этом случае инструмент ***CodeGuard*** сообщит о месте доступа к освобожденной области памяти, исходное место, где она была выделена прежде, а также место, где она была освобождена.

Приведенные ниже ошибки, найденные с помощью инструмента ***CodeGuard***, будут снова продемонстрированы на примере измененной функции `MyFunc()` из предыдущего фрагмента кода.

Вызов метода объекта, область памяти которого уже освобождена

Ошибка типа `Method Called On Freed Object` аналогична ошибке типа `Access In Freed Memory` и вызвана попыткой вызова метода объекта, область памяти которого уже освобождена, а не попыткой доступа к самой этой памяти.

```
void MyFunc()
{
   TSomeClass *MyClass = new TSomeClass;
   int Answer;
   delete MyClass;
   Answer = MyClass->Double(5);
}
```

Инструмент CodeGuard сообщит о месте вызова метода объекта, область памяти которого уже освобождена, месте создания объекта и месте освобождения области памяти этого объекта.

Ссылка на освобожденный ресурс

Ошибка типа `Reference To Freed Resource` возникает при попытке повторного доступа к уже освобожденному ресурсу. Существует несколько причин возникновения этой ошибки, и приведенный ниже фрагмент кода демонстрирует одну из них, а другую можно найти в функции `ReferenceButtonClick()` из примера приложения, который находится на прилагаемом к книге компакт-диске.

```
void MyFunc()
{
   TSomeClass *MyClass = new TSomeClass[10];
   delete[] MyClass;;
   delete[] MyClass;;
}
```

В этом случае инструмент ***CodeGuard*** сообщит о месте повторной попытки освобождения ресурса, которая вызвала появления ошибки, а также о месте выделения и первого освобождения этого ресурса.

Вызов метода объекта с неправильным приведением типа

Вызов метода вне заданного диапазона памяти приведет к появлению ошибки типа `Method Called On Illegally Casted Object`. В приведенном ниже примере создается массив из двух объектов типа `TSomeClass`, но с попыткой вызова метода третьего объекта этого массива. Напомним, что нумерация членов массива начинается с нуля.

```
void MyFunc()
{
   TSomeClass *MyClass = new TSomeClass[2];
   int Answer;
   Answer = MyClass[2].Double(5); // В массиве MyClass нет
                                  // третьего элемента MyClass[2]
   delete[] MyClass;;
}
```

В этом случае инструмент ***CodeGuard*** сообщит о месте определения метода неверного вызова объекта, месте его вызова, а также о месте выделения объекта (его памяти).

Несоответствие типа ресурса

Ошибка типа `Resource Type Mismatch` возникает при освобождении ресурса способом, отличным от способа его выделения (например. с использованием оператора `free` для памяти, выделенной с помощью оператора `new`). На самом деле для корректного освобождения этой памяти следует использовать оператор `delete`. Эта ошибка может быть вызвана несколькими причинами, две из которых приведены ниже, а другие можно найти в исходном коде примера приложения на прилагаемом компакт-диске.

```
void MyFunc()
{
   TSomeClass *MyClass2 = new TSomeClass;
   delete[] MyClass2;
   TSomeClass *MyClass3 = new TSomeClass[2];
   delete MyClass3;
}
```

В этом случае инструмент ***CodeGuard*** сообщит о месте противоречивого освобождения и месте исходного выделения ресурса.

Доступ вне верхней границы

Ошибка типа `Access Overrun` возникает при попытке доступа к области памяти за пределами ее верхней границы. Например, при доступе к третьему элементу массива из двух элементов или копировании данных за пределами их области памяти. Оба эти случая продемонстрированы в следующем примере кода.

```
void MyFunc()
{
   TSomeClass *MyClass = new TSomeClass[2];
   MyClass[2].PubVal = 10; // В массиве MyClass нет
                           // третьего элемента MyClass[2]
   delete[] MyClass;;
   char *CharList = new char[10];
   strcpy(CharList, "1234567890"); // Значение NULL в конце
```

```
                                    // строки выходит за границы.
   delete[] CharList;;
}
```

В этом случае инструмент ***CodeGuard*** сообщит о месте выхода за пределы верхней границы и месте исходного выделения ресурса.

Доступ вне нижней границы

Ошибка типа `Access Underrun` аналогична ошибке типа `Access Overrun`, за исключением того, что неверная попытка доступа к памяти возникает за пределами нижней, а не верхней границы.

```
void MyFunc()
{
   int *IntList = new int[2];
   IntList [-1] == 10;
   delete[] IntList;;
}
```

В этом случае инструмент ***CodeGuard*** сообщит о месте выхода за пределы нижней границы и месте исходного выделения ресурса.

Доступ к неинициализированному стеку

Ошибка типа `Access In Uninitialized Stack` возникает при попытке доступа к неинициализированной области стека. В следующем фрагменте кода функция возвращает указатель на локальную переменную в стеке. Но при возвращении функции эта часть стека уже не существует, а потому при попытке доступа к ней возникает ошибка.

```
void LocFunc(int **LocPtr)
{
   int LocalVar;
   *LocPtr = &LocalVar;
}

void MyFunc()
{
   int *LocPtr;
   LocFunc(&LocPtr);
   *LocPtr = 10;
}
```

В этом случае инструмент ***CodeGuard*** сообщит о месте доступа к неинициализированному стеку.

Доступ за пределами стека

Ошибка типа `Access In Invalid Stack` происходит при попытке доступа ниже нижней границы стека. В следующем примере попытка доступа совершается ниже границы массива Name, который хранится в нижней части стека. Этот тип ошибки отличается от ошибки типа `Access Underrun` тем, что предпринимается попытка доступа к стеку в пределах выделенной области памяти.

```
void MyFunc()
{
```

```
   char Name[20];
   strcpy(&Name[-1], "Someone');
}
```

В этом случае инструмент ***CodeGuard*** сообщит о месте доступа за пределами нижней границы стека.

Неверный параметр

Ошибка типа `Bad Parameter` обычно возникает при попытке передать неверный дескриптор файла или другого ресурса стандартной функции Borland или Win32 API-функции. В следующем фрагменте кода указан неверный дескриптор файла потока при вызове функции `fclose`, потому что такой файл не был открыт.

```
void MyFunc()
{
   FILE *Stream;
   fclose(Stream);
}
```

В этом случае инструмент ***CodeGuard*** сообщит о месте вызова функции с неверно заданным параметром.

Сбой функции

Инструмент ***CodeGuard*** отслеживает возвращаемое значение многих стандартных функций Borland и Win32 API-функций. Возвращаемое значение `-1` означает, что при вызове функции произошла ошибка, которая будет учтена инструментом ***CodeGuard***. В следующем фрагменте кода вызывается функция `chdir()` с неверным именем каталога, что приведет к возврату значения `-1`.

```
void MyFunc()
{
   chdir("Z:\ZXCVBN");
}
```

В этом случае инструмент ***CodeGuard*** сообщит о месте вызова функции, вернувшей значение `-1`.

Утечка ресурсов

Ошибка типа `Resource Leak` возникает при выделении ресурса (например, памяти), но без последующего освобождения. Для большинства объектов библиотеки VCL освобождение памяти выполняется автоматически, а потому утечка ресурсов наиболее часто возникает при выделении разработчиком области памяти или при динамическом создании собственного объекта. Второй случай иллюстрируется приведенным ниже фрагментом кода: для созданного объекта память выделяется, но не освобождается.

```
void MyFunc()
{
   TSomeClass *MyClass = new TSomeClass;
}
```

В этом случае инструмент ***CodeGuard*** сообщит о месте создания такого ресурса.

Здесь описаны только самые распространенные ошибки, которые могут быть найдены с помощью инструмента ***CodeGuard***. Дополнительные сведения о других ошибках и их примеры можно найти в интерактивной справке C++Builder.

Более сложные методы отладки

Как уже говорилось выше, сама по себе отладка представляет собой очень сложную тему. Кроме того, существуют особые вопросы, которые выходят за пределы базовых методов отладки, представленных в первом разделе этой главы.

Для любого серьезного проекта по созданию и отладке приложения рекомендуется использовать операционную среду Windows NT (WinNT) или Windows 2000 (Win2K, которая основана на Windows NT), а не Windows 95 или Windows 98 (Win9x). WinNT и Win2K представляют собой более стабильную среду, особенно при работе с приложениями, которые содержат ошибки.

Операционные системы на основе WinNT справляются с остановом и сбоем приложения гораздо лучше, чем Win9x. Дело в том, что в операционной системе Win9x среда разработки C++Builder или даже вся система в целом могут легко выйти из строя при отладке или внезапной остановке приложения. Рекомендуется как можно реже применять команду меню Run⇨Program Reset. Лучше прекращать работу приложения с помощью средств, доступных пользователю.

Если приложение выполняет запрещенную операцию и нарушает права доступа во время ее запуска в IDE-среде C++Builder, возникает исключительная ситуация и на экране появляется диалоговое окно с сообщением о ней. В операционной системе Win9x при этом следует сбросить приложение с помощью команды меню Run⇨Program Reset перед закрытием этого диалогового окна. Это позволяет восстановить прежнее состояние более надежно, чем при применении этой команды после закрытия диалогового окна.

Для действительно серьезной отладки, особенно системных приложений Windows, можно использовать отладочную версию операционной системы Windows, которая называется "отладочным кодом" ("debug binary") для операционной системы Win9x и "контролируемым отладочным кодом" ("checked/debug build") — для WinNT/2K. В контролируемом отладочном коде предусмотрена проверка исключительных ситуаций, проверка аргументов, а также системный отладочный код для операционной системы Windows и Win32 API-функций (в основном в виде утверждений, которых нет в обычной коммерческой версии). Однако эта проверка приводит к общему снижению производительности системы. Контролируемый отладочный код операционной системы Windows включен в состав некоторых видов подписки Microsoft Developer Network (MSDN).

Иногда необходимо знать, работает ли приложение в контексте отладчика. В таком случае Win32 API-функция `IsDebuggerPresent()` возвращает значение `true`. Это можно использовать, чтобы изменить поведение приложения в процессе выполнения, например вывести дополнительную отладочную информацию для упрощения его отладки.

Рассмотрим теперь некоторые более сложные вопросы отладки.

Обнаружение источника нарушения доступа

Ранее в этой главе рассматривались основные методы обнаружения ошибок. Ошибки нарушения доступа или AV-ошибки (access violations — AV) иногда гораздо сложнее обнаружить, чем ошибки общего типа. Ниже описываются методы отладки приложения при наличии ошибок этого типа и аналогичных по трудности ошибок другого типа.

Нарушения доступа могут возникать при попытке доступа к области памяти, которая выходит за рамки адресного пространства приложения. Для обнаружения этого явления во время работы приложения следует использовать инструмент ***CodeGuard***.

Инструмент ***CodeGuard*** может обнаружить многие ошибки, которые обычно являются следствием AV-ошибки, и указать на соответствующую строку кода, которая стала причиной такой ошибки. Если инструмент ***CodeGuard*** не используется в вашем приложении или он не

смог обнаружить ошибку, которая привела к нарушению доступа, то можно воспользоваться другими способами поиска такой ошибки.

При возникновении AV-ошибки, на экране появится диалоговое окно с сообщением `Access violation at address YYYYYYYY. Read of address ZZZZZZZZ.` (Нарушение доступа по адресу YYYYYYYY. Чтение адреса ZZZZZZZZ.). В приложении эта ошибка также может вызвать появление другого сообщения, например `The instruction at 0xYYYYYYYY referenced memory at 0xZZZZZZZZ.` (Инструкция по адресу 0xYYYYYYYY сослалась на адрес 0xZZZZZZZZ.) В этих случаях адрес YYYYYYYY содержит машинный код, вызвавший эту ошибку, а адрес ZZZZZZZZ — это неверный адрес, к которому предпринимается попытка доступа.

Некоторые ошибки доступа можно обнаружить с помощью глобального обработчика исключительных ситуаций. Этот способ рассматривался в начале главы. Кроме того, можно использовать инструменты оперативной (just-in-time — JIT) отладки, которые описываются ниже в этой главе для переноса процесса в отладчик и перехода к адресу YYYYYYYY. Т.е. можно запустить приложение в среде C++Builder и подождать, пока не возникнет AV-ошибка.

Если AV-ошибку не удается воспроизвести при работе в среде C++Builder, то нужно просто приостановить работу приложения с помощью команды меню Run⇨Pause или задавать контрольную точку, затем открыть окно CPU, щелкнуть правой кнопкой и выбрать команду Goto Address из контекстного меню. Это не совсем идеальный способ, но он срабатывает довольно часто. Введите адрес кода из диалогового окна AV-ошибки в шестнадцатеричном формате `0xYYYYYYYY`. Код возле этого адреса может навести вас на мысль о причине возникновения AV-ошибки, особенно если приложение компилировалось с условием вывода отладочной информации.

Если адрес `ZZZZZZZZ` близок к нулю, например равен `00000089`, причина часто заключается в попытке доступа к неинициализированному указателю. Приведенный ниже код вызывает AV-ошибку, потому что объект `MyButton` не был создан с помощью оператора `new`.

```
TButton *MyButton;
MyButton->Height = 10;
```

На самом деле при объявлении указателя `MyButton` он инициализируется значением `0`. А адрес `00000089` является адресом свойства `Height` объекта `TButton`, если бы он располагался по адресу `0`.

Рекомендуется явно инициализировать указатели на известный объект до выделения памяти или создания объекта и возвращаться к нему при освобождении памяти или объекта. При получении сообщения об AV-ошибке с подобным значением адреса, можно с уверенностью сказать, что эта ошибка вызвана неинициализированным указателем.

Иногда AV-ошибка может возникнуть в многопотоковом приложении, в котором не контролируется взаимный доступ к общим объектам и данным. Источник таких ошибок очень трудно найти. Используйте для этого контрольные точки и вывод отладочной информации, как описано выше.

Присоединение к выполняемому процессу

При выполнении процесса вне IDE-среды C++Builder его все же можно отладить с помощью интегрированного отладчика, присоединяя его во время выполнения. Эта способность очень удобна при тестировании. При обнаружении ошибки в работе приложения можно присоединиться к процессу приложения и проследить за этой ошибкой. Единственным недостатком этого метода является то, что в Windows не предусмотрен метод отсоединения от такого процесса без его прекращения.

Для присоединения к выполняемому процессу выберите команду меню Run⇨Attach to Process. При этом откроется диалоговое окно Attach To Process со списком всех выполняемых процессов на данном локальном компьютере. Выберите нужный процесс из этого

списка и щелкните на кнопке Attach. После этого отладчик C++Builder присоединиться к процессу. Процесс будет приостановлен, а в окне CPU показано состояние процессора в текущей точке выполнения. После этого можно приступить к пошаговому выполнению кода, установке контрольных точек, отображению исходного кода (если это возможно) с помощью команды View Source контекстного меню, проверке значений и т.д.

Диалоговое окно Attach To Process может еще более эффективно использоваться для удаленной отладки. В диалоговом окне Attach To Process можно просматривать выполняемые процессы на другом компьютере (удаленном отладочном сервере) и присоединяться к ним. Более подробно эта тема рассматривается ниже.

В этом окне также можно просматривать системные процессы. Для этого нужно установить флажок параметра Show System Processes (Показать системные процессы).

Следует тщательно присоединять старые и системные процессы, потому что это может привести к сбою или зависанию операционной системы Windows. Поэтому без крайней необходимости не рекомендуется присоединять какие-либо процессы, кроме ваших собственных.

Оперативная отладка

Оперативная отладка или JIT-отладка (just-in-time — JIT) предусмотрена в операционных системах Windows NT и Windows 2000 и позволяет отладить процесс при сбое или возникновении ошибки доступа. Однако JIT-отладка не может применяться на компьютерах под управлением операционной системы Windows 9x.

Пользователи Windows NT или Windows 2000, несомненно, слышали о существовании утилиты Dr.Watson, которая является инструментом JIT-отладки в операционной системе Windows и помогает обнаружить причину возникновения ошибки. JIT-отладчик, используемый по умолчанию в операционной системе, можно заменить другим. Текущий инструмент JIT-отладки обычно задается с помощью соответствующего ключа системного реестра. Однако новинкой C++Builder 5 является возможность использования менеджера JIT-отладчиков `BORDBG51.EXE` вместо утилиты Dr.Watson. С его помощью для каждого случая JIT-отладки можно выбрать нужный инструмент с помощью менеджера JIT-отладчиков, например JIT-отладчик C++Builder, JIT-отладчик Delphi, Dr.Watson или даже отладчик Borland Turbo Debugger.

До появления C++Builder версии 5 вызов отладчика Dr.Watson можно было заменить непосредственно вызовом отладчика C++Builder, но без возможности выбора. Если в списке отладчиков присутствует только один отладчик, то именно он и будет запущен автоматически. Более подробные инструкции о способах конфигурирования менеджера отладчиков C++Builder можно найти в оперативной справке C++Builder.

Сразу после конфигурирования менеджера отладчиков можно приступать к JIT-отладке. При сбое приложения Windows запустит менеджер отладчиков. Выберите нужный отладчик из списка, например `BCB` (C++Builder) в данном случае, а потом щелкните на кнопке OK. Будет запущена среда C++Builder (если она еще не была запущена), а работа приложения будет приостановлена. Для обнаружения источника ошибки можно использовать один из описанных выше методов отладки.

Удаленная отладка

Удаленная отладка представляет собой возможность отладки приложения, которое выполняется на другом компьютере с помощью интерактивного отладчика C++Builder, запущенного на локальном компьютере. Этот очень удобно для отладки приложений, запущенных на другом компьютере, например, расположенном в труднодоступном месте. Кроме того, для этого не требуется инсталлировать C++Builder на удаленном компьютере.

Удаленная отладка очень удобна в случае распределенных приложений с использованием технологий DCOM или CORBA. При этом отладка может выполняться локально, если эта работа не очень затрудняется вследствие сокращения производительности, вызванной низкой пропускной способностью сети.

Удаленная отладка может применяться для выполняемых файлов, DLL-модулей и пакетов. Приложение должно компилироваться вместе с отладочной информацией, а файл формата `.tds` с отладочными символами должен быть доступен для приложения на удаленном компьютере. Простейший способ достижения этой цели заключается в загрузке проекта приложения в среду C++Builder на локальном компьютере. Укажите при этом в текстовом окне пути к папке вывода Final output path вкладки Directories/Conditionals в окне параметров проекта Project Options совместно используемую сетевую папку на удаленном компьютере, где запущено приложение, и скомпилируйте приложение вместе с отладочной информацией.

Удаленная отладка приложения выполняется практически прозрачно. Сразу после установления сеанса удаленной отладки можно приступать к работе, как если бы вы выполняли отладку локального приложения.

Конфигурирование удаленной отладки

Удаленная отладка начинается с запуска сервера отладки `BORDBG51.EXE` на удаленном компьютере. Можно заметить, что функции сервера отладки Borland выполняет менеджер JIT-отладчиков, который описан в предыдущем разделе этой главы. Он может выполнять любые из этих функций, в зависимости от параметров командной строки, заданных при его запуске. Сервер отладки требует инсталляции дополнительных DLL-модулей, а локальный отладчик C++Builder взаимодействует с сервером отладки.

На удаленных компьютерах под управлением операционных систем Windows NT и Windows 2000 сервер отладки обычно инсталлируется в виде сервиса, который можно найти под именем `Borland Remote Debugging Service` в окне Services системной панели управления Control Panel. Сервис сервера отладки может запускаться вручную или автоматически при запуске операционной системы, а также останавливаться в этом окне, как и всякий обычный сервис. Для инсталляции и удаления этого сервиса следует использовать параметры командной строки `-install` и `-remove`, соответственно.

На удаленных компьютерах под управлением операционной системы Windows 9x сервер отладки является изолированным процессом. Этот способ работы также можно применять на компьютерах под управлением операционной системы WinNT/2K. Во всех случаях удаленный сервер отладки должен быть запущен до начала отладки.

Сервер отладки вместе с необходимыми DLL-модулями можно инсталлировать с помощью дистрибутивного компакт-диска C++Builder и стандартного диалогового окна инсталляции либо запустить программу `SETUP.EXE` из каталога `RDEBUG` этого компакт-диска. При удаленной отладке для сообщения между локальной IDE-средой C++Builder и удаленным сервером отладки используется протокол TCP/IP. Для этого необходимо соответствующим образом сконфигурировать протокол TCP/IP на обоих компьютерах.

Для запуска сервера отладки вручную следует ввести в командной строке `BORDBG51.EXE -listen`. Для этого вам понадобятся права администрирования или отладки.

Применение удаленной отладки

К удаленной отладке можно приступать после инсталляции и запуска сервера отладки на удаленном компьютере. Для этого в IDE-среде C++Builder локального компьютера откройте проект удаленного приложения, который необходимо отладить. Выберите команду меню Run⇨Parameters, затем — вкладку Remote и с помощью параметра Remote Path задайте

полный путь к удаленному приложению и имя файла этого приложения точно так же, как если бы оно находилось на локальном компьютере, например `C:\Temp\MyProj.exe`. При отладке DLL-модуля на удаленном компьютере введите путь и имя удаленного приложения, к которому относится DLL-модуль. Введите необходимые параметры командной строки приложения в поле Parameters. В поле Remote Host укажите имя или IP-адрес удаленного компьютера.

Прежде чем начать отладку удаленного приложения, нужно загрузить его проект в среду C++Builder, щелкнув на кнопке Load. Если проект этого приложения уже загружен, установите флажок параметра Debug Project On Remote Machine (Отладка проекта на удаленном компьютере) и щелкните на кнопке OK. При выполнении отладочной команды для приложения в среде C++Builder с удаленным приложением будет установлено отладочное соединение. Теперь его отладку можно выполнять так же, как если бы оно находилось на локальном компьютере.

При получении сообщения об ошибке `Unable to connect to remote host` (Нельзя подключиться к удаленному компьютеру) следует проверить работоспособность сервиса или процесса сервера отладки, правильность имени удаленного компьютера в поле параметра Remote Host, а также наличие соединения с удаленным компьютером с помощью утилиты `ping.exe` или любой другой аналогичной сетевой утилиты. При получении сообщения об ошибке `Could not find program 'program'` (Не могу найти программу 'имя_программы') следует проверить правильность указанного пути в поле параметра Remote Path и наличие этого приложения в указанном месте.

Еще одной функциональной возможностью удаленной отладки является расширение способности присоединения к выполняемому процессу. Выберите команду меню Run⇨Attach To Process, введите имя удаленного компьютера в поле Remote Machine и нажмите клавишу <Enter>. После отображения на экране списка процессов, выполняемых на удаленном компьютере, выберите нужный процесс и щелкните на кнопке Attach для его отладки. Для присоединения к процессу на удаленном компьютере на нем должен быть запущен сервер отладки. Присоединяя выполняемый процесс, следует помнить о том, что не существует иного способа отсоединиться, кроме как прекратить его выполнение.

Отладка DLL-модулей

Отладка DLL-модуля аналогична отладке обычного выполняемого приложения, за исключением того, что при этом необходимо загрузить основное приложение этого DLL-модуля. Основное приложение может быть создано вами, но в большинстве случаев применяется существующее приложение, которое к тому же может быть на другом языке.

Загрузите проект DLL-модуля в среду C++Builder и установите нужные контрольные точки в его коде. Укажите основное приложение, для которого предназначен DLL-модуль, вводя путь и имя основного приложения в поле Host Application во вкладке Local диалогового окна Parameters, которое отображается на экране при выборе команды меню Run⇨Parameters. Введите в поле Parameters необходимые параметры командной строки приложения.

Если основное приложение указано, выберите либо команду Load для запуска основного приложения и начните отладку, либо просто щелкните на кнопке OK и запустите основное приложение с помощью команды меню Run⇨Run. Это можно сделать даже после установки дополнительных контрольных точек или указания просматриваемых выражений.

Вот и все. По достижении контрольной точки в коде DLL-модуля можно приступить к пошаговому выполнению кода и использовать инспектор отладки Debug Inspector, просматриваемые выражения или любой другой способ отладки. Этот метод можно применять для отладки COM-объектов и компонентов ActiveX, хотя отдельные процессы могут выполняться только в операционных системах Windows NT и Windows 2000, которые позволяют выполнять межпроцессовую отладку.

Другие инструменты отладки

Завершая описание процесса отладки, стоит упомянуть другие инструменты отладки.

Как уже говорилось в этой главе, ошибки часто возникают при внесении изменений в код. Во избежание таких неприятностей рекомендуется использовать систему контроля версий или ее разновидности, даже просто периодическое копирование содержимого папки проекта. В большинстве систем контроля версий имеется встроенная функция `diff`, предназначенная для сравнения двух версий исходного кода, которая может оказать существенную помощь при отслеживании новых ошибок. Существует также очень много других условно бесплатных версий утилиты `diff`. Некогда на дистрибутивных компакт-дисках некоторых версий Windows поставлялась также малоизвестная утилита `windiff.exe`. Кроем того, подобную утилиту Visual Diff можно бесплатно копировать с Web-узла компании Starbase по адресу `http://www.starbase.com`.

В состав корпоративной версии C++Builder 5 Enterprise входит клиентская часть системы параллельного контроля версий ***TeamSource***, причем ее можно купить отдельно для последующего использования в профессиональной версии C++Builder 5 Professional. Она позволяет сравнивать две версии одного и того же файла с кодом. Работа с программой ***TeamSource*** более подробно рассматривается в главе 29.

Еще один программный продукт CodeSite фирмы Raize Software, Inc. (`http://www.raize.com/`) представляет собой недорогой, но достаточно мощный инструмент отладки. Он позволяет пересылать данные из вашего приложения в среду централизованного просмотра, которая содержит инструменты для их анализа. При этом допускается пересылка всех типов данных, включая достаточно сложные объекты. Каждое выходное сообщение имеет отметку времени и может также быть использовано для хронометража основного кода. Утилита Overseer Debugger по функциональности аналогична программе CodeSite. Ее можно бесплатно скопировать по адресу `http://delphree.clexpert.com/pages/projects/nexus/overseer_debugger.htm`.

Программа GExperts имеет интерфейс отладки `dbugint` для регистрации отладочной информации в централизованном месте хранения для последующего ее просмотра с помощью инструмента `Gdebug.exe`. С ее помощью ваше приложение может регистрировать разные типы сообщений и получает дополнительные возможности контроля собранной отладочной информации. Программа GExperts бесплатно предлагается на сопроводительном компакт-диске Companion Tools вместе с профессиональной и корпоративной версиями C++Builder. Ее также можно скопировать по адресу `http://www.gexperts.org/`.

Программа ClassExplorer Pro фирмы ToolsFactory (`http://www.toolsfactory.com`) является дополнением сред C++Builder и Delphi и упрощает процесс создания и отладки приложений. Она существенно упрощает работу с классами и файлами, а также содержит контекстную строку, в которой указано имя класса и метода для той строки кода, в которой находится курсор. Она также предоставляет полноценный конструктор классов с возможностью генерации шаблонов кода, помогает создавать справочную документацию созданных вами классов и многое другое. Для некоммерческого использования ее можно использовать совершенно бесплатно.

Программа Sleuth QA Suite фирмы TurboPower Software Company (`http://www.turbopower.com`) содержит два программных продукта: программу Sleuth CodeWatch для проверки ресурсов и API-функций во время выполнения, которая аналогична инструменту CodeGuard, а также профайлер Sleuth StopWatch. Обе программы предназначены для использования вместе с C++Builder и Delphi. Программа CodeWatch обнаруживает и отфильтровывает известные причины утечки ресурсов для библиотеки VCL, что позволяет программисту сконцентрироваться на поиске собственных ошибок. Ее диаграммы с оперативным представлением информации помогают найти и устранить причины регулярной утечки памяти. Программа Sleuth StopWatch была кратко рассмотрена в главе 6.

Программа обнаружения утечки ресурсов MemProof содержится на сопроводительном компакт-диске Companion Tools для профессиональной (Professional) и корпоративной (Enterprise) версий C++Builder. Ее также можно скопировать по адресу `http://www.totalqa.com/`. Программа является условно бесплатной и интегрируется вместе со средой C++Builder. Она способна отслеживать выделение ресурсов BDE и Interbase, отображает вызовы API-функций, а также оперативные счетчики и содержит функции трассировки SQL-команд.

Кроме того, следует хотя бы кратко упомянуть инструменты отладки сторонних разработчиков.

- PR-Tracker фирмы Softwise Company является относительно недорогим решением составления отчетов о замеченных проблемах, ошибках и дефектах (`http://www.prtracker.com/`).
- SWBTracker фирмы Software With Brains также представляет собой относительно недорогую программу, которая обладает многими функциональными возможностями для поиска и регистрации ошибок (`http://www.softwarewithbrains.com/`).
- TestTrack фирмы Seapine Software Inc. (`http://www.seapine.com/`).

Программа Doc-o-Matic фирмы ToolsFactory является системой документирования, которая генерирует HTML-код и файлы справки на основе комментариев в исходном коде вашего приложения (`http://www.doc-o-matic.com`).

Компонент WinSight поставляется вместе с профессиональной (Professional) и корпоративной (Enterprise) версиями C++Builder. Он позволяет собирать отладочную информацию о классах, окнах и сообщениях во время выполнения приложения. Его можно использовать для просмотра способов создания классов и окон в вашем приложении, а также отслеживать сообщения окон.

Обозреватель объектов OLE/COM Object Viewer (`OleView.exe`) является бесплатным приложением к пакету Microsoft Platform SDK, который можно бесплатно скопировать с Web-узла фирмы Microsoft.

В области отладки приложений существует бесчисленное множество инструментов, которые могут оказать помощь в отладке приложения. Дайте мне знать, если найдете еще что-то действительно ценное и полезное!

Тестирование

Тестирование является естественным продолжением процесса отладки и заключается в наблюдении за ошибками и иногда обнаружении ошибок. Эти две смежные грани процесса создания программного обеспечения пересекаются в области обнаружения ошибок, когда достаточно информативные результаты тестирования помогают найти ошибку.

Тестирование —совершенно особая часть процесса создания программного обеспечения. Без него вряд ли удастся создать корректно работающее приложение. Уровень тестирования зависит от типа создаваемого приложения, требований к его устойчивости и безопасности, а также ресурсов, доступных при его создании (если вы ограничены во времени, следует ограничить функциональность приложения в пользу его качества).

Тестирование — обширная и сложная тема, и ей посвящено достаточно много книг. Здесь же мы ограничимся описанием только самых общих принципов тестирования.

Этапы и методы тестирования

Отдельные этапы тестирования выполняются непосредственно при создании приложения. Некоторые из них перечислены ниже. Каждый этап обычно разбивается на несколько меньших повторяющихся этапов. Эти тестовые этапы обычно выполняются в таком порядке.

- **Тестирование модуля**. *Модуль* (*unit*) — небольшая группа связанных функций, один или несколько объектов. Тестирование и отладка модуля часто выполняется совместно при создании приложения.
- **Рецензирование кода**. На этом этапе код одного программиста проверяет другой программист, обычно просто просматривая и обсуждая его с автором кода.
- **Тестирование компонента**. *Компонент* (*component*) — группа модулей или объектов, или даже целая программа, которая выполняет особый набор функций. Он тестируется целиком.
- **Интегрированное тестирование**. Один или несколько компонентов соединяются в единую функциональную систему и тестируется взаимодействие этих компонентов с остальной системой.
- **Системное тестирование**. На этом этапе тестируется система в целом, внимание при этом обычно концентрируется на производительности или надежности.
- **Приемочное тестирование**. Это обычно некий ориентир, который обозначает, что система успешно прошла тесты компонентов и их интеграции и готова к работе.
- **Полевые испытания**. Обычно полевые испытания проводятся путем рассылки альфа- и бета-версий системы пользователям, которые впоследствии сообщают о найденных ошибках.
- **Регрессивное тестирование**. Это повторение предыдущих этапов тестирования после интеграции изменений в работающую систему для проверки корректности работы. Такое тестирование обычно выполняются после обнаружения ошибки или добавления нового компонента.

Помимо такого тестирования, существует много других способов. Среди них следует назвать тестирование по методу *белого ящика* (*whitebox testing*), называемое также тестированием по методу *стеклянного ящика* (*glassbox testing*), или *структурное тестирование*; а также тестирование по методу *черного ящика* (*blackbox testing*), или *функциональное тестирование*.

При тестировании по методу белого ящика полностью обнажается внутренняя структура тестируемой части системы. Большое количество тестов используется для проверки корректности пути выполнения через каждую строку кода. При этом для каждой ветви пути выполнения необходимо применить отдельный тест. В некоторых случаях приходится вносить изменения в код для поддержки полноценного тестирования по методу белого ящика. Для малых приложений или отдельных частей больших приложений для этого можно применить интерактивный отладчик.

Тестирование по методу белого ящика обычно выполняется на этапах тестирования модулей и компонентов.

При тестировании по методу белого ящика следует использовать вывод отладочной информации, контрольной точки или профайлер, который может сообщить о том, какая часть кода была протестирована, а какая — нет. Сформулируйте такие тесты, которые могли бы гарантированно проверить каждый участок кода.

При тестировании по второму методу тестируемая часть системы рассматривается как “черный ящик”. Это значит, что в целях тестирования внутренняя структура системы (и принципы ее работы) полностью скрыта, а единственной видимой частью системы является ее интерфейс. При этом в систему вводятся какие-либо данные и анализируется правильность выходных данных, а происходящие внутри процессы не рассматриваются. Этот способ обычно применяется на этапах тестирования компонентов, интегрированного и системного тестирования.

При тестировании по методу черного ящика необходимо проверить широкий диапазон вводимых данных, включая обычные правильные входные данные, неожиданные правильные входные данные, а также разные неправильные входные данные. Тщательно должны быть проверены все граничные условия и промежуточные данные.

Проблема тестирования по методу черного ящика заключается в том, что нельзя с уверенностью сказать, какая часть кода была протестирована, а какая — нет. Поэтому одного такого тестирования недостаточно. Наилучший результат обычно можно получить на основании комбинации нескольких методов тестирования.

Оба эти способа тестирования очень подробно описываются в учебниках по тестированию программного обеспечения.

Советы, касающиеся тестирования

Всегда тестируйте созданный вами код! В прошлом автору доводилось много раз работать со своим собственным и чужим кодом, не подвергнутым тестированию, и всякий раз это напоминало о себе впоследствии. Лучше в итоге получить более ограниченный по возможностям, но надежный программный продукт, чем большой набор ошибочно работающих компонентов. Тестируйте код, даже если это приведет к срыву ранее установленных сроков выпуска программного продукта, проинформировав об этом руководителя группы или менеджера проекта.

Тестировать приложение следует еще при создании, а не только в готовом виде. Лучше всего тестировать приложение, когда его код еще свеж в памяти разработчика. По возможности процесс тестирования следует автоматизировать.

При создании заменяемого приложения прежде всего выполните преобразование данных. Это не только станет гарантией лучшего понимания существующей системы, но и позволит более точно определить сроки создания приложения. Некоторое время при этом потребуется затратить для создания инструментов преобразования данных, особенно если опущены подробности существующих данных или самой системы.

Не забудьте протестировать сами тесты! Это особенно важно для инструментов встроенного и автоматического тестирования. Если исходные тесты неправильно или в недостаточной степени тестируют приложение, то они могут обмануть ваши ожидания.

Ниже перечислены полезные рекомендации с подробным описанием методов тестирования.

- Прекрасной книгой с описанием всех аспектов тестирования программного обеспечения для программистов, тестировщиков и менеджеров проекта является книга *Testing Computer Software*, Kaner, Nguyen, Falk. Издательство: John Wiley & Sons. ISBN 0-471-35846-0, 1999.
- Журнал *Dr. Dobb's Journal* (`http://www.ddj.com/`) время от времени публикует статьи о тестировании, которые можно приобрести как по отдельности, так и на компакт-диске в составе подшивки за последние 11 лет по цене около 80 долларов США. В частности, мартовский выпуск 2000 года содержит несколько статей, посвященных вопросам тестирования и отладки.

Наконец, следует заметить, что тесты дают только некие дополнительные сведения о работе приложения. Прежде всего нужно четко представлять себе принципы работы приложения, а затем убедиться в том, что оно работает соотвествующим образом.

Резюме

Отладка представляет собой очень сложный процесс поиска и исправления ошибок. Для полного описания всех тонкостей этого процесса может понадобиться целая книга. Автор надеется, что предлагаемая сокращенная версия может пролить свет на эту часть процесса создания приложения.

В C++Builder 5 также усовершенствованы некоторые компоненты отладки, которые не рассмотрены в этой главе. Полный список этих компонентов с подробным описанием всегда можно найти в оперативной справке C++Builder 5.

Прекрасным источником ценной информации о методах поиска и исправления ошибок являются списки рассылки новостей Borland. Они перечислены в соответствующем разделе приложения А. Здесь также следует упомянуть прекрасную книгу *Writing Solid Code*, Steve Maguire, опубликованную в издательстве Microsoft Press. ISBN 1-556-15551-4, 1993.

По мере приобретения опыта работы в создании программного обеспечения старайтесь овладевать новыми методами отладки. Чтобы сэкономить время, потраченное на создание, поддержку и сопровождение приложения, нужно сделать все возможное для предотвращения возможных ошибок.

Глава

8

Компоненты библиотеки VCL

Малькольм Смит
Крис Винтерс
Дэймон Чандлер
Джарод Холингвэрт
Халид Алманай

C++Builder представляет собой среду быстрого создания приложений или RAD-среду (Rapid Application Development — RAD) с библиотекой визуальных компонентов VCL (Visual Component Library — VCL), которая состоит из готовых к употреблению визуальных и невизуальных объектов и оболочек. Она позволяет с минимальными затратами создавать приложения, в то же время предоставляя определенную степень независимости от библиотеки VCL.

Эта глава содержит сведения о том, для чего в основном предназначена C++Builder, — объектно-ориентированной разработке приложений. При работе с компонентами в C++Builder широко используется понятие повторного использования объектов. Компоненты являются экземплярами классов, которые доступны с помощью палитры компонентов Component Palette. Что может быть проще при создании приложений, чем просто опустить нужный компонент в форму, задать его свойства, создав затем один или несколько обработчиков событий.

Мы рассмотрим здесь классы библиотеки VCL, которые являются наследниками класса `TObject`, а также способы работы инспектора объектов ***Object Inspector*** для получения информации о типах в процессе выполнения или RTTI-информации (Runtime Type Information — RTTI). Она необходима разработчику для предоставления в оперативном режиме средств инспекции и редактирования общих свойств и событий, которые совместно используются несколькими компонентами. Обычные классы C++, классы библиотеки VCL и расширения языка сравниваются здесь для демонстрации многих преимуществ, которые получают разработчики, работая с C++Builder.

Мы рассмотрим стандартные и общие элементы управления, а также их взаимосвязь с динамической библиотекой `COMCTL32.DLL` Microsoft Windows, которую необходимо учитывать разработчику. Затем будут кратко рассмотрены обновления, которые предусмотрены в C++Builder.

После этого мы приступим к изучению наиболее важных расширений библиотеки VCL в C++Builder 5: новых справочных подсказок Help Hints, команд меню и возможностей доступа к ключам реестра, компонентов `TApplicationEvents` и `TIcon`.

Далее приводится описание еще одного расширения библиотеки VCL — класса `TStringList`. В этом же разделе описаны способы использования мощного контейнера-списка строк, а также стандартные структуры и классы хранения. После этого обсуждаются новые пользовательские события рисования на основе классов `TTreeView`, `TListView` и `TToolBar`, а также способы создания приложений панели управления Control Panel с помощью C++Builder. Наконец, мы рассмотрим, как компоненты сторонних разработчиков позволяют ускорить создание приложения и заложить основу для дальнейшего усовершенствования достаточно сложных оболочек для API-функций. Эти компоненты создаются на основе библиотеки VCL C++Builder и модели, которая основана на работе со свойствами, методами и событиями, или PME-модели (properties, methods, events — PME).

Обзор библиотеки VCL

Вместе с C++Builder поставляется диаграмма, в которой схематически представлена иерархическая структура объектов библиотеки VCL. Не удивительно, что эта диаграмма расширена в новой версии C++Builder. Сложность структуры библиотеки VCL объясняется тем, что она представляет собой нечто гораздо большее, чем просто совокупность объектов, унаследованных от других объектов.

Библиотека VCL основана на PME-модели, т.е. на работе со свойствами, методами и событиями (Properties, Methods, Events — PME). Ее архитектура связана с палитрой компонентов ***Component Palette***, инспектором объектов ***Object Inspector*** и IDE-средой, которая предоставляет разработчикам способ быстрого создания приложений или RAD-способ (Rapid Application Development — RAD). Программисту достаточно перетащить нужные компоненты в форму и в результате этого он получит приложение Windows, почти не создавая новых строк кода.

Очевидно, для придания приложению большей функциональности какой-то код все-таки придется создать, но большую часть работы все же выполнит библиотека VCL, что позволит повысить эффективность и упростить процесс создания приложения. При этом большую часть времени программист теперь сможет потратить на создание рабочих блоков приложения, а не на создание стандартной оболочки, которой должно обладать приложение Windows.

Ниже кратко описана общая иерархия объектов библиотеки VCL.

Все начинается с класса TObject

Библиотека VCL представляет собой группу объектов, которые происходят от абстрактного класса `TObject`. Этот класс обладает способностью откликаться на процессы создания и удаления объектов, поддерживает управление сообщениями, а также содержит сведения о типе класса и RTTI-информацию его открытых свойств.

RTTI-информация позволяет определить тип объекта во время выполнения, даже если в коде используется только указатель или ссылка на этот объект. Например, в C++Builder указатель `TObject` передается каждому его событию (это может быть щелчок мыши или получение объектом фокуса). Используя RTTI-информацию, можно выполнить динамическое приведение типа (с помощью оператора `dynamic_cast`, который будет рассмотрен ниже) для использования объекта либо для определения типа объекта. RTTI-механизм также позволяет проверить тип объекта с помощью оператора `typeid`. Дополнительную информацию по этой теме можно найти в оперативном справочном руководстве *C++Builder Language Guide*.

Объект `TObject` наследуют многие простые классы с неусточийвыми данными, оболочки и потоки, например `TList`, `TStack`, `TPrinter`, `TRegistry` и `TStream`.

Термин *устойчивые данные* (*persistent data*) в библиотеке VCL означает механизм хранения значений свойств. Простейший пример —надпись на кнопке или текстовой надписи. Во время создания приложения текст надписи вводится с помощью окна инспектора объектов Object Inspector. Она сохраняется для использования в других сеансах программирования и доступна при выполнении приложения. Вот такие данные и называются устойчивыми. *Классами с неустойчивыми данными* (*non-persistent data classes*) называются простые классы, которые предназначены для выполнения частных функций, но не сохраняют и не передают информацию о своем состоянии в другие сеансы программирования.

Оболочка (*wrapper*) представляет собой способ обрамления наиболее сложных функций Windows API. Оболочки предназначены для создания и применения простых и удобных в использовании объектов или компонентов. Такие компоненты или объекты скрывают от разработчика структуру API-функций и в то же время позволяют ему использовать всю мощь этих функций. В следующих главах предлагается дополнительная информация по этому поводу.

Другим часто используемым наследником `TObject` является класс `Exception`, который предоставляет много встроенных классов, предназначенных для обработки исключительных ситуаций (например, при делении на ноль или возникновении ошибок при работе с потоками). Этот класс также может быть использован для создания пользовательских классов в ваших приложениях с минимальными затратами.

Другими основными наследниками класса `TObject` являются `TPersistent`, `TComponent`, `TControl`, `TGraphicControl` и `TWinControl`.

Класс `TPersistent` дополняет класс `TObject` методами, которые позволяют объекту сохранять свое состояние вплоть до удаления и повторно загружать это состояние при его повторном создании. Этот класс имеет очень большое значение при создании компонентов, которые содержат пользовательские классы в качестве свойств. Если свойство необходимо передать потоку, его нужно сделать наследником класса `TPersistent`, а не класса `TObject`. Эта

ветвь наследования содержит многие типы классов, из которых наиболее распространенными являются классы TCanvas, TClipboard, TGraphic, TGraphicsObject и TStrings. Класс TComponent, также наследник класса TPersistent, является общим базовым классом всех компонентов библиотеки VCL.

Передача свойств потоком (*property streaming*) означает механизм, благодаря которому значения свойств объекта записываются в файл формы. При повторном открытии формы значения свойств передаются потоком (или считываются) обратно с восстановлением их прежних значений.

Объекты TComponent образуют основу приложений C++Builder. Они могут размещаться в палитре компонентов Component Palette, могут быть родителями других компонентов, управлять другими компонентами, а также выполнять операции управления потоками.

Компоненты делятся на визуальные и невизуальные. Невизуальные компоненты не требуют никакого видимого интерфейса, а потому являются непосредственными наследниками класса TComponent. Визуальные компоненты должны быть визуализированы и взаимодействовать с пользователями при выполнении приложения. Класс TControl дополняет его процедурами рисования и событиями окон, которые необходимы для определения визуального компонента. Эти визуальные компоненты делятся на две группы: оконные (TWinControl) и неоконные (TGraphicControl).

Компоненты TGraphicControl полностью отвечают за рисование своего внешнего вида, но никогда не могут получить фокус ввода. Примерами таких компонентов являются классы изображения TImage, надписи TLabel, фаски TBevel и формы TShape.

Компоненты TWinControl аналогичны TGraphicControl, за исключением того, что могут получать фокус ввода и взаимодействовать с пользователями. Эти компоненты называются *оконными элементами управления* (*windowed controls*). Они имеют дескриптор окна и могут содержать другие элементы управления или быть их родителями.

Более подробная информация о визуальных и невизуальных компонентах приводится в главе 9 о создании пользовательских компонентов.

Программирование на основе существующих объектов

Объектно-ориентированная архитектура C++Builder позволяет быстрее создавать приложения на основе повторного использования уже существующих объектов и классов. Способность объектов компонента предоставлять открытые свойства разработчику с помощью инспектора объектов ***Object Inspector***, образно говоря, открывает разработчику выход в новое измерение, что позволяет оптимизировать процесс создания приложения.

Инспектор объектов Object Inspector позволяет не только просматривать и редактировать открытые свойства компонента. На рис. 8.1 показано, что в инспекторе объектов Object Inspector можно работать с общими свойствами разных элементов управления TLabel, TEdit, TButton и TCheckBox. Просматривая иерархию этих элементов управления, можно заметить, что объект TLabel является наследником класса TGraphicControl, а остальные — наследниками класса TWinControl. Общим предком всех четырех элементов управления является класс TControl (так как TGraphicControl и TWinControl являются наследниками класса TControl).

Следовательно, на рис. 8.1 показаны общие свойства компонентов, которые являются наследниками класса TControl. При изменении этих свойств для выбранных элементов управления это изменение одновременно отражается во всех элементах управления.

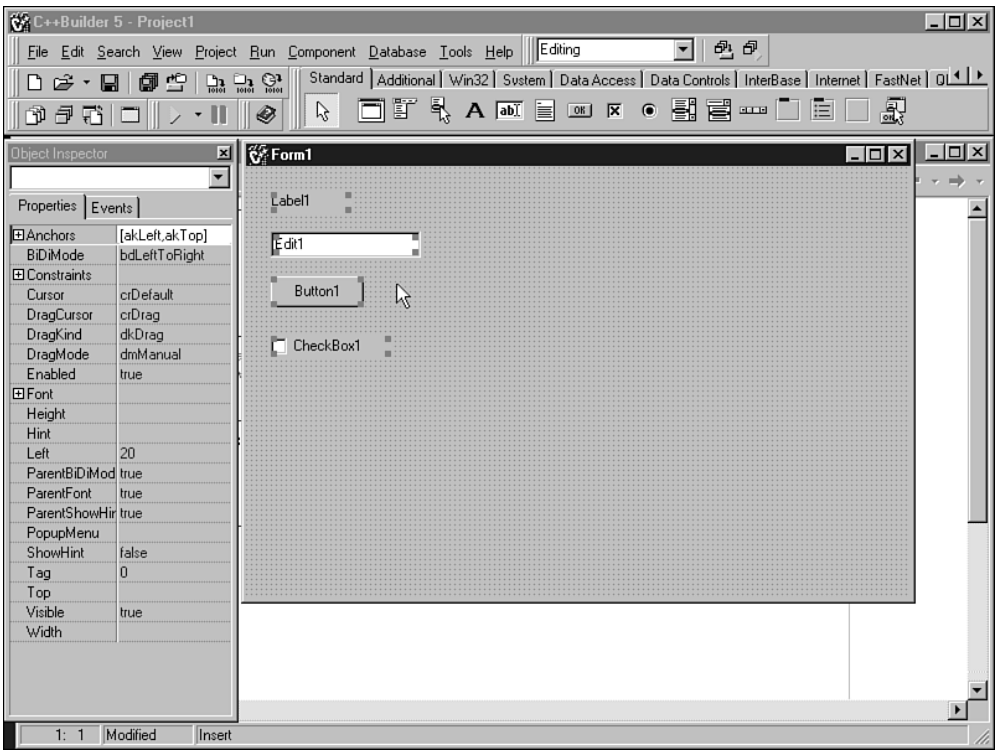


Рис. 8.1. Общие свойства нескольких выбранных компонентов

На рис. 8.2 показан пример использования окна инспектора объектов Object Inspector, в котором демонстрируется способность обнаружения компонентов, которые являются наследниками класса редактируемого свойства. Свойство `LinkedEdit` имеет в примере тип `TCustomMemo`, а в форме имеются элементы управления типа `TRichEdit` и `TMemo`, которые являются наследниками типа `TCustomMemo`.

Создание объектов на основе существующих классов или объектов позволяет дополнять объекты-наследники новыми функциями и в то же время использовать базовые классы для новых наследников, для которых нужно применить базовую функциональность вместе с другими дополнительными и уникальными чертами. Некоторые разработчики могут возразить, что этот дополнительный класс и связанная с ним RTTI-информация приводят только к возрастанию накладных расходов, связанных с разработкой приложения. Но эти накладные расходы оправдываются преимуществами, которые обеспечивает эта объектная модель. Объекты и компоненты C++Builder могут выглядеть по-разному, но обладать общими чертами, унаследованными от объектов-предков, что существенно упрощает процесс создания приложения.

Применение библиотеки VCL

Важно понимать разницу между библиотекой VCL и обычными классами и объектами. Библиотека VCL происходит от библиотеки Object Pascal. Поэтому все объекты библиотеки VCL создаются в куче и на них ссылаются с помощью указателей, а не как на статические объекты. Так создаются и используются только объекты библиотеки VCL, а все стандартные объекты C/C++ могут использоваться и по-другому.

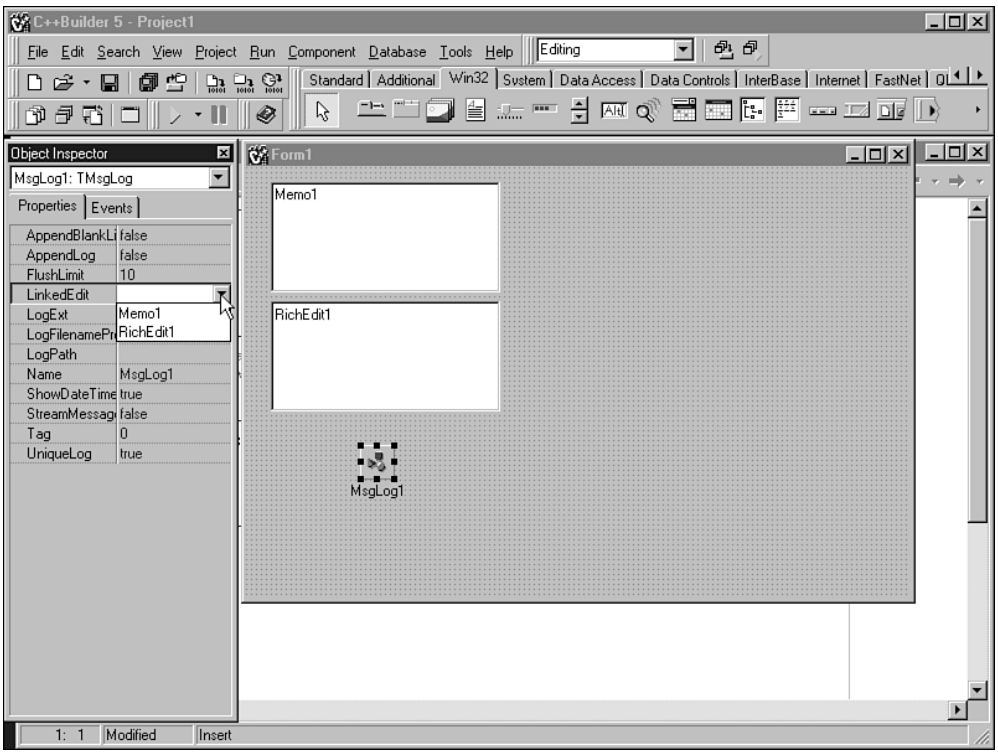


Рис. 8.2. Обнаружение наследников типа редактируемого свойства

Рассмотрим следующий пример. В приведенном ниже фрагменте кода показано, как стандартный класс C++ сначала создается в стеке, а потом — в куче. Затем приводится пример создания объекта библиотеки VCL.

Вот фрагмент кода для класса C++.

```
class MyClass
{
private:
   int MyVar;
public:
   MyClass(void);
};
```

Далее показан пример создания экземпляра этого класса в стеке. При выходе класса за пределы видимости выделенная для этого объекта память будет освобождена автоматически.

```
MyClass Tmp;
Tmp.MyVar = 10;
// Какие-то другие действия.
```

Теперь рассмотрим создание экземпляра этого класса в куче. В этом случае создатель объекта отвечает за удаление объекта до выхода за пределы видимости. В противном случае память не будет освобождена, что приведет к утечке памяти.

```
MyClass *Tmp;
Tmp = new MyClass;
```

```
Tmp->MyVar = 10;
// Какие-то другие действия.
delete Tmp;
```

Как показано во втором примере, объекты библиотеки VCL создаются в куче. При попытке создать объект библиотеки VCL в стеке компилятор сообщит, что классы библиотеки VCL должны конструироваться с помощью оператора `new`.

В C++Builder также предусмотрено автоматическое удаление объектов, которые имеют владельцев. Предположим, что объект `TLabel` создан динамически с передачей указателя `this` на его владельца.

```
TLabel *MyLabel = new TLabel(this);
```

При выходе объекта `MyLabel` за границы области видимости можно предположить, что при этом произойдет утечка памяти. Однако в данном случае не следует беспокоиться о явном освобождении памяти для объектов такого типа, потому что библиотека VCL имеет встроенный механизм освобождения всех дочерних объектов до удаления их объекта-владельца. Почему при этом предпринимается попытка избежать упоминания термина *родительский объект* (*parent object*), будет объяснено ниже.

Невизуальные компоненты имеют владельцев, а визуальные — и владельцев, и родителей. Простейший способ отличить их основан на сравнении владельца с создателем, а родителя — с контейнером, который допускает существование этого компонента. Рассмотрим компонент `TPanel`, который находится в форме и содержит три компонента-надписи. В зависимости от способа создания надписей возможны следующие сценарии.

В первом сценарии надписи могут быть перетащены в панель во время создания приложения. В этом случае родителем всех надписей является панель, а владельцем — само приложение. После закрытия приложения библиотека VCL гарантирует, что каждый дочерний объект (панель и три надписи) будут удалены до удаления всего приложения.

Во втором сценарии в форму может быть перетащен сложный компонент, состоящий из нескольких компонентов. В этом примере предполагается, что сложным компонентом является панель с тремя надписями. Этот компонент создает панель и три надписи на ней. Панель остается родителем этих надписей, но их владельцем и владельцем панели является тот компонент, который был перетащен в форму. При удалении этого компонента автоматически удаляется эта панель и надписи.

Это очень удобно при создании приложения, но если объекты все же создаются во время выполнения приложения, то их лучше удалить явным образом. Если вы не сделаете этого, утечка памяти не возникнет, но это все же повышает читабельность кода и позволяет ясно документировать назначение кода. Ниже приводится фрагмент кода, который демонстрирует создание этих объектов при выполнении приложения.

```
TPanel *Panel1;
TLabel *Label1, *Label2, *Label3;
Panel1 = new TPanel(this);
Panel1->Parent = Form1;
Label1 = new TLabel(this);
Label1->Parent = Panel1;
Label2 = new TLabel(this);
Label2->Parent = Panel1;
Label3 = new TLabel(this);
Label3->Parent = Panel1;
// Установка значений других свойств: надписей, позиций, и т.д.
// Другие действия.
delete Label1;
```

```
delete Label2;
delete Label3;
delete Panel1;
```

В этом примере создания панели ее владельцем является приложение, а родителем — форма. При этом надписи создаются аналогичным образом, с той лишь разницей, что панель является родителем надписей. Надписи могут быть перемещены в другую панель только с помощью редактирования свойства `Parent`. При удалении панели все ее надписи будут удалены автоматически, но автор предпочитает удалять их явным образом. Это приобретает особо большое значение при работе с реальными приложениями, в которых совместно используются глобальные указатели. Кроме того, для этих указателей рекомендуется задавать значение `NULL` (ноль) после удаления (если только они не будут находиться вне области видимости). Дело в том, что, если другая часть приложения попытается удалить уже удаленный объект, а его указателю не было присвоено значение `NULL`, то это может привести к нарушению доступа.

Расширения языка C++

Среда C++Builder содержит расширения языка C++, которые превращают ее в достаточно мощный продукт, способный использовать библиотеку VCL и прозрачно интегрироваться с PME-моделью. Как программиста автора совсем не беспокоят спорные вопросы, связанные с применением расширений языка C++, до тех пор, пока они способствуют созданию мощного приложения, совместимы со стандартами ANSI и ускоряют процесс создания приложения. Со всеми этими обязанностями C++Builder справляется очень хорошо.

Ниже приводятся эти расширения вместе с краткой характеристикой. Обратите внимание, что каждое расширение содержит префикс в виде двух символов подчеркивания.

Расширение `__automated`

OLE-технология связывания и внедрения объектов (object linking and embedding — OLE) не предоставляет информацию о типах в библиотеке типов. Раздел автоматизации класса, производного от `TAutoObject` или наследника `TAutoObject`, используется приложениями, в которых применяется OLE-автоматизация. Необходимая OLE-автоматизация генерируется для членов-функций и свойств, объявленных в этом разделе.

```
class TMyAutoClass :public TAutoObject
{
public:
   virtual __fastcall TMyAutoClass(void);

__automated:
   AnsiString __fastcall GetClassName(void)
                          { return("MyAutoClass"); }
};
```

В заключение заметим, что объявленный метод имеет тип `__fastcall` (который подробнее будет описан ниже). Это объявление необходимо для функций автоматизации и оно указывает компилятору на попытку передачи параметров в регистрах, а не в стеке.

Расширение `__classid(class)`

Расширение `__classid` используется для организации связи между RTTI-функциями библиотеки VCL и языком C++. Оно используется в тех случаях, когда RTTI-функции необходимо получить информацию о классе в качестве параметра.

Например, метод RegisterComponentEditor() позволяет создавать и регистрировать пользовательский редактор для созданных вами компонентов. Так, в моей компании специально для использования в среде C++Builder применяется пакет компонентов обеспечения безопасности MJFSecurity. Этот пакет содержит несколько пользовательских компонентов, предназначенных для защиты приложений от пиратского распространения. Один из них называется TAppLock. Он использует пользовательский редактор TAppLockEditor. Частью метода Register() этого пакета является редактор, который регистрируется с помощью представленного ниже кода.

```
namespace Applock
{
   void __fastcall PACKAGE Register()
   {
      // ... здесь пропущен код
      RegisterComponentEditor(__classid(TAppLock),
                              __classid(TAppLockEditor));
   }
}
```

Метод RegisterComponentEditor() принимает в качестве параметров два указателя на объекты TMetaClass (способ представления в C++Builder ссылочного типа на класс из Object Pascal). Объект TMetaClass для класса можно получить с помощью оператора __classid. Компилятор использует оператор __classid для генерации указателя на vtable (виртуальную таблицу) для имени класса.

Более подробную информацию о компоненте MJFSecurity можно получить по адресу http://www.mjfreelancing.com/. Кроме того, в главе 28, посвященной способам распространения программного обеспечения, рассматривается компонент AppLock и обсуждаются другие вопросы обеспечения безопасности.

Расширение __closure

В стандарте C++ допускается присвоение экземпляра производного класса указателю базового класса, но не допускается присвоение члена-функции производного класса указателю члена-функции базового класса. В листинге 8.1 приведен пример такого присвоения.

Листинг 8.1. Недопустимое присвоение члена-функции производного класса

```
enum HandTypes {htHour, htMinute, htSecond};

class TWatch
{
public:
   void MoveHand(HandTypes HandType);
};

class TWatchBrand :public TWatch
{
public:
   void NewFeature(bool Start);
};

void (TWatch::*Wptr)(bool);      // объявление указателя на
```

```
                                  // член-функцию базового класса
Wptr = &TWatchBrand::NewFeature; // недопустимое присвоение
                                  // члена-функции производного
                                  // класса
```

Для разрешения этой проблемы в C++Builder предусмотрено расширение `__closure`. В листинге 8.2 демонстрируется способ употребления классов из листинга 8.1 вместе с этим расширением.

Листинг 8.2. Присвоение члена-функции производного класса указателю члена-функции базового класса с помощью расширения `__closure`

```
TWatchBrand *WObj = new TWatchBrand; // создание объекта
void (__closure *Wptr)(bool); // определение указателя замыкания
Wptr = WObj->NewFeature;      // указание его
                              // для члена-функции NewFeature
Wptr(false); // вызов функции, передача значения false
Wptr = 0;    // задание для указателя значения NULL
delete WObj; // удаление объекта
```

Обратите внимание, что указатели замыкания можно также присваивать членам-функциям базового класса, как показано в листинге 8.3.

Листинг 8.3. Присвоение указателей замыкания членам-функциям базового класса

```
void (__closure *Wptr2)(HandTypes);
Wptr2 = WObj->MoveHand;
Wptr2(htSecond);
```

Ключевое слово `__closure` преимущественно используется для обработки событий в среде C++Builder.

Расширение `__declspec`

На классы библиотеки VCL налагаются следующие ограничения.

- Не допускается создание базовых виртуальных классов.
- Не допускается множественное наследование.
- Память для них должна выделяться динамически в куче с помощью глобального оператора `new`.
- Они должны иметь деструктор.
- Конструкторы копии и операторы присваивания не генерируются компилятором для классов, производных от библиотеки VCL.

Для преодоления упомянутых выше ограничений в библиотеке VCL предусмотрено ключевое слово `__declspec`. Для применения этого ключевого слова в файле `sysmac.h` предусмотрены соответствующие макросы.

`__declspec(delphiclass, package)`

В файле `sysmac.h` макрос `DELPHICLASS` определен как `__declspec(delphiclass, package)`. Аргумент `delphiclass` используется для объявления классов, производных от класса `TObject`.

Если класс транслируется из библиотеки Object Pascal, то компилятор должен знать, что этот класс является наследником класса TObject. Поэтому здесь используется этот модификатор.

Аналогично, при создании нового класса, который будет использоваться как свойство компонента, и при необходимости предварительного объявления класса, необходимо использовать ключевое слово __declspec(delphiclass, package), чтобы сообщить обо всем этом компилятору. В листинге 8.4 приводится пример прямого объявления класса, когда новый компонент имеет тип указателя на этот класс.

Листинг 8.4. Предварительное объявление класса для применения его как члена другого класса

```
class TMyObject;

class TMyClass : public TComponent
{
private:
   TMyObject *FMyNewObject;

public:
   __fastcall TMyClass(TComponent *Owner): TComponent(Owner){}
   __fastcall ~TMyClass(void);
};

class TMyObject : public TPersistent
{
public:
   __fastcall TMyObject(void){}
   __fastcall ~TMyObject(void);
};
```

Код из листинга 8.4 будет компилироваться без проблем, потому что компилятор использует только класс TMyObject. При создании свойства типа TMyObject необходимо сообщить компилятору, что этот класс является наследником класса TObject. Измененный вариант этого кода представлен в листинге 8.5.

Листинг 8.5. Применение макроса DELPHICLASS для прямого объявления класса, который использует свойство другого класса

```
class DELPHICLASS TMyObject;

class TMyObject;

class TMyClass : public TComponent
{
private:
   TMyObject *FMyNewObject;
public:
   __fastcall TMyClass(TComponent *Owner) : TComponent(Owner){}
   __fastcall ~TMyClass(void);
   __published:
   __property TMyObject *NewObject = {read = FMyNewObject};
};
```

```
class TMyObject : public TPersistent
{
public:
   __fastcall TMyObject(void){}
   __fastcall ~TMyObject(void);
};
```

В файле sysmac.h также определен близко связанный макрос RTL_DELPHICLASS в виде __declspec(delphiclass). Этот макрос используется в тех случаях, когда в классе необходимо применить функции незапакованной библиотеки времени выполнения или RTL-библиотеки (runtime library — RTL). Макрос DELPHICLASS используется для компонентов в пакетах только для выполнения или пакетов для выполнения/создания. Более подробную информацию о пакетах только для выполнения и пакетах только для создания можно найти в главе 2.

__declspec(delphireturn, package)

В файле sysmac.h макрос DELPHIRETURN определен как __declspec(delphireturn, package).

Параметр delphireturn используется в среде C++Builder для поддержки встроенных типов данных и конструкций языка Object Pascal. В качестве примеров следует упомянуть типы Currency, AnsiString, Variant, TDateTime и Set.

Аналогично расширению __declspec(delphiclass) для расширения __declspec (delphireturn) также определен макрос, который называется RTL_DELPHIRETURN. Он используется в тех случаях, когда в незапакованных классах необходимо использовать семантику Delphi.

__declspec(dynamic)

В файле sysmac.h макрос DYNAMIC определен как __declspec(dynamic). Аргумент dynamic применяется для динамических функций и только для классов, производных от класса TObject. Они аналогичны виртуальным функциям, за исключением того, что информация из таблицы vtable хранится только в объектах, которые создали эти функции. При вызове несуществующей динамической функции родительские таблицы vtable просматриваются до тех пор, пока не будет найдена нужная функция. Динамические функции эффективно сокращают размер таблиц vtable за счет короткой задержки, необходимой для подстановки родительских таблиц.

Динамические функции не могут быть объявлены как виртуальные, и наоборот.

__declspec(hidesbase)

В файле sysmac.h макрос HIDESBASE определен как __declspec(hidesbase). Аргумент hidesbase используется для сохранения семантики Object Pascal. Рассмотрим класс C1 и его наследника — класс C2. Если оба класса содержат функцию foo, то C++ интерпретирует C2::foo() как замену функции C1::foo(). В Object Pascal C2::foo() является совершенно другой функцией, отличной от функции C1::foo(). Для поддержания этой семантики при работе с классами C++ используется макрос HIDESBASE. В листинге 8.6 показан пример использования этого макроса.

Листинг 8.6. Применение макроса HIDESBASE для переопределения методов класса в классах-наследниках

```
class TMyObject : public TPersistent
{
public:
   __fastcall TMyObject(void){}
   __fastcall ~TMyObject(void);
```

```
   void __fastcall Func(void){}
};

class TMyObject2 : public TMyObject
{
public:
   __fastcall TMyObject2(void){}
   __fastcall ~TMyObject2(void);
   HIDESBASE void __fastcall Func(void){}
};
```

__declspec(hidesbase, dynamic)

В файле sysmac.h макрос HIDESBASEDYNAMIC определен как __declspec(hidesbase, dynamic). Он используется для поддержания семантики Object Pascal для динамических функций. Макрос HIDESBASE используется для поддержания принятого в Pascal способа переопределения методов в классах-наследниках. Макрос HIDESBASEDYNAMIC выполняет ту же функцию для динамических методов.

__declspec(package)

В файле sysmac.h макрос PACKAGE определен как __declspec(package). Аргумент package означает, что этот код определяет класс, который может компилироваться в виде пакета. Это позволяет импортировать и экспортировать классы из полученного в результате компиляции BPL-файла.

__declspec(pascalimplementation)

В файле sysmac.h макрос PASCALIMPLEMENTATION определен как __declspec(pascalimplementation). Аргумент pascalimplementation означает, что код этого класса создан с помощью Object Pascal. Этот модификатор представлен в заголовочном файле переносимости Object Pascal с расширением .hpp.

Ключевое слово __fastcall

Ключевое слово __fastcall используется для объявления функций, параметры которых предполагается передавать через регистры. Если параметр имеет тип числа с плавающей запятой или тип struct, то регистры не используются.

Вообще, ключевое слово __fastcall используется только для объявления членов-функций библиотеки VCL. Использование этого модификатора в других случаях скорее всего приведет к снижению производительности. Типичные примеры его использования можно найти во всех классах форм и членах-функциях компонентов. Следует отметить, что модификатор __fastcall может иметь скорректированное имя.

Ключевое слово __property

Ключевое слово __property используется для объявления свойств в классах (даже для классов не из библиотеки VCL), т.е. членов-данных со следующими особенностями.

- Они имеют связанные с ними методы чтения и записи.
- Они могут иметь значения по умолчанию.
- Они могут быть направлены в виде потока в (или из) файла формы.

- Они могут расширять свойство, определенное в базовом классе.
- Они могут использоваться только для чтения или только для записи.

Более подробно свойства описываются в главе 9.

Ключевое слово __published

Ключевое слово __published допускается использовать в классах-наследниках объекта TObject. Правила видимости для раздела __published класса библиотеки VCL те же, что и для public, с добавлением RTTI-информации, генерируемой для объявленных членов-данных и свойств. RTTI-информация позволяет запрашивать члены-данные, свойства и члены-функции класса.

RTTI-информация также очень полезна для обработчиков событий. Все стандартные события имеют параметр, который передается с типом TObject*. Этот параметр можно запросить, если неизвестен тип передающего объекта. В следующем примере показано, как это выполняется при вызове события OnButtonClick после щелчка мышью или организованном вручную вызове, например Button1Click(NULL). Обратите внимание, что вручную вызов можно организовать, передавая указатель другой кнопки, например Button1Click(Button2). В листинге 8.7 показан пример такой ситуации.

Листинг 8.7. Использование RTTI-информации для запрашивания типа объекта

```
void __fastcall TForm1::Button1Click(TObect *Sender)
{
   TButton *Button = dynamic_cast<TButton *>(Sender);
   if (Button)
   {
      // Какие-то действия по нажатию кнопки.
   }
   else
   {
      // Какие-то действия после передачи
      // значения NULL для *Sender.
   }
}
```

Механизм управления потоками

Как уже неоднократно говорилось, C++Builder является средой быстрой разработки приложений (или RAD-средой). Отчасти это объясняется наличием GUI-интерфейса и объектно-ориентированной природой самого языка. Кроме того, в C++Builder используется механизм управления потоками (т.е. управление процессами чтения и записи) для хранения значений свойств.

Во время создания приложения компоненты могут иметь разные свойства. IDE-среда хранит эти значения (подробнее об этом сказано чуть ниже) в виде части формы, к которой эти компоненты относятся. Формы хранятся как часть проекта в файле с расширением .dfm. Сохраненные в файле формы сведения будут загружены при выполнении программы. Другими словами, эти свойства являются устойчивыми.

В C++Builder 5 предусмотрена возможность сохранения формы в текстовом или двоичном формате (как и в предыдущих версиях C++Builder). Выберите команду меню Tool⇨Environment Options, а затем в диалоговом окне Environment Options — вкладку Preferences. В ней есть специальный параметр New forms as text, который предназначен

для указания способа сохранения формы в текстовом формате. Благодаря этому параметру файл формата .dfm можно открыть в любом текстовом редакторе.

Стандартный класс C/C++ содержит три основных раздела: public, private и protected. В C++Builder также предусмотрен дополнительный раздел __published. Расширение __published доступно только для классов, которые являются наследниками класса TObject.

```
class TSample : public TObject
{
public:
private:
protected:
__published:
}
```

Раздел __published используется для определения хранимых свойств, т.е. свойств, которые записываются в файл формы при создании приложения. В качестве примера откройте любой проект в среде C++Builder и щелкните правой кнопкой мыши на какой-либо из форм. Из контекстного меню выберите команду View as Text для просмотра способа хранения информации о форме. Щелкните правой кнопкой мыши на коде и выберите команду View as Form для возврата к графическому представлению формы. Сохранение свойств выполняется с помощью механизма Store-and-Load Mechanism, который в большинстве случаев удовлетворяет требованиям создателя компонента. Однако в следующем разделе описываются ситуации, когда средств этого механизма все же недостаточно.

При создании компонента разработчик задает набор используемых по умолчанию значений для его опубликованных (published) свойств. Эти значения присваиваются конструктором компонента, и при создании приложения пользователь может модифицировать эти свойства с помощью окна Object Inspector. На самом деле каждое свойство, представленное в окне Object Inspector, является опубликованным. Свойства могут быть объявлены в разделе public определения компонента (класса), но в таком случае они будут доступны только во время выполнения.

Хотя в главе 9 свойства описываются более подробно, здесь нам все же понадобятся краткие сведения о принимаемых по умолчанию и хранимых свойствах. Рассмотрим два свойства: одно с принимаемым по умолчанию значением 10, а другое с указанием того, что оно не является хранимым.

```
__property int SomeProperty1 = {read=FProp1,
                                write=FProp1,
                                default=10};
__property AnsiString SomeProperty2 = {read=FProp2,
                                       stored=false};
```

В первом случае значение 10 всегда присваивается не в самом объявлении SomeProperty1, а в конструкторе. Ключевое слово default используется для того, чтобы сообщить IDE-среде о сохранении значения этого свойства в файле формы только в том случае, если оно имеет значение, отличное от 10.

Во втором случае для свойства SomeProperty2 предписывается не хранить его значение в файле формы. Примером такого свойства может быть указание текущей версии компонента. Так как номер версии не может измениться, его не нужно хранить в файле формы.

При сохранении компонента информация о значении свойства, отличном от заданного по умолчанию, записывается в файл формы. При следующем открытии формы создается экземпляр каждого компонента, для свойств компонентов задаются их значения по умолчанию, а затем считываются и присваиваются хранимые (т.е. отличные от предлагаемых по умолчанию) значения.

Как видите, программисту не нужно беспокоиться о создании компонентов и управлении потоками для связанного свойства.

Соблюдение требований при работе с потоками

Свойства компонентов могут иметь числовой, символьный, строковый, перечислимый тип (включая логический), тип набора или даже более сложный тип составного объекта, например пользовательской структуры и класса. C++Builder имеет встроенные инструкции для управления потоками при работе с простыми типами данных. Поддержка потоковой работы с пользовательскими объектами предусмотрена также для классов, производных от класса `TPersistent`.

Класс `TPersistent` позволяет присваивать элементы одних объектов другим, а также выполнять чтение и запись их свойств в поток и из потока. Более подробные сведения по этому поводу можно найти в оперативной справке.

Некоторые типы свойств, например массивы, должны иметь собственный редактор свойств. Без такого редактора инспектор объектов Object Inspector не может предоставить программисту интерфейс для редактирования содержимого этого свойства. Более подробно эта тема рассматривается в главе 10, посвященной созданию редакторов свойств и компонентов. А более подробные сведения о свойствах и способах их использования можно найти в главе 9.

Передача в поток неопубликованных свойств

До сих пор нам было известно, что инспектор объектов Object Inspector во время создания приложения предоставляет программисту интерфейс для доступа к опубликованным свойствам компонента. Кроме того, свойства, производные от класса `TPersistent`, могут передаваться в поток файла формы и извлекаться из него. Это значит, что существует способ создания устойчивых свойств, которые отсутствуют в окне Object Inspector. Помимо этого можно создавать потоковые методы для свойств, для которых в C++Builder не предусмотрена возможность чтения или записи.

Сохранение неопубликованного свойства достигается за счет создания кода, который сообщает C++Builder о способе чтения и записи значений этого свойства. Процесс состоит из следующих этапов.

- Переопределение метода `DefineProperties()`. Прежде определенные методы передаются *объекту-навигатору* (*filer object*).
- Создание методов чтения и записи значения свойства.

Выше уже говорилось о том, что опубликованные свойства автоматически передаются в поток файла формы. Этот процесс осуществляется методами чтения и записи, которые определены для разных типов свойств. Имена свойств и используемые методы передачи данных в поток определяются методом `DefineProperties()`. Для передачи в поток неопубликованного свойства нужно переопределить этот метод.

В листинге 8.8 приведен пример компонента `TSampleComp`, который имеет три неопубликованных свойства. Этот компонент может передавать в поток значения своих свойств с помощью специальных методов. Он создает экземпляр второго компонента `TComp` во время выполнения приложения и ссылается на него с помощью свойства `Comp3`. Так как этот компонент создан в форме без прямого участия разработчика, то его свойства не будут автоматически передаваться в файл формы. Попробуем добавить в компонент код, необходимый для передачи в поток этой информации. Изучите внимательно код в листинге 8.8 , прежде чем мы приступим к его обсуждению.

Листинг 8.8. Передача в поток значений неопубликованных свойств

```
// Минимальное объявление класса,
// необходимое для компиляции данного примера
class TComp : public TComponent
{
public:
   __fastcall TComp::TComp(TComponent *Owner)
     : TComponent(OWner){}
};

class TSampleComp : public TComponent
{
private:
   int FProp1;
   AnsiString FProp2;
   TComp *FComp3;

   void __fastcall ReadProp1(TReader *Reader);
   void __fastcall WriteProp1(TWriter *Writer);
   void __fastcall ReadProp2(TReader *Reader);
   void __fastcall WriteProp2(TWriter *Writer);
   void __fastcall ReadComp3(TReader *Reader);
   void __fastcall WriteComp3(TWriter *Writer);

protected:
   void __fastcall DefineProperties(TFiler *Filer);

public:
   __fastcall TSampleComp(TComponent*Owner);
   __fastcall ~TSampleComp(void);

   __property int Prop1 =
      {read = FProp1, write = FProp1, default = 10};
   __property AnsiString Prop2 =
      {read = FProp2, write = FProp2, nodefault};
   __property TComp *Comp3 = {read = FComp3, write = FComp3};
};

void __fastcall TSampleComp::TSampleComp(TComponent*Owner) :
   TComponent(Owner)
{
   FProp1 = 10;              // По умолчанию эти свойства
   FComp3 = new TComp(NULL); // передаются в поток.
}

void __fastcall TSampleComp::~TSampleComp (void)
{
   if(FComp3)
   delete FComp3;
}
```

```
void __fastcall TSampleComp::DefineProperties(TFiler *Filer)
{
   // Сначала вызов базового метода
   TComponent::DefineProperties(Filer);

   Filer->DefineProperty("Prop1", ReadProp1, WriteProp1,
                         (FProp1 != 10));
   Filer->DefineProperty("Prop2", ReadProp2, WriteProp2,
                         (FProp2 != ""));

   // Определения, необходимые для записи свойств Comp3
   bool WriteValue;
   if(Filer->Ancestor) // проверка унаследованного значения
   {
      TSampleComp *FilerComp =
         dynamic_cast<TSampleComp *>(Filer->Ancestor);
      if(FilerComp->Comp3 == NULL)
         WriteValue = (Comp3 != NULL);
      else
      {
         if( (Comp3 == NULL)
             || (FilerComp->Comp3->Name != Comp3->Name) )
            WriteValue = true;
         else
            WriteValue = false;
      }
   }
   else // Нет унаследованного свойства,
        // записать свойство, если оно не равно null
      WriteValue = (Comp3 != NULL);

   Filer->DefineProperty("Comp3", ReadComp3, WriteComp3,
                              WriteValue);
   }

void __fastcall TSampleComp::ReadProp1(TReader *Reader)
{
   Prop1 = Reader->ReadInteger();
}

void __fastcall TSampleComp::WriteProp1(TWriter *Writer)
{
   Writer->WriteInteger(FProp1);
}

void __fastcall TSampleComp::ReadProp2(TReader *Reader)
{
   FProp2 = Reader->ReadString();
}

void __fastcall TSampleComp::WriteProp2(TWriter *Writer)
{
```

```
   Writer->WriteString(FProp2);
}

void __fastcall TSampleComp::ReadComp3(TReader *Reader)
{
   if(Reader->ReadBoolean())
   FComp3 =( TComp *)Reader->ReadComponent(NULL);
}

void __fastcall TSampleComp::WriteComp3(TWriter *Writer)
{
   if(FComp3)
   {
      Writer->WriteBoolean(true);
      Writer->WriteComponent(Comp3);
   }
   else
      Writer->WriteBoolean(false);
}
```

Рассмотрим сначала самые простые свойства. Метод `DefineProperties()` содержит следующие строки кода для регистрации первых двух свойств:

```
Filer->DefineProperty("Prop1", ReadProp1, WriteProp1,
                      (FProp1 != 10));
Filer->DefineProperty("Prop2", ReadProp2, WriteProp2,
                      (FProp2 != ""));
```

Они сообщают C++Builder об использовании методов чтения и записи, которые предусмотрены для передачи в поток значений этих свойств. Последний параметр является флагом, который указывает на наличие данных, которые требуется сохранить. Свойства `Prop1` и `Prop2` необходимо сохранить, только если их значения отличаются от используемого по умолчанию значения.

Код свойства `Comp3` выглядит иначе, и для него требуются дополнительные пояснения. Это свойство отличается от других тем, что является компонентом (инициализированным при выполнении приложения), а не простым типом данных. В листинге 8.9 представлен раздел кода, который определяет необходимость передачи в поток данных этого компонента.

Листинг 8.9. Определение необходимости передачи в поток данных свойства, которое является компонентом

```
bool WriteValue;
if(Filer->Ancestor) // Проверка унаследованного значения
{
   TSampleComp *FilerComp =
      dynamic_cast<TSampleComp *>(Filer->Ancestor);

   if(FilerComp->Comp3 == NULL)
      WriteValue = (Comp3 != NULL);
   else
   {
      if( (Comp3 == NULL)
          || (FilerComp->Comp3->Name != Comp3->Name))
         WriteValue = true;
```

```
      else
         WriteValue = false;
   }
}
else // Нет унаследованного значения,
     // записать значение свойства, если оно не равно null
   WriteValue = (Comp3 != NULL);

Filer->DefineProperty("Comp3", ReadComp3, WriteComp3,
                      WriteValue);
```

Это свойство представляет компонент, инициализированный во время выполнения. Так как компонент не был вставлен в форму вручную, то для его свойств не выполняется предусмотренный по умолчанию механизм передачи данных в поток. Об этом сможет позаботиться метод `DefineProperties()`.

Сначала нужно проверить значение свойства `Ancestor`, и если оно равно `true`, то нужно избежать сохранения значения свойства в унаследованных формах. Если унаследованного значения нет, то приступать к передаче в поток значения этого свойства (который является компонентом) можно, если `Comp3` не равно `NULL`.

Если свойство `Ancestor` навигатора имеет значение `true`, нужно рассмотреть свойство `Comp3` предка. А если свойство `Ancestor` навигатора имеет значение `NULL`, то в таком случае нужно передать в поток значение свойства `TSampleComp>Comp3` при условии, что оно не равно `NULL`. Если свойство `Comp3` свойства `Ancestor` предка навигатора не равно `NULL`, выполняются две заключительные проверки. Если свойство `TSampleComp>Comp3` равно `NULL` или имя нашего свойства `Comp3` отличается от имени свойства предка, то приступаем к передаче в поток этого свойства (которое является компонентом).

Наконец, с помощью метода `DefineProperty()` определим наше свойство так, как уже говорилось выше.

Конечно, довольно трудно представить себе свойство, которое является компонентом. Просмотрите код еще раз и попытайтесь понять его суть.

В заключение замечание рассмотрим метод `DefineProperty()`, который используется для работы с такими типами данных, как числа, строки, символы, а также с логическими и перечислимыми значениями. Еще один метод `DefineBinaryProperty()` предназначен для передачи в поток таких бинарных данных, как изображения и звук. Более подробные сведения на эту тему можно получить в разделе `DefineBinaryProperty` части `TWriter` в интерактивной справке.

Обновления универсальных элементов управления

В отличие от других оконных сред, в Microsoft Windows предусмотрено несколько предопределенных классов элементов управления, которые с небольшими изменениями могут использоваться во всех приложениях Windows. В категорию стандартных элементов управления входят кнопки, списки, поля со списками, статичные элементы управления, текстовые поля и полосы прокрутки. Эти типы элементов управления предварительно зарегистрированы и могут использоваться без явного вызова API-функции `RegisterClass()`. Другие типы классов элементов управления, которые называются *универсальными элементами управления* (*common controls*), включают списочные представления, иерархические представления, пане-

ли инструментов, заголовки и много другое. В отличие от стандартных элементов управления, для регистрации классов универсальных элементов управления нужно использовать API-функцию `InitCommonControlEx()`. Эта функция также нужна для загрузки библиотеки универсальных элементов управления.

Библиотека универсальных элементов управления

Большинство универсальных элементов управления находится в специальном файле DLL-библиотеки, `COMCTL32.DLL`. В отличие от стандартных элементов управления, универсальные элементы управления часто обновляются, а потому существует несколько версий файла `COMCTL32.DLL`. Обычно этот файл обновляется при выпуске новой версии броузера Microsoft Internet Explorer.

Нужно соблюдать осторожность при использовании универсальных элементов управления, которые входят в состав только самых новых версий DLL-файла. Более того, классы, которые входят в состав всех версий библиотеки, могут выполнять функции только в новых версиях библиотеки. В таком случае для гарантированной совместимости элементов управления и файла библиотеки рекомендуется использовать API-функцию `DllGetVersion()` (4.71+) и макрос `_WIN32_IE`. В табл. 8.1 приведен список разных версий DLL-библиотеки универсальных элементов управления с указанием соответствующей им платформы и результата макроса `_WIN32_IE`.

Таблица 8.1. Версии DLL-библиотеки универсальных элементов управления

Версия файла `COMCTL32.DLL`	Платформа	Результат макроса `_WIN32_IE`
4.00	Windows 95/NT 4.0	0x0200
4.70	Internet Explorer 3.X	0x0300
4.71	Internet Explorer 4.0	0x0400
4.72	Windows 98/IE 4.01	0x0500
5.80	Internet Explorer 5.0	0x0500
5.81	Windows 2000	0x0501

Код в листинге 8.10 демонстрирует способ использования функции `DllGetVersion()` для определения версии библиотеки универсальных элементов управления.

Листинг 8.10. Пример использования `DllGetVersion()`

```
#include <shlwapi.h>
unsigned int __fastcall GetComctl32Version()
{
   //
   // == GetComctl32Version() ==
   // Определяет, реализована ли в DLL-библиотеке
   // функция DllGetVersion(). Если это так, извлекает
   // информацию о ее версии. Возвращаемое значение равно
   // произведению номера версии на 100, если функция
   // DllGetVersion() имеется. В противном случае
   // возвращаемое значение равно нулю.
   //
   // == См. также ==
   // http://msdn.microsoft.com/library/psdk/shellcc/shell/
```

```
↳Versions.htm
   // http://support.microsoft.com/support/kb/articles/Q186/
↳1/76.ASP
   //

   unsigned int result = 0;
   HINSTANCE hLib = LoadLibrary("COMCTL32.DLL");
   if (hLib)
   {
      DLLGETVERSIONPROC DllGetVersion =
         reinterpret_cast<DLLGETVERSIONPROC>(
            GetProcAddress(hLib, "DllGetVersion")
         );
      if (DllGetVersion)
      {
         DLLVERSIONINFO version_info;
         memset(&version_info, 0, sizeof(DLLVERSIONINFO));
         version_info.cbSize = sizeof(DLLVERSIONINFO);

         if (SUCCEEDED(DllGetVersion(&version_info)))
         {
            result =
               100 * version_info.dwMajorVersion +
               version_info.dwMinorVersion;
         }
      }
      FreeLibrary(hLib);
   }
   return result;
}

__fastcall TForm1::TForm1(TComponent*Owner)
   : TForm(Owner)
{
   //
   // Проверка версии DLL-библиотеки универсальных элементов
   // управления: она не должна быть старше, чем 4.71...
   //
   if (GetComctl32Version() < 471)
   {
      throw EWin32Error(
         "Для работы этой программы нужно иметь\n "
         "библиотеку универсальных элементов не старше 4.71.\n"
         "Более подробные сведения о модернизации версии \n"
         "DLL-библиотеки можно найти по адресу: \n \n "
         "http://support.microsoft.com/support/kb/articles
↳/Q186/1/76.ASP"
         );
   }
}
```

Действительно, компоненты библиотеки VCL, которые содержат классы универсальных элементов управления, предоставляют функции, которые доступны только в новых версиях библиотеки универсальных элементов управления. При работе с компонентом, который использует новые функции, следует учесть необходимость установки нужной версии этой DLL-библиотеки на той платформе, где предполагается использовать приложение. Часто эта версия отличается от версии, используемой на платформе, где разрабатывалось приложение. В библиотеке VCL не предусмотрены средства проверки совместимости библиотеки универсальных элементов управления и специальных компонентов или их характеристик. Поэтому при использовании в проекте компонентов на основе универсальных элементов управления рекомендуется использовать функцию проверки версии библиотеки `GetComctl32Version()`.

Модернизация универсальных элементов управления C++Builder

Библиотека VCL содержит компоненты, которые инкапсулируют многие классы универсальных элементов управления Windows. Точнее говоря, те компоненты, которые расположены во вкладке Win32 палитры компонентов Component Palette.

При каждой модернизации библиотеки универсальных элементов управления в ней появляются новые структуры, сообщения и функции. Разработчики библиотеки VCL постоянно стремятся включать в компоненты этой библиотеки новые конструкции. Этот непрекращающийся процесс повышает ценность каждой новой версии C++Builder по сравнению с предыдущими. Действительно, в DLL-библиотеке универсальных элементов управления C++Builder 5 содержится много дополнений и усовершенствований. В этом разделе рассматриваются только некоторые обновления C++Builder 5:

- обновления класса `TListView`;
- обновления класса `THeaderControl`;
- поддержка пользовательской панели класса `TToolBar`.

Обновления класса `TListView`

Класс `TListView` инкапсулирует представление списка Windows, который, несомненно, является наиболее разносторонним из всех компонентов библиотеки VCL. Класс `TListView` позволяет использовать практически любой аспект соответствующего ему представления списка. Рассмотрим несколько обновлений этого класса, которые появились в C++Builder 5. При этом мы ограничимся только теми аспектами, которые связаны с библиотекой универсальных элементов управления.

Изображения подчиненных элементов

Расширенный стиль `LVS_EX_SUBITEMIMAGES`, который впервые появился в библиотеке `COMCTL32.DLL` версии 4.70, элемента управления с представлением списка позволяет отображать возле каждого подчиненного элемента рисунок. Этот стиль особенно полезен в тех случаях, когда каждый столбец используется для представления информации, которая лучше всего воспринимается вместе с рисунком. Класс `TListView` по умолчанию добавляет этот стиль ко всем остальным стилям. Класс `TListItem` содержит свойство `SubItemImages`, которое можно использовать для организации связи между изображением (из соответствующего свойства `TListView::SmallImages`) и подчиненным элементом представления списка.

Пауза перед выделением элемента после размещения на нем указателя мыши

Сообщение LVM_SETHOVERTIME впервые появилось в библиотеке COMCTL32.DLL в версии 4.71. При использовании его совместно с расширенным стилем LVS_EX_TRACKSELECT оно позволяет приложению регулировать продолжительность паузы между размещением на элементе указателя мыши и его выделением, т.е. *время висения* (*hovering time*). Это сообщение поддерживается в классе TListView с помощью свойства TListView::HoverTime. Задавая значение true для свойства TListView::HotTrack и нужное значение паузы в миллисекундах для свойства HoverTime, можно управлять временем висения.

Рабочие области

Уникальной возможностью представления списка является способность поддерживать работу с несколькими рабочими областями. То есть, клиентская область представления списка может быть разбита на несколько прямоугольных областей. Эта возможность впервые появилась в библиотеке COMCTL32.DLL версии 4.71 в виде расширенного стиля LVS_EX_MULTIWORKAREAS и сообщений LVM_GETWORKAREAS, LVM_SETWORKAREAS и LVM_GETNUMBEROFWORKAREAS. Эта функциональная возможность поддерживается в классе TListView с помощью свойства TListView::WorkAreas. Это свойство имеет тип TWorkAreas* и содержит набор объектов TWorkArea. Класс TWorkArea содержит свойство TWorkArea::Rect, которое может быть использовано для определения рамки рабочей области. Сообщение LVM_SETWORKAREAS используется именно в этом классе, а точнее — в закрытом члене-функции TWorkArea::Update.

Класс TListView также содержит свойство ShowWorkAreas, которое используется совместно с расширенным стилем LVS_EX_MULTIWORKAREAS. Класс TListView расширяет поддержку рабочих областей в базовом представлении списка возможностями окрашивания рамки и введения в ней заголовка. Эти функции можно реализовать, используя свойства TWorkArea::Color и TWorkArea::Caption класса TListView и управляя сообщением WM_PAINT и внутренним членом-функцией TCustomListView::DrawWorkAreas.

Подсказки

Расширенный стиль LVS_EX_INFOTIP позволяет отображать подсказки для каждого элемента в представлении списка. Впервые он появился в библиотеке COMCTL32.DLL версии 4.71. Он также генерирует соответствующее сообщение LVN_GETINFOTIP. В ответ на это сообщение класс TListView инициирует событие OnInfoTip. Создавая обработчик этого события, можно отображать окно с подсказкой для каждого элемента представления списка.

Обновления класса THeaderControl

Класс THeaderControl инкапсулирует заголовок Windows, который наиболее широко известен именно благодаря использованию в представлении списка. В DLL-библиотеке универсальных элементов управления версии 4.70 вводится несколько стилей и сообщений, которые позволяют использовать дополнительные функциональные возможности заголовков. Класс THeaderControl обновлен в C++Builder версии 5 как раз в соответствии с этими новшествами.

Перестановки с помощью мыши

В классе THeaderControl с помощью свойства DragReorder предусмотрена поддержка стиля HDS_DRAGDROP. Это свойство позволяет указать на возможность перетаскивания с помощью мыши отдельных частей заголовка. Если это свойство имеет значение true, пользователь может просто с помощью мыши перетащить этот раздел заголовка в новое место. Для этого в классе THeaderControl предусмотрены свойство DragReorder и события OnSectionDrag и OnSectionEndDrag. Первое событие возникает в ответ на сообщение

HDN_ENDDRAG, которое посылается после завершения операции перетаскивания, второе — в ответ на сообщение NM_RELEASECAPTURE, которое посылается после отпускания кнопки мыши, что неявно означает завершение перетаскивания.

Изменение размеров

Стиль HDS_FULLDRAG позволяет задавать новые размеры для разделов заголовка. Результат этого действия аналогичен эффекту при изменении размеров окна с указанием параметра Show Window Contents While Dragging (Показывать содержимое окна при перетаскивании) во время установки параметров рабочего стола. Действительно, стиль HDS_FULLDRAG не будет иметь никакого эффекта, если не задан этот параметр. Этот стиль поддерживается в классе THeaderControl с помощью свойства FullDrag.

Настройка панели инструментов с помощью класса TToolBar

Класс TToolBar инкапсулирует панель инструментов Windows. В DLL-библиотеке версии 4.70 предусмотрена поддержка настройки (служба Custom Draw) многих элементов управления, включая панели инструментов. В версии 4.71 содержится также несколько специальных усовершенствований функций для настройки панели инструментов. В соответствии с этим класс TToolBar расширен в C++Builder 5 для поддержки службы настройки ***Custom Draw*** и ее новых свойств.

Как уже говорилось выше, служба ***Custom Draw*** впервые появилась в версии 4.70 DLL-библиотеки универсальных элементов управления. Точнее, в этой версии появились сообщение NM_CUSTOMDRAW, структура NMCUSTOMDRAW, стиль TBSTYLE_CUSTOMERASE, а также специализированные флаги TBCDRF_NOEDGES, TBCDRF_HILITEHOTTRACK, TBCDRF_NOOFFSET, TBCDRF_NOMARK, TBSTATE_MARKED и TBCDRF_NOETCHEDEFFECT.

Поддержка службы настройки ***Custom Draw*** в классе TToolBar осуществляется с помощью событий OnCustomDraw, OnCustomDrawButton, OnAdvancedCustomDraw и OnAdvancedCustomDrawButton. Более того, с помощью типа TTBCustomDrawFlags предусмотрена поддержка специализированных флагов. К сожалению, стиль TBSTYLE_CUSTOMERASE не поддерживается в C++Builder 5. Более подробные сведения о событиях службы настройки ***Custom Draw*** класса TToolBar можно найти далее в этой главе.

Итоговые замечания по поводу обновлений универсальных компонентов управления C++Builder 5

Библиотека универсальных компонентов управления постоянно совершенствуется. Это побуждает разработчиков Borland постоянно обновлять библиотеку VCL для отражения этих изменений. При этом классы, инкапсулирующие универсальные компоненты управления, всегда на один шаг отстают от соответствующих им API-функций. Доказательством этого является наличие новых функций в библиотеке универсальных компонентов, которые отсутствуют в библиотеке VCL, или обещание реализовать их в следующей версии. Например, служба настройки ***Custom Draw*** для ползунковых регуляторов (track bar) была предусмотрена еще в версии 4.70, но она до сих пор не реализована в библиотеке VCL. Хотя следует признать, что в библиотеке VCL предусмотрена поддержка наиболее распространенных компонентов и допускается прямое использование Windows API-функций. Более того, на примере C++Builder 5 можно убедиться в том, что в библиотеке VCL постоянно адаптируются и даже расширяются некоторые аспекты библиотеки универсальных элементов управления. Например, в классе TListView поддержка нескольких рабочих областей намного превышает возможности базового представления списка. Несомненно, что в следующих версиях эти улучшения и обновления будут еще более значительными.

Другие обновления библиотеки VCL

В C++Builder 5 введены также другие обновления библиотеки VCL, которые существенно упрощают работу разработчиков C++Builder. Некоторые из них перечислены в следующих разделах этой главы.

Новые свойства подсказок и команд меню

Теперь с помощью новых свойств `HintFont` и `MenuFont` класса `TScreen` можно задавать шрифт для подсказок и команд меню. А с помощью параметра `State` нового события `OnAdvancedDrawItem` можно создавать пользовательские команды меню на основе дополнительных состояний, которых нет у события `OnDrawItem` класса `TMenuItem`. Например, неактивен (disabled), серый (grayed), используется по умолчанию (default) и активен (inactive).

Новое свойство `AutoHotKeys` классов `TMainMenu` и `TMenuItem` может быть использовано для автоматической установки клавиш ускоренного доступа к командам меню с помощью свойства `maAutomatic`. Этот параметр также позволяет разрешать конфликтные ситуации при совпадении клавиш ускоренного доступа за счет присвоения разных клавиш.

Команды контекстного меню теперь можно создавать с помощью нового события `OnContextPopup` класса `TControl`. Например, можно отобразить пользовательское контекстное меню, диалоговое окно, изображение или что-либо другое в ответ на щелчок правой кнопкой мыши на элементе управления.

С помощью свойства `MenuAnimation` класса `TPopupMenu` можно управлять способом представления контекстного меню на экране, т.е. его можно отобразить стандартным образом или с помощью анимационного раскрытия слева, справа, сверху или снизу. Эта функция может быть реализована только для операционных систем Windows 98, Windows NT или более поздних версий.

Доступ к реестру

Наиболее долгожданной новинкой является свойство `Access` классов `TRegistry`, `TRegistryIniFile` и `TRegIniFile`. Оно позволяет указать уровень доступа при использовании открытых ключей реестра. Стандартными значениями этого свойства являются `KEY_ALL_ACCESS`, `KEY_READ` и `KEY_CREATE_SUB_KEY`.

Ранее для использования этих функций для чтения ключей раздела `HKEY_LOCAL_MACHINE` в операционной системе Windows NT без административных привилегий требовалось использовать компоненты сторонних разработчиков. При этом разработчику часто приходилось сталкиваться с тем, что такое приложение хорошо работало в операционной системе Windows 98 и плохо — в операционной системе Windows NT.

Усовершенствования документации библиотеки VCL

Документация библиотеки VCL теперь содержит список процедур для каждого модуля VCL. Его можно найти в разделе “Routines Listing, by Unit/Header” руководства “Visual Component Library Reference” интерактивной справки C++Builder. Кроме того, можно, щелкнув на имени модуля в соответствующем разделе справки для нужного объекта, отобразить список всех классов этого модуля. На рис. 8.3 показан такой список для модуля `stdctrls`, в котором определен класс кнопки `TButton`.

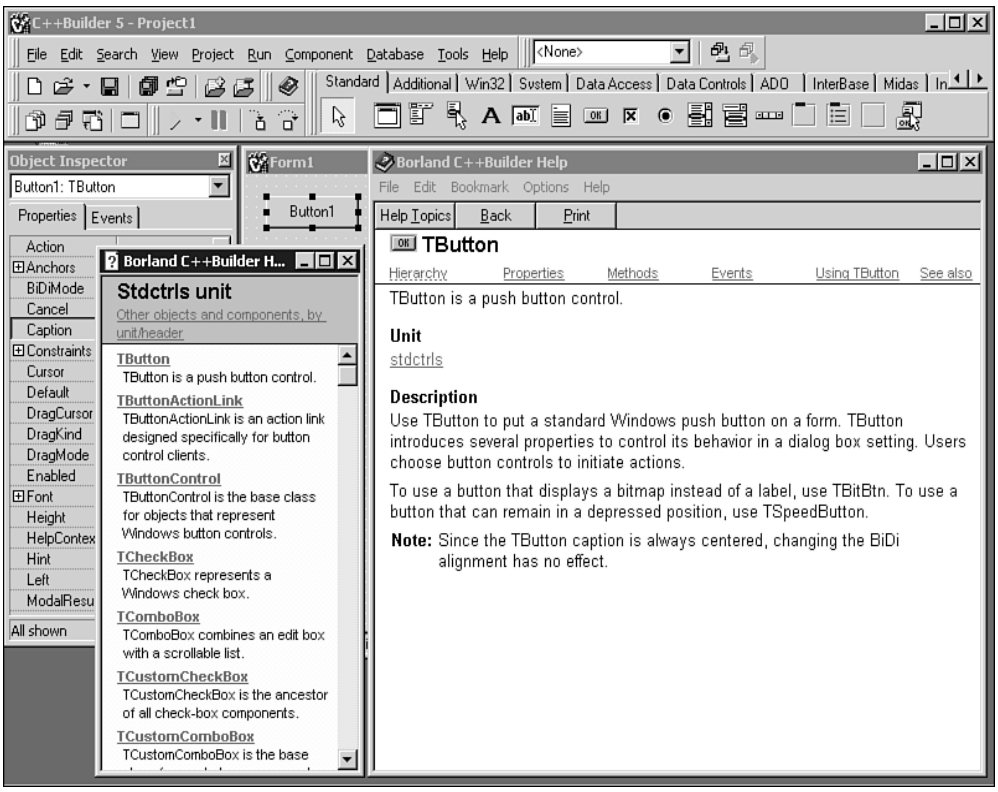


Рис. 8.3. Список компонентов модуля `stdctrls`, в котором определен класс кнопки `TButton`

Новый компонент `TApplicationEvents`

Новый компонент `TApplicationEvents` (во вкладке компонентов Additional) упрощает создание обработчиков событий на уровне приложения с помощью окна Object Inspector. Опуская компонент `TApplicationEvents` в форму, можно довольно легко создать обработчики для таких событий, как, например, `OnActive`, `OnDeactivate`, `OnIdle` или `OnMessage`. Это делается с помощью двойного щелчка на нужном свойстве события компонента `TApplicationEvents` в инспекторе объектов Object Inspector так же, как это обычно делается для любых других компонентов.

Улучшения класса `TIcon`

Класс `TIcon` теперь поддерживает работу с пиктограммами разных размеров и с палитрами, содержащими больше 16 цветов. При этом в приложении будет использована та пиктограмма, которая в наибольшей степени соответствует заданным свойствам `Width` и `Height`. В этом случае значения свойств `Width` и `Height` имеют лишь рекомендательный характер, а фактическое изображение пиктограммы будет растянуто или сжато, если заданные и фактические размеры не будут совпадать.

Другие усовершенствования библиотеки VCL

Среди других усовершенствований библиотеки VCL следует упомянуть свойство `AutoSnap` компонента `TSplitter`, которое используется для обработки попытки пользователя перетащить разделительную полосу, чтобы сделать объект меньше его минимального размера. Новые свойства `BiDiKeyboard` и `NonBiDiKeyboard` класса `TApplication` задают раскладку клавиатуры, а новый модуль поддержки контейнеров `Contnrs` содержит несколько классов для управления стеками и очередями.

Более подробные сведения обо всех усовершенствованиях библиотеки VCL можно получить в интерактивной справке C++Builder.

Расширения библиотеки VCL — не только объект `TStringList`

Класс `TStringList` — один из самых популярных классов для быстрого создания приложений в тех случаях, когда оптимизация работы программы по скорости и эффективности использования оперативной памяти не имеют первостепенного значения. Класс `TStringList` представляет собой очень удобный контейнер для хранения строк и связанных с ними объектов. Он содержит несколько свойств и методов, которые позволяют манипулировать этими строками и хранимыми объектами.

В этом разделе представлен типичный пример использования наследника класса `TStringList` для хранения текстовой информации типа `AnsiString` и связанной с ней вспомогательной информации, которая описывает этот текст. В данном случае необходимо учесть все доступные сеансы базы данных, псевдонимы, таблицы и связанные поля в системе. Этот класс должен быть достаточно универсальным, чтобы в нем можно было сохранить всю связанную с базой данных информацию (о сеансах, псевдонимах, таблицах и полях) без необходимости поддерживать работу сразу с четырьмя разными типами объектов.

Применение класса `TStringList` в качестве контейнера

Справочные файлы C++Builder содержат достаточно полные сведения о применении класса `TStringList`. Здесь же мы рассмотрим способ сохранения целых объектов по сравнению с сохранением отдельных строк. Для более близкого знакомства со свойствами и методами класса `TStringList` настоятельно рекомендуется внимательно прочесть руководство Visual Component Library Reference из справки C++Builder. Изучите сначала свойства `Strings`, `Objects` и `CommaText`, а затем — методы `Add()`, `Delete()` и `IndexOf()`.

Объект `TStringList` — не просто класс, который способен хранить строки с возможностями сортировки, обмена данными, вставки данных, добавления и удаления элементов. Помимо этого он может хранить ссылки на другие объекты библиотеки VCL. Зачем это нужно?

Если внимательно рассмотреть объявление свойства `Objects`, можно заметить, что в нем возвращается указатель типа `TObject*`.

```
__property System::TObject*Objects[int Index] =
  {read=GetObject, write=PutObject};
```

Теперь возникает вопрос о том, как с его помощью хранить информацию из вашего приложения. Если нужно сохранить указатель на существующий объект или компонент библиотеки VCL, то присвоение выполняется с помощью приведения типа этого объекта к типу TObject*.

```
StringList1->Objects[index] = dynamic_cast<TObject *>(Memo1);
```

А при извлечении хранимой информации потребуется выполнить обратное приведение типа.

```
TObject *MyObject = StringList1->Objects[i];
TMemo *MyMemo = dynamic_cast<TMemo *>(MyObject);
```

В этом примере кода предполагается, что объект ссылается на объект TMemo. Перед использованием ссылки MyMemo в приложении следует проверить наличие у нее определенного значения, сравнивая его со значением NULL.

Сохранение других объектов

Свойство Objects класса TStringList имеет тип TObject и, следовательно, должно содержать ссылку на объект этого типа или его наследников. Для хранения структур или классов в списке строк следует создать класс-наследник класса TObject, как показано в листинге 8.11.

Листинг 8.11. Создание класса-наследника класса Tobject для хранения объектов

```
class MyStruct : public Tobject
{
public:
   int Age
   char Sex;
   bool Married;
};
```

Создав класс-наследник для класса TObject (или для класса-наследника класса TObject), можно связать эту структуру с нашим списком. В этом разделе мы рассмотрим пример класса-наследника для класса TStringList, который будет хранить сведения обо всех доступных сеансах работы с базой данных, а именно имена сеансов в свойстве Strings. Затем будут определены все псевдонимы этих сеансов. Эта информация будет храниться в другом экземпляре класса того же типа и будет связана с родительским классом с помощью свойства Objects. Этот процесс будет продолжен для таблицы каждого псевдонима, а также для каждого поля во всех таблицах.

Как предполагается достигнуть этой цели?

- Класс сеанса должен содержать имена сеансов и ссылки на все связанные с ними псевдонимы.
- Класс псевдонима должен содержать имена псевдонимов и ссылки на все связанные с ними таблицы.
- Класс таблицы должен содержать имена таблиц и ссылки на все связанные с ними поля.
- Класс поля должен содержать имена полей.

Во всех четырех случаях потребуется сохранить список имен и их объектов. Так как объект для каждого имени хранит связанную информацию, то можно создать объекты одинакового типа и заполнить их разными данными.

В папках TDBList-Lite и TDBList-Full на прилагаемом к книге компакт-диске находятся две версии проекта-примера TDBListDemo.bpr. Упрощенная версия (Lite) демонстрирует рас-

смотренные в предыдущих разделах методы и может использоваться как справочное пособие при чтении этого раздела. Полная версия (Full) включает расширения упрощенной версии, которые кратко описаны ниже в этой главе.

На заметку

В зависимости от конфигурации системы, пользователи приложений могут увидеть несколько диалоговых окон с предложением подключения к базам данных. Это вызвано тем, что при инсталляции C++Builder было задано несколько драйверов BDE (узнать об этом можно с помощью команды Start⇨Settings⇨Control Panel⇨BDE Administrator). Если вам не известна необходимая для регистрации информация, нажмите на клавишу <Escape>. При запуске этого приложения в IDE-среде из-за сбоев процесса регистрации подключения к базе данных могут встретиться исключительные ситуации. Представленный в приложении код не предназначен для полной обработки всех исключительных ситуаций. Поэтому пользователю предлагается самостоятельно создать код их обработки.

Связывание строк с объектами одинакового типа

Класс `TStringList` — очень удобный контейнер для хранения списков строк. Он обладает дополнительной возможностью хранения указателя на другой объект типа `TObject` или одного из его наследников. Код инициализации и удаления этих объектов следует создать вручную, но после присвоения указателя свойству `Objects` класса `TStringList` при добавлении, удалении или вставке строк управление им выполняется автоматически. Так как этот класс создает объекты для каждого члена, он также ответственен за освобождение памяти, связанной с каждым из этих объектов, при его удалении. Неудачная попытка выполнения этой операции приведет к утечке памяти.

Рассмотрим класс `TDBList`, который является наследником класса `TStringList`. Начнем с изучения фрагмента кода заголовочного файла для класса `TDBList`, который показан в листинге 8.12.

Листинг 8.12. Объявление класса `TDBList` в файле `dblist.h`

```
#include <dbtables.hpp>

enum DBListType {ltSession, ltAlias, ltTable, ltField};

class TDBList;

class TDBList : public TStringList
{
private:
   TTable *TempTable;
   TDBList *FListParent;
   DBListType FListType;
   AnsiString FParentItemName;

   void __fastcall GetSessions(void);
   void __fastcall GetAliases(void);
   void __fastcall GetTables(void);
   void __fastcall GetFields(void);
```

```
public:
   void __fastcall DeleteObjects(void);
   void __fastcall EnumDBList(void);
   virtual void __fastcall Clear(void);
   __fastcall TDBList(TDBList *pListParent,
      DBListType pListType, AnsiString pParentItemName);
   __fastcall ~TDBList(void);
   __property TDBList *ListParent = {read = FListParent};
   __property DBListType ListType = {read = FListType};
   __property AnsiString ParentItemName =
      {read = FParentItemName};
};
```

Класс `TDBList` создан для использования экземпляра того же класса в свойстве `Objects`. Благодаря такому подходу исключается необходимость создания пользовательских классов для каждого сеанса, псевдонима, таблицы и поля. Для представления ранее упомянутых типов объектов можно было бы создать базовый класс и его наследников, но разумнее воспользоваться следующим способом.

При создании каждого объекта необходим указатель на объект-владелец, который создал его, для извлечения информации о нем. Это достигается за счет создания свойства типа `TDBList*`. Такое свойство используется только для чтения и называется `ListParent`.

Объекту также необходим флаг для указания типа представляемого объекта. Это можно сделать с помощью перечислимого типа `DBListType` и свойства `ListType`. Это свойство указывает вид выполняемых задач, включая извлечение логически связанной информации и сохранение ее в свойстве `Objects`.

Наконец, этому классу также потребуется знать *имя элемента* родительского класса, которое содержит имя текущего сеанса, псевдонима или таблицы. По мере достижения уровня имени поля можно с помощью свойства `ListParent` постепенно извлечь имена элементов всех родительских классов. Этот процесс станет более понятным при подробном рассмотрении кода.

Чтобы класс содержал свойство такого же типа, необходимо предварительно объявить (forward declaration) этот класс. Предварительное объявление располагается непосредственно перед классом `TDBlist`. Хотя методы `GetSessions()`, `GetAliases()`, `GetTables()` и `GetFields()` понятны без лишних объяснений, мы все же рассмотрим их при подробном изучении кода.

Основная часть кода сосредоточена именно в методе `EnumDBList`, а метод `DeleteObjects` используется для освобождения предварительно выделенной памяти для свойства `Objects`. Эти две области класса имеют наибольшее значение и будут рассмотрены ниже более подробно.

Вот так выглядит заголовочный файл. Он имеет очень впечатляющий вид, если учесть, что он применяется для работы с достаточно большим объемом информации. Приступим теперь к рассмотрению исходного кода. Начнем с конструктора и постепенно перейдем к рассмотрению тех задач, которые выполняет класс.

В этом месте может возникнуть вопрос: а какое отношение все это имеет к классу `TStringList`? Так вот, мы создали класс-наследник класса `TStringList` и каркас объекта, который может самостоятельно наполняться данными. По мере заполнения объекта данными он может извлекать список имен (строки) и создавать другой объект `TDBList` (который также способен самостоятельно наполняться данными). Этот объект связан со свойством `Objects`.

Рассмотрим теперь способ создания такого специализированного класса. Обсуждение этого вопроса будет закончено демонстрацией универсальности этого класса благодаря переопределению метода сортировки класса `TStringList`. Эта модификация отсортирует все строки списка, а затем продолжит сортировку всех строк во всех связанных объектах `Objects`.

При первой инициализации класса конструктор задаст значения всех закрытых переменных, а затем перечислит элементы списка после выполнения некоторых простых проверок на основе только что созданного списка. Код этого конструктора приведен на рис. 8.13.

Листинг 8.13. Конструктор класса `TDBList`

```
__fastcall TDBList::TDBList(TDBList *pListParent,
                            DBListType pListType,
                            AnsiString pParentItemName)
                  : FListParent(pListParent),
                    FListType(pListType),
                    FParentItemName(pParentItemName),
                    TStringList()
{
   TempTable = 0;

   switch(FListType)
   {
      case ltSession:
         FParentItemName = ""; // не применяется для ltSession
         FListParent = 0;      // сеанс без родителя (создателя)
         break;                // установка значения NULL
                               // только для проверки!
      case ltAlias:
         if (FParentItemName == "")
         FParentItemName = "Default";
         break;
      case ltTable:
         break;                // без обработки имен таблиц
      case ltField:
         TempTable = new TTable(NULL);
         break;
   }

   EnumDBList();

   if (TempTable)
      delete TempTable;
}
```

Проверка нужна для устранения возможных ошибок при передаче классу неверных или ненужных значений параметров со стороны пользователей. Если пользователь создает список `TDBList` с информацией о сеансе, то для свойства `ParentItemName` задается пустая строка, а для свойства `ListParent` — значение `NULL`. Если пользователь передаст для этих параметров какие-то конкретные параметры, то никакого побочного эффекта не возникнет. Но все-таки для обеспечения безопасности следует создать специальный проверочный код, поскольку нельзя предугадать, как будет расширен этот класс в будущем. Можно, например, создать метод, который будет просматривать список в обратном порядке, пока не достигнет самого высокого уровня. Простейший способ определения самого верхнего элемента заключается в проверке равенства свойства `ListParent` значению `NULL`. Добавление такого кода гарантирует отсутствие исключительных ситуаций из-за ошибки пользователя.

Если пользователь создаст список псевдонимов и свойство ParentItemName (имя сеанса) содержит пустую строку, ее значение будет заменено строкой "Default". Это позволяет методу GetAliases() корректно функционировать в отсутствие имени сеанса ParentItemName за счет использования сеанса по умолчанию.

На этом этапе создания класса не проверяется создание пользователем списка таблиц. Подробнее эта часть кода будет рассмотрена при изучении метода GetTables().

Если пользователь создает список полей, то для этого потребуется только инициализировать новый объект TTable. Эта часть кода рассматривается при описании метода GetFields().

Наконец, в объекте TDBList с помощью метода EnumDBList() перечисляются элементы списка. Если бы объект TTable создавался после возвращения результата метода EnumDBList, его следовало бы удалить, т.к. он больше не нужен.

На заметку

Объект TTable нужен для перечисления имен полей. Для создания этого объекта могут использоваться два подхода. Использованный здесь метод создает объект TTable в случае необходимости и после использования удаляет его. Недостатком этого подхода является замедление процесса работы с объектом TDBList при большом количестве перечисляемых таблиц. Альтернативный подход основан на создании объекта TTable при создании основного объекта TDBList. Затем можно проверить существование списка ParentList во всех последующих объектах TDBList и, если он существует, то продолжить поиск родительского объекта TDBList, который создал объект TTable. Этот альтернативный подход снова рассматривается в конце этого раздела, а его код приводится на прилагаемом компакт-диске.

Рассмотрим теперь метод EnumDBList(), который показан в листинге 8.14.

Листинг 8.14. Метод EnumDBList() класса TDBList

```
void __fastcall TDBList::EnumDBList(void)
{
   // Этот метод собирает все имена объектного типа
   // (сеанс, псевдоним, таблица, поле), а затем
   // организует связь с другим объектным типом.

   // Удаление текущего списка и его элементов.
   if (Count)
      DeleteObjects();

   // Получение списка объектов и их имен.
   switch(ListType)
   {
      case ltSession : GetSessions();
                       break;

      case ltAlias : GetAliases();
                     break;

      case ltTable : GetTables();
                     break;
```

```
      case ltField : GetFields();
                     break;
   }
}
```

Код листинга 8.14 достаточно прост и понятен без излишних пояснений. Сначала проверяется наличие данных в списке. В этом случае список удаляется с помощью метода `DeleteObjects()`, который показан в листинге 8.15.

Листинг 8.15. Метод `DeleteObjects()` класса `TDBList`

```
void __fastcall TDBList::DeleteObjects(void)
{
   for(int i = 0; i < Count; i++)
   {
      if (Objects[i])
      {
         delete Objects[i];
         Objects[i] == 0;
      }
   }
   Clear ();
}
```

Метод `DeleteObjects()` итеративно перебирает все элементы списка. Если текущая строка имеет связанный с ней объект, то его область памяти освобождается, а для его указателя задается значение `NULL`. Метод `DeleteObjects()` вызывается деструктором класса `TDBList` (см. листинг 8.15). Это гарантирует автоматическое освобождение памяти для всех связанных с ним объектов. В конце работы цикла список строк очищается. В результате все связанные объекты освобождаются (т.е. удаляются), а текущий объект `TDBList` становится пустым.

Вернемся к методу `EnumDBList()`, код которого приведен в листинге 8.14. Этот метод определяет типы элементов, которые требуется перечислить с помощью `ListType`, и вызывает соответствующий метод.

Методы `GetSessions()`, `GetAliases()`, `GetTables()` и `GetFields()` показаны в листинге 8.16 с кратким описанием их назначения. Все вопросы, связанные с взаимодействием с базой данных, будут рассмотрены лишь в той мере, в которой они необходимы для обсуждения темы этого раздела.

Листинг 8.16. Извлечение информации о сеансах, псевдонимах, таблицах и полях

```
void __fastcall TDBList::GetSessions(void)
{
   try
   {
      Sessions->GetSessionNames(this);
      // Теперь для каждого имени сеанса
      // нужно собрать все его псевдонимы
     for (int i = 0; i < Count; i++)
     {
        TDBList *AliasList =
           new TDBList(this, ltAlias, Strings[i]);
```

```
        Objects[i] == dynamic_cast<TObject *>(AliasList);
      }
   }
   catch(...)
   {
      DeleteObjects();
   }
}

void __fastcall TDBList::GetAliases(void)
{
   AnsiString SessionName = ParentItemName;

   try
   {
      Sessions->List[SessionName]->GetAliasNames(this);

      for(int i = 0; i < Count; i++)
      {
         TDBList *TableList =
            new TDBList(this, ltTable, Strings[i]);
         Objects[i] == dynamic_cast<TObject *>(TableList);
      }
   }

   catch(Exception &e)
   {
      DeleteObjects();
   }
}

void __fastcall TDBList::GetTables(void)
{
   AnsiString AliasName = ParentItemName;
   AnsiString SessionName("Default");

   if(ListParent)
      SessionName = ListParent->ParentItemName;
   else
      SessionName = "Default";
   try
   {
      Sessions->List[SessionName]->GetTableNames(AliasName,
                                                 "", true,
                                                 false, this);

      for(int i = 0; i < Count; i++)
      {
         try
         {
            TDBList *FieldList =
```

```
                new TDBList(this, ltField, Strings[i]);
             Objects[i] == dynamic_cast<TObject *>(FieldList);
          }
          catch(Exception &e)
          {
             DeleteObjects();
          }
       }
    }
    catch(...)
    {
       DeleteObjects();
    }
}

void __fastcall TDBList::GetFields(void)
{
    AnsiString SessionName("Default");
    AnsiString AliasName;
    AnsiString TableName = ParentItemName;

    TDBList *pList = ListParent;

    if(pList)
    {
       AliasName = pList->ParentItemName;
       pList = pList->ListParent;
       if(pList)
          SessionName = pList->ParentItemName;;
    }

    try
    {
       TempTable->Active = false;
       TempTable->SessionName = SessionName;
       TempTable->DatabaseName = AliasName;
       TempTable->TableType = ttDefault;
       TempTable->TableName = TableName;
       TempTable->Active = true;

       try
       {
          TempTable->GetFieldNames(this);

          for(int i = 0; i < Count; i++)
             Objects[i] == 0;
       }
       catch(Exception &e)
       {
          DeleteObjects();
       }
```

```
   }
   catch(Exception &e)
   {
      DeleteObjects ();
   }
}
```

Метод GetSessions()

Метод `GetSessions()` извлекает список всех доступных имен сеансов с помощью команды `Sessions->GetSessionNames(this)`. Передача указателя `this` методу `GetSessionNames()` возвращает имена сеансов в наш список, т.е. класс `TDBList`. Это возможно потому, что класс `TDBList` является производным от класса `TStringList` и, следовательно, обладает теми же свойствами и методами, что и класс `TStringList`.

Следующей (и наиболее важной) частью этого метода является циклический перебор всех элементов списка, который показан в приведенном ниже коде.

```
for(int i = 0; i < Count; i++)
{
   TDBList *AliasList = new TDBList(this, ltAlias, Strings[i]);
   Objects[i] == dynamic_cast<TObject *>(AliasList);
}
```

Этот код создает новый объект `TDBList` для перечисления всех доступных для сеанса псевдонимов, которые упоминаются в массиве `Strings[i]`. Возвращаясь к коду конструктора в листинге 8.13, можно заметить, что для каждого нового объекта вызывается метод `EnumDBList()`; это приводит к вызову метода `GetAliases()`.

Каждый из новых объектов информируется о том, что это список `ParentList` — это гарантирует постоянную связь между всеми созданными объектами `TDBList`.

На заметку

Надежность кода имеет очень большое значение для таких простых и мощных объектов, как `TDBList`. Представленный здесь код можно было бы записать, как показано ниже (потому что указатель `AliasList` является только временным указателем на вновь созданный объект `TDBList`), но он не очень прост для восприятия. Автор иногда создавал код в показанном выше виде, а затем возвращался обратно для устранения ошибок и внесения необходимых изменений. Для этого очень важно тщательно документировать код, показывая прежний и текущий варианты, чтобы впоследствии было проще вносить в него изменения.

Исходная версия:

```
for(int i = 0; i < Count; i++)
{
   TDBList *AliasList = new TDBList(this, ltAlias, Strings[i]);
   Objects[i] == dynamic_cast<TObject *>(AliasList);
}
```

Альтернативный вариант:

```
for(int i = 0; i < Count; i++)
{
   Objects[i] == dynamic_cast<TObject *>(TDBList(this, ltAlias,
                                                 Strings[i]));
}
```

Если в процессе перечисления возникнет исключительная ситуация, она будет перехвачена, а все объекты, созданные для элемента, будут удалены. Это может произойти, например, при попытке подключения с использованием псевдонима, но пользователь не может ввести правильные параметры регистрации. Теоретически никакие объекты не должны создаваться, но такой код все же следует создать для тех случаев, когда сетевое подключение может быть прервано в процессе перечисления.

Метод GetAliases()

Метод GetAliases() сначала определяет имя сеанса с помощью свойства ParentItemName. Если родительского объекта нет, конструктор задает для свойства ParentItemName значение "Default". С помощью имени сеанса SessionName метод находит список всех псевдонимов с помощью команды Sessions->List [SessionName]->GetAliasNames(this). Библиотека VCL имеет глобальную переменную Sessions, которая содержит список доступных сеансов. Переменная SessionName используется для хранения указателя на заданный сеанс, а затем вызывает метод GetAliasNames(), передавая указатель this на список строк, который следует наполнить данными.

Этот метод используется для создания новых объектов TDBList, но в данном случае — с целью хранения имен таблиц для каждого найденного псевдонима. Здесь снова вызывается конструктор для каждого нового объекта EnumDBList(), что приводит к вызову метода GetTables.

При наличии родительского списка ParentList мы получим список сеансов, а для каждого элемента — связанный список псевдонимов. Теперь для каждого псевдонима создан список его таблиц.

Метод GetTables()

Метод GetTables() начинает работу с определения параметров сеанса и псевдонима, задавая соответствующие значения по умолчанию. Обратите внимание, как имя сеанса извлекается с помощью элемента ParentItemName свойства ListParent. Свойство ListParent, если оно существует, является владельцем текущего объекта. Оно должно содержать список псевдонимов, а элемент ParentItemName псевдонима должен содержать имя сеанса для этого псевдонима.

Используя заданные имена сеанса и псевдонима, можно извлечь список всех доступных таблиц. Параметры метода GetTableNames() очень подробно описаны в интерактивной справке. Перед их изменением рекомендуется внимательно прочитать соответствующие разделы справочного руководства.

Для каждой строки списка следует создать дополнительный объект TDBList для хранения списка с именами полей. Каждый из этих объектов связан со свойством Objects, что позволяет продолжить список связанных объектов.

Метод GetFields()

Метод GetFields() является заключительным звеном этой цепочки. Сначала он определяет имя таблицы, псевдонима и сеанса (используя свойства ParentItemName и ListParent). Конструктор для каждого объекта списка полей создает временный объект TTable, который используется для перечисления имен полей для данной таблицы. (Позже мы рассмотрим методы оптимизации этого кода.)

Сразу после создания объекта TTable можно приступать к извлечению списка имен полей, а затем указать значение NULL для свойства Object каждого объекта. Хотя эти действия выполняются по умолчанию, все же рекомендуется сделать это для повышения читабельности кода (на основе предполагаемой функциональности).

Создание цепочки событий

В начале описания этого объекта мы рассмотрели способ вызова метода GetSessions() методом EnumDBList(). Метод, в свою очередь, вызывает метод GetSessions(), который вызывает метод GetAliases(), который в свою очередь вызывает метод GetTables(), а он вызывает метод GetFields(). В результате этого процесса получится целый список сеансов и связанных с ними псевдонимов, таблиц и полей. Совсем неплохой результат, который получен в процессе создания всего лишь одного объекта.

Проект TDBListDemo.bpr на прилагаемом компакт-диске содержит пример исследования этого объекта для получения содержащейся в нем информации. Эта информация содержится в элементе управления TMemo. Поскольку структура класса TDBList использует связанные объекты одного типа, это демонстрационное приложение для извлечения информации использует рекурсивный процесс.

Сначала создается один объект TDBList, затем с помощью переопределенного метода сортировки сортируются все его элементы, после чего вызывается метод PopulateMemo() нашей формы. При этом учитывается тип элементов, которые содержатся в переданном списке. Код этого метода показан в листинге 8.17.

Листинг 8.17. Перечисление и отображения информации

```
void __fastcall TMainForm::StartButtonClick(TObject *Sender)
{
   TDBList *TheDBList = new TDBList(0, ltSession, "");

   if(TheDBList)
   {
      TheDBList->Sort(); // Сортирует все уровни
      PopulateMemo(TheDBList);
      delete TheDBList;
   }
}
```

Метод PopulateMemo() в листинге 8.18 форматирует строку с учетом ее типа, а также определяет наличие связанного с ней объекта. Если такой объект существует, его тип явно преобразуется к типу TDBList* и снова передается методу PopulateMemo(). Этот процесс прекратится после достижения списка с именами полей, потому что для него свойство Objects имеет значение NULL.

Листинг 8.18. Рекурсивное форматирование строки

```
void __fastcall TMainForm::PopulateMemo(TDBList *TheList)
{
   TDBList *NextObject;

   // Выполняется до передачи нулевого указателя
   for(int i = 0; TheList && i < TheList->Count; i++)
   {
      switch(TheList->ListType)
      {
         case ltSession : Tabs = "";
```

```
                        break;
         case ltAlias : Tabs = "\t";
                        break;
         case ltTable : Tabs = "\t\t";
                        break;
         case ltField : Tabs = "\t\t\t";
                        break;
      }

      Memo1->Lines->Add(Tabs +TheList->Strings [i ]);

      if(TheList->Objects [i ])
      {
         NextObject =
            dynamic_cast<TDBList *>(TheList->Objects[i]);
         PopulateMemo (NextObject);
      }
   }
}
```

Это еще один пример использования объектно-ориентированного подхода для создания простого и эффективного кода.

Сортировка списков

Объект `TStringList` имеет метод `Sort()`, который наследуется классом `TDBList`. Единственная проблема заключается в том, что при сортировке списка созданных в объекте строк он будет сортировать только списки этого объекта. Что же нужно сделать, чтобы сортировались также строки во всех связанных списках? Эту задачу очень просто решить с помощью того же метода, который используется при создании объекта `TDBList`.

Метод `Sort()` объявляется таким же, как и в классе `TStringList`. А затем это метод переопределяется, как показано в листинге 8.19.

Листинг 8.19. Переопределение метода сортировки `Sort()` класса `TStringList`

```
void __fastcall TDBList::Sort(void)
{
   TStringList::Sort();

   TDBList *NextList;
   for(int i = 0; i < Count; i++)
   {
      if(Objects[i])
      {
         NextList = dynamic_cast<TDBList *>(Objects[i]);
         NextList->Sort(); // Сортировка всех уровней
      }
   }
}
```

Код этого метода начинается с вызова унаследованного метода `Sort()`. Это приводит к сортировке всех строк текущего объекта. При этом автоматически удаляются все указатели, поддерживаемые с помощью свойства `Objects`. Затем проверяется наличие определенного указателя для каждого свойства `Objects[i]`. Далее свойству `Objects` явным образом присваивается тип `TDBList*`, а потом вызывается метод `Sort()` для этого объекта. При этом работа будет продолжена вплоть до уровня поля и здесь прекращена, потому что все свойства `Objects` имеют значения `NULL`.

Некоторые усовершенствования

При использовании этого кода в рабочем приложении следовало бы незначительно усовершенствовать его.

- Следует добавить параметр в код конструктора для указания необходимости автоматического заполнения данными. Если такой параметр не задан, то по умолчанию заполнение данными происходит автоматически.
- Следует модифицировать объект для создания одного объекта `TTable` для всего процесса в целом. Обратный поиск к основному объекту (с помощью указателей) выполняется гораздо быстрее, чем создание таблицы `TTable` каждый раз при необходимости перечисления списка имен полей.
- Методы `Add()`, `Clear()`, `Delete()` и `Insert()` следует переопределить для автоматического наполнения (или удаления) данными полностью связанного списка элементов.

Более подробные сведения об этих усовершенствованиях можно получить, изучив полную версию проекта `TDBListDemo.bpr` на прилагаемом компакт-диске. Ниже приводится несколько замечаний в отношении изменений, которые внесены в этот код.

Переопределенный метод `Clear()` теперь гарантирует, что любые связанные объекты удаляются до очистки списка. Это позволяет предотвратить утечку памяти.

Метод `EnumDBList()` теперь вызывает метод `Clear()` вместо метода `DeleteObjects()`, потому что по сути он выполняет задачи метода `DeleteObjects()`.

Метод `DeleteObjects` был модифицирован только для удаления связанных объектов. Теперь он не выполняет очистку списка. Дело в том, что ранее он удалял список строк, что не нужно делать в данном случае. Гораздо важнее то, что вызов метода `Clear()` приведет к циклическому выполнению методов `Clear()` и `DeleteObjects()`, т.е. к переполнению стека. Для удаления объектов следует вызвать метод `DeleteObjects()`. Если нужно очистить список строк и связанных объектов, то вызвается метод `Clear()`.

Наибольшее усовершенствование заключается в переопределении метода `Add()`. Если пользователь хочет создать новую строку в списке, список будет рассматривать ее так же, как и все другие строки `ListType`, а значит, будет искать связанную с ней информацию. Иначе говоря, если пользователь добавляет строку в список с именами псевдонимов, то такой список должен найти связанные с ней имена таблиц и полей. Эта задача выполняется с помощью метода `Add()`. Интересно отметь, что методы `GetSessionNames()`, `GetAliasNames()`, `GetTableNames()` и `GetFieldNames()` используют метод `Add()` для создания списка строк. Эта особенность, несомненно, является преимуществом этого способа, поэтому всякий раз при создании нового элемента списка методы класса автоматически найдут всю связанную с этим элементом информацию. Теперь можно удалить все циклы в методах `GetSessions()`, `GetAliases()` и `GetTables()`. В противном случае они просто продублируют ту работу, которую уже выполнил метод `Add()`. После создания строки вызывается новый метод

CreateObject(). Этот метод используется для назначения нового объекта для вновь добавленной в список строки.

Метод Insert() также переопределен только за счет дополнительного вызова метода CreateObject(). Как и в случае метода Add(), это позволяет назначить новый объект TDBList для вновь добавленной в список строки.

Метод удаления текущего элемента Delete()переопределяется для предварительного удаления всех связанных с ним объектов.

Этот класс усовершенствован для создания единственного экземпляра объекта TTable. Для поиска этой таблицы необходимо добавить свойство MasterTable, которое отвечает за обратный поиск среди связанных объектов первого инициализированного объекта TTable.

Усовершенствованные пользовательские события рисования

Пользовательские события рисования используются в элементах управления TTreeView, TListView и TToolBar. Они позволяют управлять рисованием канвы элемента управления или его подчиненных элементов. Задавая для свойства DefaultDraw значение false, разработчик может самостоятельно рисовать содержимое канвы.

В C++Builder 5 вводятся усовершенствованные пользовательские события рисования, которые позволяют программисту управлять процессом рисования компонентов. Все новые события содержат в названиях слово Advanced.

Рассмотрим эти компоненты и связанные с ними пользовательские события рисования.

Компонент TTreeView

Компонент TTreeView имеет четыре пользовательских события рисования, которые перечислены в табл. 8.2.

Таблица 8.2. Пользовательские события рисования иерархического представления

События	Описание
OnCustomDraw	Возникает до рисования элемента управления и используется для рисования на нем изображения
OnCustomDrawItem	Возникает до рисования элемента управления и используется для рисования на нем отдельных элементов
OnAdvancedCustomDraw	Возникает аналогично событию OnCustomDraw за исключением того, что оно возникает в ходе всего процесса рисования
OnAdvancedCustomDrawItem	Возникает аналогично событию OnAdvanced-CustomDraw, но используется для рисования отдельных элементов

Перечисленные в табл. 8.2 элементы характерны для компонента TTreeView, но как будет показано ниже, компоненты TListView и TToolButton имеют аналогичные события.

Каждый процесс рисования состоит из четырех этапов:

1. cdPrePaint (перед рисованием);
2. cdPostPaint (после рисования);

3. `cdPreErase` (перед стиранием);
4. `cdPostErase` (после стирания).

Обратите внимание, что события `OnCustomDraw` и `OnCustomDrawItem` возникают только перед процессом рисования, тогда как усовершенствованные события возникают на всех стадиях рисования.

Событие `OnAdvancedCustomDrawItem` элемента управления `TTreeView` обладает еще одним преимуществом по сравнению с событием `OnCustomDrawItem`. Оно позволяет включать или отключать рисование изображения, связанного с текущим узлом. Эта возможность предусмотрена только для компонента `TTreeView`.

Компонент `TListView`

Пользовательские события рисования компонента `TListView` аналогичны событиям компонента `TTreeView`. Компонент `TListView` также имеет два дополнительных пользовательских события рисования: `OnCustomDrawSubItem` и `OnAdvancedCustomDrawSubItem`. Они используются для рисования подчиненных элементов компонента `TListView`, которые отображаются, если для свойства `ViewStyle` компонента `TListView` задано значение `vsReport`.

Компонент `TToolBar`

События компонента `TToolBar` аналогичны событиям компонента `TTreeView` за исключением того, что он имеет события `OnCustomDrawButton` и `OnAdvancedCustomDrawButton` вместо событий `OnCustomDrawItem` и `OnAdvancedCustomDrawItem`, соответственно. Кроме того, параметр `TTreeNode` заменен параметром `TToolBar`.

Пример использования пользовательских событий рисования

В каталоге `CustomDraw` на прилагаемом компакт-диске содержится проект `CustomDraw.bpr`, где демонстрируются способы использования некоторых событий, описанные в предыдущих разделах. Он демонстрирует использование события `OnCustomDraw` компонента `TTreeView` для рисования пользовательского изображения в канве иерархического представления, а также элементов управления каждого элемента этого представления с помощью события `OnAdvancedCustomDrawItem`. Для знакомства с основными принципами работы с этим событием следует внимательно изучить код в файле `Unit1.cpp`.

Программа-мастер создания компонентов панели управления *Control Panel Applet Wizard*

В Windows 3.x компоненты панели управления использовались в основном для конфигурирования аппаратного обеспечения. В операционных системах Windows 95/98/2000 они могут использоваться для конфигурирования не только аппаратного, но и программного обеспечения, а также приложений.

Они могут конфигурировать приложение извне без обязательной загрузки этого приложения для изменения его конфигурационных параметров, например, с помощью команд меню.

С помощью компонентов панели управления можно конфигурировать как окружение, так и поведение приложения. После этого приложение может быть запущено в обычном режиме. Это позволяет удалить конфигурационный код из основного приложения и задать параметры конфигурации для нескольких приложений одновременно.

В C++Builder 5 предусмотрена программа-мастер для создания компонентов панели управления. С ее помощью можно очень просто создавать пиктограммы этих компонентов, справочных файлов, а также форм и параметров каждого компонента.

Основные принципы работы с компонентами панели управления

Компонент панели управления представляет собой небольшую утилиту, которая предназначена для управления поведением аппаратного или программного обеспечения. Эта утилита может задавать значения переменных окружения перед загрузкой приложения, что позволяет управлять поведением приложения. Такие утилиты обычно имеют небольшой размер и используются только для этих целей. На рис. 8.4 показан их внешний вид в панели управления.

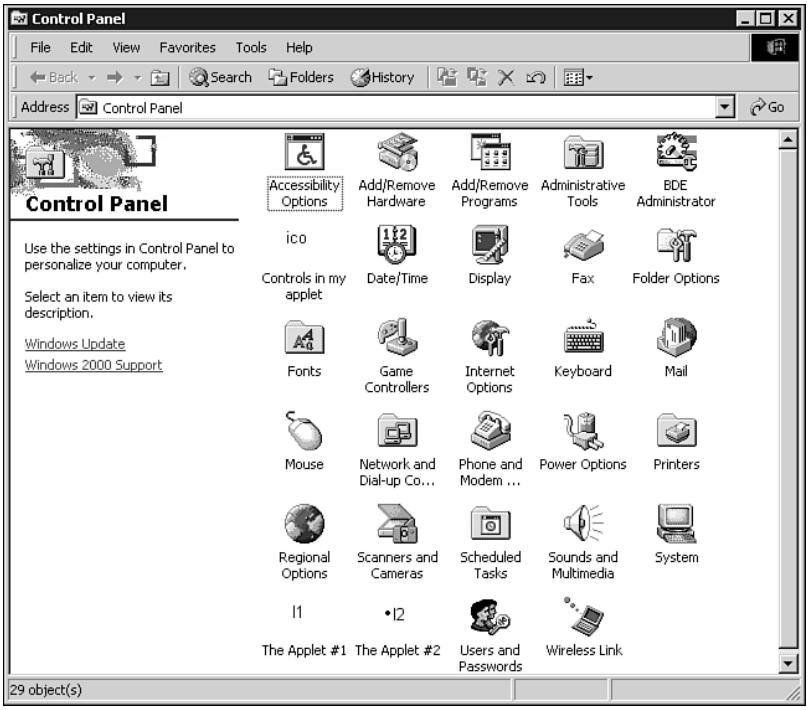


Рис. 8.4. Компоненты панели управления в операционной системе Windows 2000

Фактически компонент панели управления представляет собой DLL-библиотеку с расширением `.cpl`. Управляющее приложение считывает файл с расширением `.cpl` и извлекает из него нужную информацию. DLL-библиотека содержит специальную функцию с точкой входа, `CPLApplet()`, которая вызывается для запуска компонента. Управляющим приложением обычно является сама панель управления (обычно это файл `CONTROL.EXE`), но на самом деле

любое приложение может управлять работой компонентов панели управления. Для этого приложение должно управлять сообщениями и работой компонента.

Компоненты панели управления могут содержать другие компоненты. Например, можно создать приложение с двумя формами конфигурирования двух разных приложений с двумя разными пиктограммами. Например, компоненты System и Add/Remove Hardware, которые входят в состав операционной системы Windows 95/98, запускаются с помощью компонента SYSDM.CPL из системного каталога Windows. Они выполняют разные функции и представлены отдельными пиктограммами. Хотя они имеют разные формы и выглядят как различные программы, они управляются одним компонентом панели управления.

Компоненты панели управления вызываются управляющим приложением, которое посылает сообщения нужному компоненту, а оно, в свою очередь, возвращает нужные значения управляющему приложению. Именно эти сообщения затем используются для управления поведением компонента панели управления. Возвращаемые значения зависят от параметров, переданных компоненту из управляющего приложения. Код компонента панели управления должен создаваться в строго определенном прядке, потому что посылаемые ему сообщения имеют строго заданный порядок. В табл. 8.3 показан порядок этих сообщений и значений.

Таблица 8.3. Порядок сообщений и способ их действия

Порядок	Действие
Сообщение: CPL_INIT (#define константа = 1) Вызывается первым сразу после загрузки управляющим приложением CPL-файла	Это сообщение посылается для указания того, что функция CPlApplet() найдена. Параметры lParam1 и lParam2 не используются. Возвращаются значения true или false, которые обозначают продолжение или прекращение работы с панелью управления. В случае неудачи прекращается управление компонентом и обмен данными с CPL-файлом
Сообщение: CPL_GETCOUNT (#define константа = 2) Вызывается после сообщения CPL_INIT и возвращает ненулевое значение	Это сообщение посылается для определения количества отображаемых компонентов. Оно возвращает ненулевое значение, а именно количество компонентов, которые нужно отобразить в панели управления. Параметры lParam1 и lParam2 еще не используются
Сообщение: CPL_INQUIRE (#define константа = 3) Вызывается после сообщения CPL_GETCOUNT и возвращает значение не меньше 1. Функция CPLApplet() вызывается один раз для каждого диалогового окна с указанием индекса каждого из них в параметре lParam1	Это сообщение посылается для указания каждого компонента. Функция CPLApplet() предоставляет информацию о диалоговом окне с помощью параметра lParam1. Параметр lParam2 указывает на структуру CPLINFO, которая содержит имя и пиктограмму
Сообщение: CPL_NEWINQUIRE (#define константа = 8) Оно применяется, если не используется сообщение CPL_INQUIRE. Оно используется в том же порядке и с тем же результатом, что и событие CPL_INQUIRE	Его применение приводит к тому же результату, что и использование сообщения CPL_INQUIRE, за исключением того, что параметр lParam2 содержит указатель на структуру NEWCPLINFO с новыми данными самого CPL-компонента. Для достижения более высокой производительности в операционных системах Win 95/NT используется сообщение CPL_INQUIRE

Окончание табл. 8.3

Порядок	Действие
Сообщение: `CPL_DBLCLK (#define константа = 5)` Вызывается после двойного щелчка мышью на пиктограмме компонента	Это сообщение посылается после двойного щелчка мышью на пиктограмме компонента. Параметр `lParam1` содержит номер выбранного компонента, а параметр `lParam2` — значение переменной `lData`. Это сообщение инициирует открытие диалогового окна компонента и формы заданного компонента
Сообщение: `CPL_STOP (#define константа = 6)` Вызывается только один раз для каждого диалогового окна (индекс которого задается значением параметра `lParam1`) перед закрытием приложения	Функция `CPLApplet()` попытается освободить все ресурсы, выделенные для этого диалогового окна. Это сообщение посылается каждому компоненту при закрытии панели управления. Параметр `lParam1` содержит номер компонента, а параметр `lParam2` — значение переменной `lData`. Все операции очистки содержимого компонента выполняются именно на этом этапе
Сообщение: `CPL_EXIT (#define константа = 7)` Это сообщение посылается сразу после последнего сообщения `CPL_STOP`, посланного индексированными диалоговыми окнами. Оно вызывается после использования управляющим приложением функции `FreeLibrary()` для освобождения CPL-файла, содержащего приложение	Это сообщение посылается непосредственно перед вызовом функции `FreeLibrary()` панели управления. Параметры `lParam1` и `lParam2` здесь не используются. При этом выполняются все специфические для компонента операции очистки: освобождение ресурсов, освобождение памяти и т.п.

Не пугайтесь: в C++Builder обработка этих сообщений выполняется в событиях компонентов. С помощью новой программы-мастера C++Builder создание компонента существенно упрощается. В главе 24, посвященной использованию Win32 API-функций, подробно рассматриваются способы создания компонентов панели управления без использования программы-мастера CPL Wizard.

В этом разделе рассматривается способ создания простого компонента с помощью программы-мастера ***Control Panel Applet Wizard***, предназначенного для создания компонентов в среде C++Builder.

На заметку

В этой главе не рассматриваются способы создания пиктограммы компонента. Предполагается, что читатель уже знаком с основными принципами создания ICO-файлов пиктограмм с помощью редактора изображений ***Image Editor***. Так как для данного примера нужен ICO-файл, то прежде чем продолжать работу, создайте для него пиктограмму или используйте уже готовую.

Запустите C++Builder и после его загрузки выберите команду меню File⇨Close All для закрытия всех окон в IDE-среде.

Затем выберите команду меню File⇨New для открытия диалогового окна New Items. После этого выберите пиктограмму компонента Application, которая показана на рис. 8.5. Обратите внимание, что рядом с ней расположена пиктограмма компонента Module, который используется для добавления модулей в компонент панели управления.

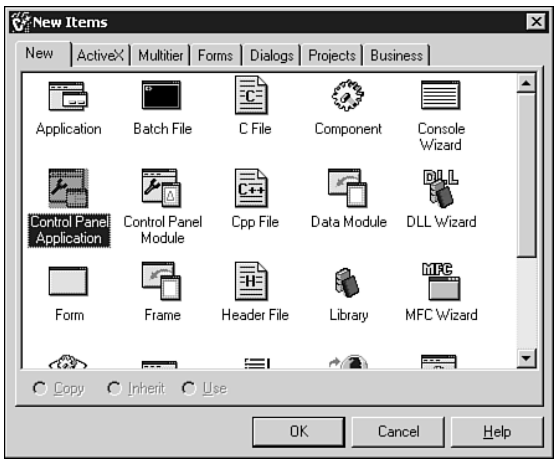


Рис. 8.5. Выбор программы-мастера создания компонентов панели управления **Control Panel Applet Wizard** *в среде C++Builder.*

После этого в среде C++Builder будет запущена программа-мастер создания компонентов панели управления ***Control Panel Applet Wizard***. Она приступит к сбору полезной информации и генерации кода для компонента панели управления. Затем появится окно, которое внешне напоминает диалоговое окно Data Module Designer, за исключением двух новых вкладок — Components и Data Diagram, которые показаны на рис. 8.6. Завершив процесс создания кода, C++Builder генерирует класс `TAppletModule` для компонента панели управления. Нажав клавишу <F12>, можно перейти к непосредственному редактированию этого кода, который был автоматически создан средой C++Builder.

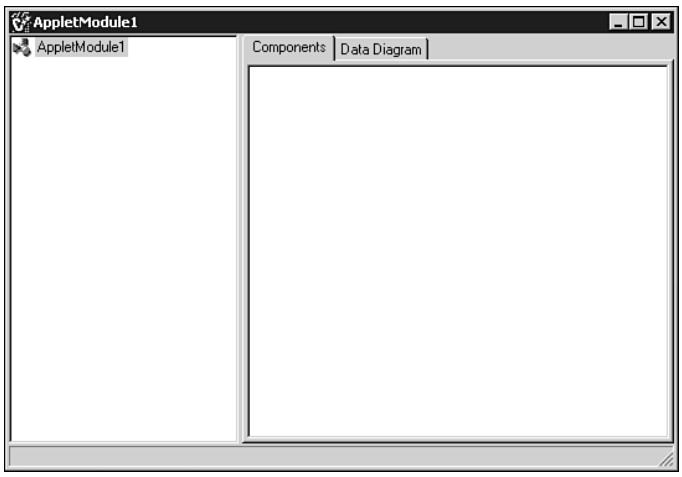


Рис. 8.6. Окно просмотра модулей, которое внешне напоминает диалоговое окно Data Module Designer, но содержит перечень модулей и компонентов

Это приложение происходит от модуля `CtlPanel` среды C++Builder и скрывает от программиста все рутинные операции по созданию компонента панели управления. Модуль `CtlPanel`

содержит три важных элемента панели управления: EAppletException, TAppletApplication и TAppletModule. Класс TAppletApplication является одноэлементным контейнером для хранения модуля TAppletModule, например только что созданного модуля. При необходимости в проект можно включить бо́льшее количество модулей TAppletModule. Например, для включения дополнительных модулей в наш компонент панели управления нужно выбрать команду меню File⇨New, а потом — пиктограмму Module. Это приведет к созданию другого модуля TAppletModule и вставке его в проект компонента панели управления.

Глобальная переменная приложения TAppletApplication объявлена в модуле CtlPanel и представляет собой приложение проекта компонента панели управления. При создании нового проекта компонента панели управления среда C++Builder создает объект-приложение и присваивает его глобальной переменной Application. Если проект представляет собой приложение панели управления, то глобальная переменная Application инициализируется глобальной переменной TAppletApplication. В результате один модуль TAppletModule содержит всю функциональную часть и инкапсулирует весь компонент панели управления.

Свойства модуля TAppletModule

Для управления компонентом панели управления в среде C++Builder необходимо задать значения некоторых свойств. Эти свойства могут определять справку компонента, его пиктограмму и имя. Свойства модуля TAppletModule в окне Object Inspector показаны на рис. 8.7.

Ниже приводится подробная характеристика свойств.

- Свойство AppletIcon содержит указатель на пиктограмму для данного модуля компонента панели управления. При включении другого модуля TAppletModule необходимо также включить пиктограмму этого модуля или компонента в данное приложение. Эти пиктограммы представляют собой ICO-файлы. При двойном щелчке на этом свойстве в среде C++Builder будет автоматически вызвано диалоговое окно Icon File Finder для поиска нужной пиктограммы с расширением ICO. Если для этого свойства задана соответствующая пиктограмма, окно Object Inspector автоматически свяжет ее с данным компонентом панели управления.

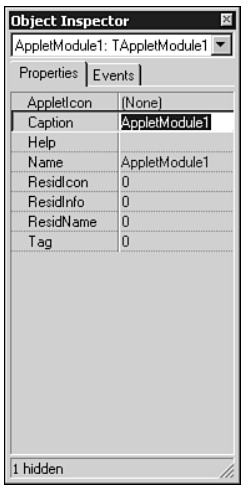


Рис. 8.7. Окно Object Inspector, в котором показаны свойства модуля AppletModule1

- Свойство Caption содержит строку, которая располагается под пиктограммой компонента в панели управления. Обычно она совпадает с заголовком самого компонента.
- Свойство Help содержит строку, которая располагается в строке состояния панели управления. Она обычно содержит описание самого компонента.
- Свойство ResidIcon содержит идентификатор ресурса для пиктограммы, которая связана с данным модулем компонента. Это свойство является взаимно исключающим для свойства AppletIcon.
- Свойство ResidInfo содержит идентификатор ресурса для справочной строки, которая связана с данным модулем компонента. Это свойство является взаимно исключающим для свойства Help.
- Свойство ResidName содержит идентификатор ресурса для строки заголовка, которая связана с данным модулем компонента. Это свойство является взаимно исключающим для свойства Caption.

Обычно при создании одного компонента панели управления с одним модулем, достаточно просто нужным образом задать значения свойств `AppletIcon`, `Caption` и `Help` и не трогать свойства `ResidIcon`, `ResidInfo` и `ResidName`.

События модуля `TAppletModule`

На рис. 8.8 показаны события модуля `TAppletModule`. Однако следует отметить, что для процесса создания компонентов панели управления не предусмотрена интерактивная справка, а для `TAppletApplication` и `CtlPanel` она очень ограничена. К счастью, некоторые справочные сведения можно получить при работе с событиями модуля `TAppletModule`. Для этого, находясь в поле свойства `Events`, нужно нажать комбинацию клавиш <Ctrl+F1>.

Рис. 8.8. Окно Object Inspector, в котором показаны свойства модуля `AppletModule1`

Большинство событий не понадобятся в данном примере. Как известно, события `OnCreate` и `OnDestroy` возникают при создании и удалении компонента панели управления. Эти события также возникают при открытии и закрытии панели управления. Напомним, что панель управления обменивается данными (заголовок, справка, пиктограмма и т.п.) с событиями компонента, даже если компонент еще не активизирован двойным щелчком мыши.

События `OnInquire`, `OnNewInquire`, `OnStop` и `OnStartWParams` не будут использованы в этом примере, но они понадобятся для обмена информацией с приложением.

В этом примере используется событие `OnActivate`, которое возникает после двойного щелчка на компоненте `Application` в панели управления. Событие `OnActivate` имеет аргумент `Sender`, который указывает на соответствующий активированный модуль. Оно также содержит аргумент `Data` для обработки событий `OnInquire` или `OnNewInquire`. Более подробную информацию о работе с этими событиями можно найти в главе 24 в разделе, посвященное созданию приложение панели управления традиционным способом.

GUI-интерфейс компонента панели управления

Рассмотрим способы создания графического интерфейса пользователя для компонента панели управления. Рассмотрим их на примере создания простого компонента с тремя кнопками.

Выберите модуль AppletModule1, а потом дважды щелкните на свойстве AppletIcon в окне Object Inspector для отображения на экране диалогового окна редактора изображений Picture Editor. Выберите подходящую пиктограмму или создайте ее сами с помощью редактора Image Editor и сохраните в файле с расширением .ICO.

Для свойства Caption задайте строку My Control Panel Applet (Мой компонент панели управления), а потом для свойства Help задайте строку This is an example of my applet (Это пример моего компонента).

Выберите команду меню File⇨New Form и на экране появится новая форма. Задайте для нее нужные значения ширины и высоты. Это можно сделать с помощью указателя мыши, либо задав значения свойств `Width` и `Height` в окне Object Inspector.

Выберите компонент `TButton` во вкладке Standard и перетащите его в форму. Задайте для свойств кнопки значения, показанные в табл. **8.4**.

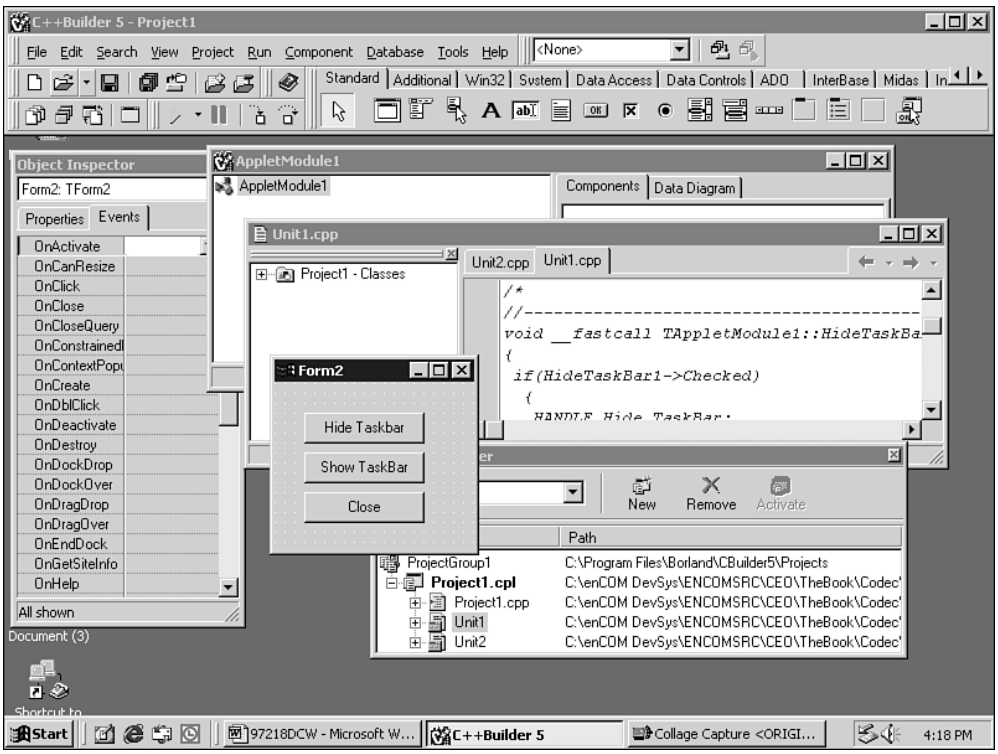


Рис. 8.9. Окно Object Inspector, в котором показаны события модуля `AppletModule1`

Таблица 8.4. Свойства кнопок компонента

Компонент	Свойство	Значение
Button1	Caption	Hide Taskbar
	Width	100
Button2	Caption	Show Taskbar
	Width	100
Button3	Caption	Close
	Width	100

Чтобы создать код обработчика события `OnClick` для кнопки `Button1`, дважды щелкните на ней левой кнопкой мыши и введите следующий код.

```
void __fastcall TForm2::Button1Click(TObject *Sender)
{
   HANDLE Hide_TaskBar;
   Hide_TaskBar = FindWindow("Shell_traywnd", "");
   SetWindowPos(Hide_TaskBar, 0, 0, 0, 0, 0, SWP_HIDEWINDOW);
}
```

Чтобы создать код обработчика события OnClick для кнопки Button2, дважды щелкните на ней левой кнопкой мыши и введите следующий код.

```
void __fastcall TForm2::Button2Click(TObject *Sender)
{
   HANDLE Hide_TaskBar;
   Hide_TaskBar = FindWindow("Shell_traywnd", "");
   SetWindowPos(Hide_TaskBar, 0, 0, 0, 0, 0, SWP_SHOWWINDOW);
}
```

Наконец, чтобы создать код обработчика события OnClick для кнопки Button3, дважды щелкните на ней левой кнопкой мыши и введите следующий код.

```
void __fastcall TForm2::Button3Click(TObject *Sender)
{
   Close();
}
```

Теперь нужно изменить способ создания формы. При добавлении формы в проект среда C++Builder добавила в проект код автоматического создания формы. Его нужно удалить и создать форму динамически во время запуска компонента. Во-первых, откройте файл Project1.cpp в окне редактора кода Code Editor и удалите следующую строку из кода функции DllEntryPoint().

```
Application->CreateForm(__classid(TForm2), &Form2);
```

Затем откройте окно просмотра модулей Module Explorer для модуля AppletModule1 с помощью команды меню View⇨Forms или комбинации клавиш <Shift+F12>, а потом выберите модуль AppletModule1 и щелкните на кнопке OK. В окне Object Inspector дважды щелкните справа от события OnActivate во вкладке Events для создания обработчика события OnActivate. Добавьте в него следующий код.

```
void __fastcall TAppletModule1::
↳AppletModuleActivate(TObject *Sender, int Data)
{
   // Динамическое создание формы.
   if (!Form2){
      Form2 = new TForm2(this);
   }

   // Отобразить форму и подождать,
   // пока пользователь не завершит с ней работу.
   Form2->ShowModal();

   // Удаление динамически созданной формы
   // (освобождение созданного объекта).
   delete Form2;
}
```

Окончательный вид этого проекта показан на рис. 8.10.

По окончании всей этой работы сохраните проект с помощью команды меню File⇨Save All в нужном каталоге.

Для проверки работоспособности компонентов панели управления среда C++Builder может автоматически переименовать ваш файл и передать его в соответствующий системный каталог Windows. Эту проверку можно выполнить в модуле компонента. От-

кройте окно просмотра модулей Module Explorer и щелкните правой кнопкой мыши на модуле компонента для отображения контекстного меню с тремя командами: Install Applet (Инсталлировать компонент), Uninstall Applet (Деинсталлировать компонент) и Launch (Запустить панель управления) (рис. 8.11). В данном случае выберите команду меню Install Applet.

После этого C++Builder откомпилирует компонент и выдаст сообщение о его успешной инсталляции. Щелкните правой кнопкой мыши и выберите из контекстного меню команду Launch для открытия панели управления. Дважды щелкните на пиктограмме только что созданного компонента для проверки его работоспособности.

В этот момент на экране должно открыться созданное диалоговое окно с кнопками. Щелкните на кнопке Close для прекращения работы с этим компонентом (рис. 8.12).

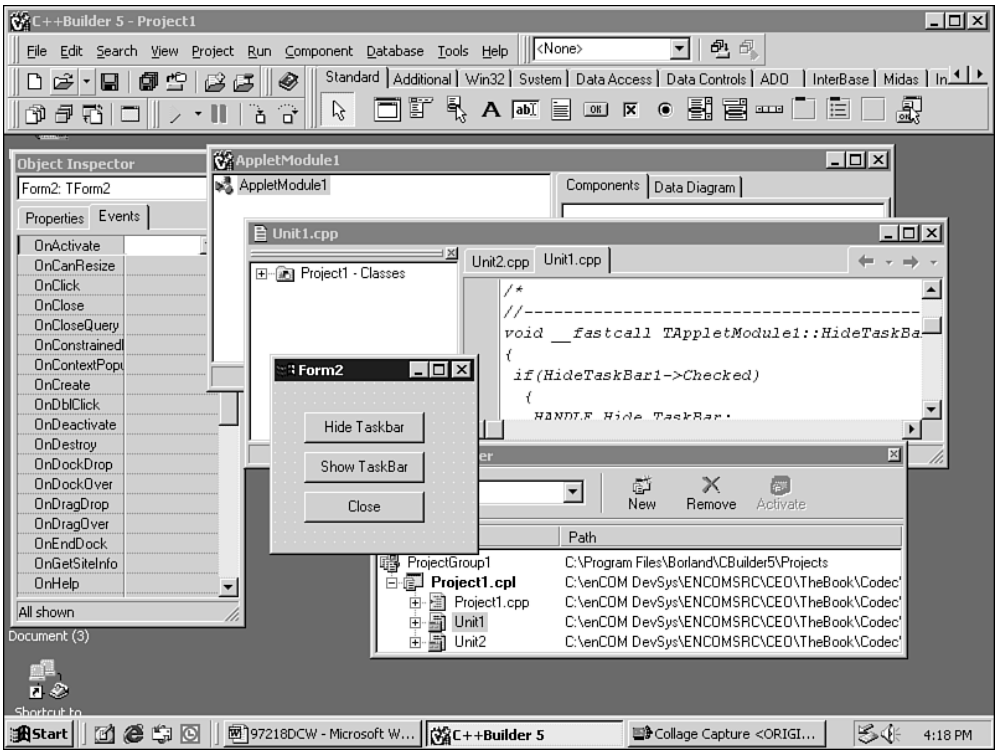


Рис. 8.10. Окончательный вид проекта, готового к проверке работоспособности

Еще один способ запуска компонента панели управления без открытия самой панели управления основан на выборе команды Start⇨Run и вводе приведенной ниже команды в строке запуска команд.

```
rundll32 shell32.dll,Control_RunDLL project1.cpl @0
```

Здесь символ `@` обозначает номер диалогового окна компонента, которое нужно открыть. Например, `@0` означает открытие первого диалогового окна, т.е. первой формы этого компонента.

Так как в нашем приложении только одно диалоговое окно, то для него не обязательно использовать символ `@`. При наличии у компонента двух форм приведенная ниже команда откроет второе диалоговое окно.

```
rundll32 shell32.dll,Control_RunDLL mycpl.cpl @1
```

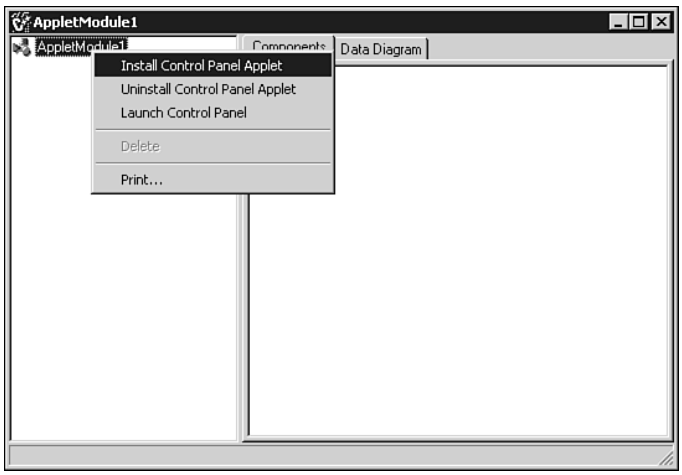


Рис. 8.11. После щелчка правой кнопкой мыши в окне Module Explorer *откроется контекстное меню с командами инсталляции и деинсталляции компонента и командой вызова панели управления*

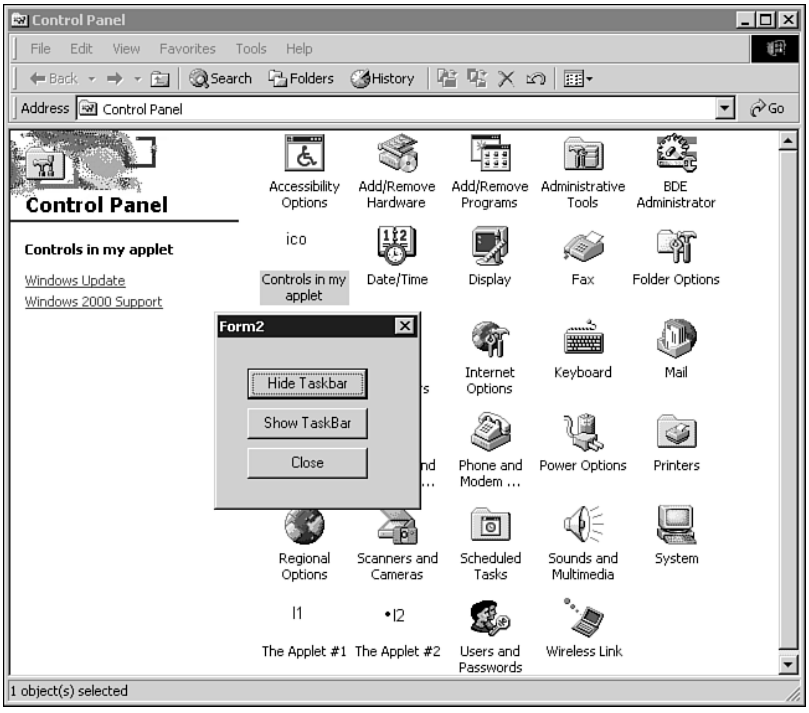


Рис. 8.12. Внешний вид рабочего компонента панели управления

Как это происходит? Программа `rundll32.exe` вызывает библиотеку `shell32.dll`, которая содержит специальную функцию для вызова компонентов и параметров. Попробуйте применить ее с другими компонентами панели управления операционной системы

Windows. Для некоторых из них эта команда будет выполнена успешно, для других — нет. Дело в том, что не все они имеют в своем составе несколько модулей или приложений. Тем не менее эта команда позволяет управлять работой приложения панели управления без использования самой панели управления.

Заключительные замечания относительно компонентов панели управления

Создание приложений панели управления может быть очень длительной и утомительной процедурой, поскольку она включает создание и компиляцию приложения, переименование файла, копирование его в системную папку `System` основного каталога Windows, а также тестирование. Причем панель управления не всегда адекватно отображает созданный вами компонент. Однако среда C++Builder может автоматически выполнить большую часть этой работы.

Применение компонентов сторонних разработчиков

C++Builder содержит большое количество встроенных компонентов. Точное их число зависит от номера версии C++Builder. Как правило, встроенные компоненты подходят для большинства решаемых разработчиками задач. Однако часто бывает, что программисту для решения специализированной задачи требуется особый компонент. В таком случае у него есть два способа решения этой проблемы. Во-первых, приобрести компонент у сторонних разработчиков (или получить демонстрационную версию по сети Internet). Во-вторых, создать его самому. В этом разделе рассматривается первый вариант, а в главах 9, 10 и 11 подробно описывается второй вариант.

Преимущества и недостатки компонентов сторонних разработчиков

Использование компонентов сторонних разработчиков имеет как преимущества, так и недостатки.

Преимущества использования компонентов сторонних разработчиков

- Не нужно изобретать колесо. Если кто-то уже преуспел в деле создания нужного вам компонента, то вряд ли стоит заниматься разработкой и отладкой этой части вашего проекта.
- Приобретая исходный код, можно продолжить работу над этим компонентом и исправить ошибки, найденные при тестировании.
- Иногда компоненты сторонних разработчиков сопровождаются дополнительной поддержкой, обычно по электронной почте.
- Если в проекте не требуется передавать исходный код, эти компоненты можно использовать в последующих проектах, сокращая время их создания.
- Если компонент является оболочкой решения сложной задачи, то при использовании такого компонента можно сэкономить много времени и ресурсов, которые понадобятся для создания других важных функций создаваемого приложения.

Недостатки компонентов сторонних разработчиков

- Некоторое время придется потратить на тестирование компонентов, чтобы убедиться в их работоспособности и качестве.
- При отсутствии исходного кода этого компонента его нельзя модернизировать.
- Нет никаких гарантий, что этот компонент не приводит к утечке памяти или другим нежелательным побочным эффектам. Эти сомнения можно разрешить при наличии исходного кода, но для этого также придется затратить время, чтобы разобраться в методологии использования компонента и тщательно проверить его работу.
- Многие разработчики создают компоненты с помощью Delphi, а это значит, что их код написан на языке Pascal, а не C++. Однако есть надежда, что в будущем это соотношение изменится.
- Если компонент работает не так, как ожидалось, или возникают трудности с его реализацией, необходима помощь разработчика. Если условно бесплатный компонент был скопирован в сети Internet, то такой вид поддержки не всегда гарантируется.
- Приобретаемые компоненты повышают общую стоимость проекта. Если эти расходы не были учтены с самого начала, вам придется самостоятельно решить, оправдано ли его приобретение. Это не очень важно, если компонент планируется также использовать в будущих проектах, где расходы могут полностью окупиться.
- Если клиент оплачивает исходный код вашего проекта, то ему понадобятся также компоненты сторонних разработчиков. Поэтому вам придется организовать передачу прав владения компонентов (или приобрести дополнительные компоненты) по окончании работы над проектом.

Дополнительные ресурсы C++Builder

Глобальная сеть Internet — самый большой источник компонентов сторонних разработчиков. Но сеть Internet огромна, где же начать поиск?

В сети Internet существует огромное количество специальных Web-узлов, которые содержат гораздо больше компонентов, чем нужно. Помимо компонентов, там можно найти огромное количество справочной информации, учебных пособий и других материалов, которые помогут вам в процессе создания проекта C++Builder. Дополнительный список Web-узлов с полезной информацией приведен в приложении А.

Резюме

C++Builder — это инструмент быстрого создания приложений, основанный на объектно-ориентированной технологии и идеологии повторного использования кода. В этой главе рассматриваются способы использования существующих компонентов и обсуждаются их преимущества и недостатки.

Кратко рассмотрена библиотека VCL, основной объект `TObject` и пять его основных наследников — `TPersistent`, `TComponent`, `TControl`, `TGraphicControl` и `TWinControl`.

Затем рассмотрен механизм определения типов при выполнении приложения (RTTI), который предоставляет IDE-среде C++Builder возможность отображать свойства и события компонентов с помощью окна Object Inspector. При этом подчеркивается особое значение способности окна Object Inspector определять открытые свойства и события объектов, которые совместно используют основные объекты.

Рассмотрены объекты библиотеки VCL с точки зрения их размещения в куче, а не в стеке. При этом описаны расширения языка C++, которые позволяют интегрировать конструкции Object Pascal из библиотеки VCL в код на языке C++ среды C++Builder.

Затем рассмотрены стандартные и универсальные элементы управления, которые предусмотрены в C++Builder. Эти элементы управления используют в работе библиотеку `COMCTL32.DLL`, которая входит в состав операционной системы Microsoft Windows. Основным препятствием для разработчиков компонентов и классов является необходимость обновления этих компонентов и классов, которая вызвана постоянной модернизацией продуктов фирмы Microsoft. Разработчику всегда нужно учитывать, что клиенты могут использовать его программные продукты в самых разных версиях операционной системы. При этом не каждый из них может иметь последнюю версию DLL-библиотеки, а потому необходимо предусмотреть возможность ее модернизации.

Каждая версия C++Builder содержит модернизированную версию библиотеки VCL, в которой учитываются обновления, внесенные в библиотеку `COMCTL32.DLL` фирмы Microsoft. В то же время фирма Inprise постоянно усовершенствует возможности библиотеки VCL. В этой главе рассмотрены усовершенствования пользовательских событий рисования для элементов управления `TTreeView`, `TListView` и `TToolBar`. Кроме того, кратко описан новый компонент подсказки (Help Hint), компонент `TApplicationEvents`, расширенные возможности для работы с меню, компонент `TIcon` и новые способы доступа к ключам реестра.

Компоненты панели управления представляют собой стандартный способ конфигурирования аппаратного и программного обеспечения. В этой главе приведен пример создания собственного компонента панели управления с помощью C++Builder.

Для демонстрации широких возможностей библиотеки VCL показано, как класс `TStringList` списка строк может применяться в качестве базового класса для более мощного объекта, способного связывать другие объекты (того же типа) с каждой строкой списка. При этом особое внимание уделено вопросу правильного применения объектно-ориентированного подхода для создания простого и удобного в использовании и сопровождении кода. Класс `TStringList` содержит виртуальные методы, которые выполняют такие задачи, как создание, сортировка и удаление строк. Метод сортировки `Sort` использовался в качестве примера для демонстрации способов переопределения метода при сортировке всех объектов, связанных с каждой строкой списка.

C++Builder — мощный и простой в употреблении объектно-ориентированный RAD-инструмент. Знание основных принципов работы библиотеки VCL и ее взаимосвязи с конструкциями Object Pascal, а также опыт и знания, полученные вами при работе с языком C++ и расширениями C++Builder, явятся надежным фундаментом для работы над будущими проектами.

Глава

9

Создание пользовательских компонентов

Малькольм Смит
Шон Рок
Джэйми Оллсоп

Библиотека визуальных компонентов или библиотека VCL (Visual Component Library — VCL) — чрезвычайно мощный инструмент, который позволяет существенно упростить создание приложения с помощью многочисленных компонентов, классов и методов, которые предусмотрены в среде C++Builder. Однако в некоторых ситуациях можно обнаружить, что имеющиеся компоненты не полностью охватывают тот круг задач, которые должно решать приложение. Способность создавать и модифицировать компоненты в среде C++Builder является ощутимым преимуществом по сравнению с другими языками программирования. Именно благодаря этой способности C++Builder пользуется большой популярностью среди программистов всего мира. Изучая процесс создания пользовательских компонентов, можно ближе познакомиться с принципами работы библиотеки VCL и повысить производительность работы с C++Builder. Судя по тому количеству коммерческих компонентов, который предлагаются в сети Internet, можно сделать вывод, что это занятие очень выгодно.

В этой главе вы познакомитесь с созданием свойств, методов и событий обычных компонентов; созданием компонентов, связанных с данными; связыванием компонентов; обработкой исключительных ситуаций в компонентах; модификацией существующих компонентов; способами реагирования на сообщения, получаемые компонентом; а также научитесь рисовать компоненты на экране. При этом будут описаны некоторые аспекты работы с Windows API-функциями, поэтому для более близкого знакомства Win32 API-функциями читателю рекомендуется также прочесть главу 24. Кроме того, подробную информацию о Win32 API-функциях можно получить в справке C++Builder. В этой главе также упоминаются пакеты, поэтому читателю будет полезно еще раз просмотреть главу 2, посвященную работе с пакетами, и главу 15, в которой рассмотрено использование DLL-библиотек и дополнительных модулей. Дополнительную информацию по этой теме можно также найти в разделах “PACKAGE macro” и “Creating packages and DLLs” в справке C++Builder.

Для чего нужны пользовательские компоненты

На первый взгляд задача создания компонентов может показаться очень сложной. Однако после внимательного изучения нескольких статей и учебных пособий по этой теме нетрудно найти отправную точку для начала работы над созданием компонентов. Самый простой способ заключается в редактировании функций и внешнего вида уже существующего компонента.

Еще один тривиальный способ основан на настройке и расширении стандартных компонентов библиотеки VCL в соответствии с требованиями создаваемого вами рабочего приложения. Например, при создании приложения базы данных можно разместить компонент `TDBGrid` в форму и отредактировать нужным образом значения соответствующих свойств. Для создания утилит внутреннего пользования можно разместить компонент строки состояния `TStatusBar` в форме, добавить несколько панелей и удалить маркер изменения размеров. Вместо того чтобы делать это в каждом проекте, следует создать собственный пользовательский компонент, в котором эти свойства будут задаваться автоматически. Это не только ускорит процесс создания приложения, но и позволит избежать появления в них ошибок. Кроме того, при обнаружении в таком компоненте ошибки достаточно просто перекомпилировать все проекты, в которых он используется, а не исправлять их вручную в каждом из проектов. Проекты унаследуют все изменения без необходимости их дополнительного перепрограммирования.

Общие принципы создания компонентов

Существуют разные типы компонентов, поэтому типы компонентов-наследников создаваемых вами компонентов будут предопределены типом их компонента-предка.

Невизуальные (*non-visual*) компоненты происходят от компонента `TComponent`. Компонент `TComponent` является самым базовым компонентом, который может быть использован для создания компонента, так как обладает необходимой функциональностью для интеграции в IDE-среде и потоковой обработки значений свойств. Более подробно механизм обработки потоков описывается в главе 8, посвященной использованию компонентов библиотеки VCL.

Невизуальным называется компонент, который является оболочкой сложного кода без какого-либо видимого представления для пользователя. Например невизуальным является компонент, который получает сообщение об ошибке и автоматически посылает его соответствующему элементу управления типа `TMemo` или `TRichEdit` или записывает его в файл на жестком диске. Сам по себе этот компонент невидим для пользователя приложения, но может работать в фоновом режиме, предоставляя свои функции для использования в приложении.

Оконные (*windowed*) компоненты происходят от компонента `TWinControl`. Эти объекты видны пользователю во время выполнения приложения, и он может с ними взаимодействовать (например, выбирать файл в списке). На основе `TWinControl` можно создавать собственные компоненты. Для упрощения этой задачи в C++Builder предусмотрен компонент `TCustomControl`.

Графические (*graphic*) компоненты аналогичны оконным, за исключением того, что не имеют атрибутов окна и потому пользователь не может взаимодействовать с ними в интерактивном режиме. Отсутствие этих атрибутов уменьшает количество потребляемых ресурсов. Хотя эти компоненты не взаимодействуют с пользователем, можно организовать их ответную реакцию на такие оконные сообщения, как например, генерируемые мышью события. Эти компоненты происходят от компонента `TGraphicControl`.

Почему в качестве заготовок стоит использовать соответствующие компоненты

Преимущество использования в качестве заготовок уже имеющихся компонентов заключается в сокращении времени создания проектов. При этом сводится к минимуму вероятность появления ошибок во всех производных компонентах, которые используются в ваших проектах.

Рассмотрим, например, компонент-надпись `TLabel`, который может несколько раз использоваться в одном и том же проекте. Если в каждом новом проекте необходимо придерживаться определенного стиля оформления, то вам потребуется создать много таких надписей и однообразно изменить их свойства в каждом новом приложении. Но, создавая пользовательский компонент-наследник от компонента `TLabel`, потребуется только добавить их в форму и изменить их подпись и расположение.

Для демонстрации этого способа создадим компонент в течение одной минуты за счет ввода только трех строк кода. Для этого в среде C++Builder выберите команду меню Component⇨New Component. Затем в диалоговом окне New Component в поле Ancestor type (Тип предшественника) введите **`TLabel`**, а в поле Class Name — **`TStyleLabel`**. Для компонента, который предполагается разместить в панели компонентов Component Palette среды C++Builder и использовать впоследствии в других приложениях, вероятно, стоит выбрать информативное имя для класса. В этом примере следует принять все предлагаемые по умолчанию значения других параметров и щелкнуть на кнопке OK. После этого C++Builder автоматически создаст необходимые файлы, и все, что потребуется от программиста, — до-

бавить в код компонента несколько строк с указанием значений свойств данной надписи. После внесения необходимых изменений, сохраните файл и выберите команду меню Component⇨Install Component. Если файл этого компонента открыт в среде C++Builder, то в поле Unit File Name (Имя файла модуля) будет показано имя файла. Щелкните на кнопке OK для инсталляции этого компонента в палитре компонентов Component Palette. (Прим. ред. — При этом вам также потребуется указать имя уже существующего или нового пакета.) В листингах 9.1 и 9.2 показан полный код нового компонента.

Листинг 9.1. Заголовочный файл `StyleLabel.h` компонента `TStyleLabel`

```
//---------------------------------------------------------------
#include <SysUtils.hpp>
#include <Controls.hpp>
#include <Classes.hpp>
#include <Forms.hpp>
#include <StdCtrls.hpp>
//---------------------------------------------------------------
class PACKAGE TStyleLabel :public TLabel
{
private:
protected:
public:
   __fastcall TStyleLabel(TComponent*Owner);
   __published:
};
//---------------------------------------------------------------
#endif
```

Листинг 9.2. Файл `StyleLabel.cpp` компонента `TStyleLabel`

```
//---------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "StyleLabel.h"
#pragma package(smart_init)
//---------------------------------------------------------------
// Функция ValidCtrCheck используется для того, чтобы
// гарантировать создание компонентов без пустых виртуальных
// функций.
//
static inline void ValidCtrCheck(TStyleLabel *)
{
   new TStyleLabel(NULL);
}
//---------------------------------------------------------------
__fastcall TStyleLabel::
↳TStyleLabel(TComponent*Owner) : TLabel(Owner)
{
   Font->Name = "Verdana";
   Font->Size = 12;
```

```
   Font->Style = Font->Style << fsBold;
}
//---------------------------------------------------------
namespace Stylelabel
{
   void __fastcall PACKAGE Register()
   {
      TComponentClass classes[1] == {__classid(TStyleLabel)};
      RegisterComponents("TestPack", classes, 0);
   }
}
//---------------------------------------------------------
```

Еще одно преимущество использования в качестве заготовок уже имеющихся компонентов — возможность создания базового класса с необходимой функциональностью и неопубликованными свойствами. Например, можно создать компонент `TListBox` с неопубликованным свойством `Items`. Создавая компонент-наследник от компонента `TCustomListBox`, можно опубликовать те свойства, к которым пользователь сможет осуществлять доступ при создании приложения, а к другим свойствам (например, к свойству `Items`) — только во время выполнения.

Наконец, свойства и события, добавленные в уже имеющийся компонент, позволяют сэкономить время и усилия на создание этого компонента с самого начала.

Создание пользовательских компонентов

Хотя это очевидно, все же стоит обратить ваше внимание на то, что создавая пользовательские компоненты, следует придерживаться тех же общих правил, что и при создании приложения в целом. Прежде всего следует тщательно обдумать назначение создаваемых компонентов. Упомянутые выше компоненты с перечнем сведений о базах данных вряд ли стоит создавать на основе компонента `TListBox`. Лучше создать пользовательскую версию класса `TCustomListBox`, содержащую дополнительные свойства, которые будут использоваться во всех создаваемых наследниках. Каждый новый компонент будет основан на этой пользовательской версии, что исключит необходимость создания трех разных версий одного и того же кода. Окончательная версия каждого компонента будет содержать только тот код свойств, методов и событий, который отличается от кода в других версиях.

Диаграмма классов библиотеки VCL

Чтобы получить более полное и четкое представление об архитектуре библиотеки VCL, рекомендуется внимательно изучить диаграмму ее классов, которая поставляется вместе с C++Builder. Она содержит краткое визуальное представление не только основных, но и производных компонентов библиотеки VCL.

В процессе изучения основных принципов проектирования и создания компонентов рекомендуется моделировать архитектуру создаваемых компонентов в таком же объектно-ориентированном стиле. Это можно сделать с помощью строгих и универсальных базовых классов, на основе которых создаются пользовательские компоненты. Хотя исходный код для компонентов C++Builder создается на языке Pascal, все же стоит познакомиться с этими базовыми классами поближе, чтобы понять основные принципы их взаимодействия. Благодаря этому вскоре вы поймете, что компоненты, которые совместно используют некоторые свойства, на самом деле являются производными от одного базового класса или его наследниками.

Наконец, эта диаграмма позволяет сразу же определить базовые классы, которые необходимые для удовлетворения потребностей вашего пользовательского компонента. Вместе с файлами справки библиотеки VCL они позволят вам быстро найти наиболее подходящий класс, на основе которого можно вывести собственные компоненты. Как уже упоминалось ранее, наиболее общими базовыми классами являются классы `TComponent`, `TWinControl` или `TGraphicControl`, в зависимости от типа создаваемого компонента.

Создание невизуальных компонентов

Каждый компонент содержит свойства, события и методы. Они подробно рассматриваются в этом разделе, а также описывается структура компонентов и принципы их взаимодействия, которые позволяют использовать их в C++Builder в качестве строительных блоков создаваемых приложений.

Свойства

Свойства бывают опубликованными и неопубликованными. *Опубликованные* (*published*) свойства доступны пользователю в IDE-среде C++Builder при создании и выполнении приложения. *Неопубликованные* (*non-published*) свойства используются только при выполнении приложения. Рассмотрим сначала неопубликованные свойства.

Неопубликованные свойства

Компонент представляет собой запакованный класс с некоторыми дополнительными функциями. Рассмотрим пример такого класса, который показан в листинге 9.3.

Листинг 9.3. Присвоение и извлечение значений закрытых переменных

```
class LengthClass
{
private:
   int FLength;
public:
   LengthClass(void){}
   ~LengthClass(void){}
   int GetLength(void);
   void SetLength(int FLength);
   void LengthFunction (void);
}
```

В листинге 9.3 показана закрытая переменная, которая используется только внутри класса, и методы, которые используются в этом приложении для записи и чтения значения этой переменной. Такой способ работы может легко привести к путанице. Рассмотрим еще один пример, показанный в листинге 9.4.

Листинг 9.4. Присвоение и извлечение значения переменной с помощью методов

```
LengthClass Rope;
Rope.SetLength(15);
// другие действия
int NewLength = Rope.GetLength();
```

Код, показанный в листинге 9.4, совсем прост, но он очень быстро усложняется в достаточно большом приложении. Не лучше ли ссылаться на переменную Length как на свойство этого класса? Эта задача легко решается с помощью C++Builder, например, так, как показано в листинге 9.5.

Листинг 9.5. Использование свойства для присвоения и извлечения значений закрытых переменных

```
class LengthClass2
{
private:
   int FLength;
public:
   LengthClass2(void){}
   ~LengthClass2(void){}
   void LengthFunction(void);
   __property int Length = {read = FLength, write = FLength};
}
```

Теперь показанный в листинге 9.4 код будет выглядеть так, как показано в листинге 9.6.

Листинг 9.6. Присвоение и извлечение значения переменной с помощью свойства

```
LengthClass Rope;
Rope.Length = 15;
// другие действия
int NewLength = Rope.Length;
```

Объявление класса теперь содержит ключевое слово __property, которое является расширением языка C++ в C++Builder. С помощью ключевых слов read и write в листинге 9.6 теперь выполняются операции чтения и записи свойства Length, т.е. переменной FLength.

Зачем такие сложности, когда переменную FLength можно просто сделать открытой? Это объясняется следующим.

- Свойство Length можно сделать доступным только для чтения, если не использовать ключевое слово write.
- Таким образом, можно предоставить открытый доступ к закрытой информации класса без изменения его реализации. Это особенно важно, если значение свойства унаследовано или при его изменении выполняется некоторое действие.

В листинге 9.7 показан еще один вариант применения этого способа.

Листинг 9.7. Комбинация присвоения и получения значения переменной с помощью методов и свойств

```
class LengthClass3
{
private:
   int FLength;
   int GetLength(void);
   void SetLength(int FLength);
public:
```

```
   LengthClass3(void){}
   ~LengthClass3(void){}
   void LengthFunction(void);
   __property int Length = {read = GetLength,
                            write = SetLength};
}
```

Показанный в листинге 9.7 пример демонстрирует широкие возможности применения свойств. В объявлении свойства сказано, что при чтении свойства `Length` вызывается метод `GetLength()` с возвратом значения закрытой переменной, а при записи нового значения для свойства `Length` вызывается метод `SetLength()`, который и задает новое значение закрытой переменной.

Метод `GetLength()` может выполнять дополнительные вычисления на основе других закрытых членов этого класса. А метод `SetLength()` может проверять корректность введенного значения и выполнять другие действия еще до окончательного присвоения нового значения закрытой переменной `FLength`.

В C++Builder примером такой ситуации может быть подключение к базе данных при изменении разработчиком имени псевдонима. В этом случае компонент отсоединяется от текущей базы данных (если такая существует) перед попыткой подключения к новому источнику данных. Эти действия скрыты от конечного пользователя, но ими можно управлять с помощью свойств.

Типы свойств

Свойство может иметь очень простой тип, например `int`, `bool`, `short` и т.п., или очень сложный, например быть экземпляром пользовательского класса. При использовании пользовательских классов в качестве типов свойств следует учитывать следующие особенности. Во-первых, этот класс должен быть производным от класса `TPersistent` (как минимум), если его содержимое предполагается передавать в потоке в форму. Во-вторых, при предварительном объявлении класса необходимо использовать ключевое слово `__declspec(delphiclass)`.

В листинге 9.8 показан типичный пример использования предварительного объявления. Обратите внимание, что свойство еще не определено.

Листинг 9.8. Предварительное объявление

```
class MyClass;
class PACKAGE MyComponent : public TComponent
{
private:
   MyClass *FMyClass;
// ...
};

class MyClass : public TPeristent
{
public:
   __fastcall MyClass (void){}
};
```

Ключевое слово `PACKAGE` между именем класса и ключевым словом `class` представляет собой макрос, который содержит код экспорта из BPL-библиотеки (Borland Package Library — BPL). Эта библиотека представляет собой разновидность DLL-библиотеки и позво-

ляет нескольким приложениям совместно использовать код. Более подробную информацию о пакетных библиотеках и макросе PACKAGE можно найти в разделах “PACKAGE macro” и “Creating packages and DLLs” в интерактивной справке C++Builder.

Однако при добавлении свойства типа MyClass потребуется изменить это предварительное объявление, как показано в листинге 9.9.

Листинг 9.9. Свойство типа пользовательский класс

```
class __declspec(delphiclass)MyClass;

class PACKAGE MyComponent : public TComponent
{
private:
   MyClass *FMyClass;
   // ...

__published:
   __property MyClass *Class1 = {read = FMyClass,
                                 write = FMyClass};
};

class MyClass : public TPeristent
{
public:
   __fastcall MyClass (void){}
};
```

Опубликованные свойства

Опубликованные свойства предоставляют пользователям доступ к свойствам компонента внутри IDE-среды C++Builder при создании приложения. Эти свойства отображаются в окне Object Inspector, где пользователь может редактировать их текущие значения. Эти свойства также доступны и при выполнении приложения, но их основное назначение — предоставить пользователю возможность быстро задать значения свойств компонента без создания дополнительного кода. Кроме того, значения опубликованных свойств передаются с помощью потока в форму, а потому их значения являются устойчивыми. Это значит, что значения будут восстанавливаться каждый раз при открытии проекта и запуске выполняемого файла.

Опубликованные свойства определяются точно так же, как и другие свойства, но их объявления располагаются в разделе __published объявления класса. В листинге 9.10 показан типичный пример такого определения.

Листинг 9.10. Определение опубликованного свойства

```
class PACKAGE LengthClass : public TComponent
{
private:
   int FLength;
   int GetLength(void);
   void SetLength(int pLength);

public:
   __fastcall LengthClass(TObject *Owner) : TComponent(Owner) {}
```

```
   __fastcall ~LengthClass(void){}
   void LengthFunction(void);

__published:
   __property int Length = {read = Getlength,
                            write = Setlength};
}
```

Этот класс аналогичен классу из листинга 9.9, за исключением того, что свойство Length перемещено в раздел __published. Опубликованные свойства, представленные в окне Object Inspector доступны для чтения и записи их значений, однако их можно сделать доступными только для чтения и видимыми в IDE-среде с помощью подстановочного метода записи. Для создания в приведенном выше компоненте опубликованного свойства, в котором хранится номер текущей версии компонента, можно использовать код, приведенный в листинге 9.11.

Листинг 9.11. Свойство с версией компонента

```
const int MajorVersion = 1;
const int MinorVersion = 0;

class PACKAGE LengthClass : public TComponent
{
private:
   AnsiString FVersion;
   int FLength;
   int GetLength(void);
   void SetLength(int pLength);
   void SetVersion(AnsiString /*pVersion */)
   {
      FVersion = AnsiString(MajorVersion) + "." +
      AnsiString(MinorVersion);
   }

public:
   __fastcall LengthClass(TObject *Owner) : TComponent(Owner)
   { SetVersion(""); }
   __fastcall ~LengthClass(void){}
   void LengthFunction(void);

__published:
   __property int Length = {read = Getlength,
                            write = Setlength};
   __property AnsiString Version = {read = FVersion,
                                    write = SetVersion};
}
```

Здесь определена закрытая переменная FVersion, значение которой присваивается в конструкторе класса. Затем в разделе __published создается свойство Version с правами чтения и записи его значения. Ключевое слово read возвращает значение переменной FVersion,

а слово write задает новое значение переменной FVersion. Имя переменной в списке параметров функции SetVersion() закомментировано, чтобы предотвратить генерирование компилятором предупреждения о том, что переменная объявлена, но не используется. Так как свойство имеет тип AnsiString, то метод SetVersion() по определению должен в объявлении иметь параметр типа AnsiString.

Свойство-массив

Некоторые свойства могут быть массивами, а не только такими простыми типами данных как bool, int и AnsiString. Этот аспект не очень хорошо документирован для пользователей. Примером свойства-массива является свойство Lines компонента TMemo. Это свойство позволяет пользователю осуществлять доступ к отдельным строкам компонента Memo.

Свойство-массив объявляется точно так же, как и другие свойства, но имеет два отличия: объявление включает соответствующие индексы с заданными типами, и индексы не обязательно целые числа. В листингах 9.12–9.15 приведен пример использования двух свойств. Одно содержит в качестве индекса строку, а второе — целое число.

Листинг 9.12. Использование строки в качестве индекса

```
class PACKAGE TStringAliasComponent : public TComponent
{
private:
   TStringList RealList;
   TStringList AliasList;
   __AnsiString __fastcall GetStringAlias(AnsiString RawString);
   AnsiString __fastcall GetRealString(int Index);
   void __fastcall SetRealString(int Index, AnsiString Value);
public:
   __property AnsiString AliasString[AnsiString RawString] ==
      {read = GetStringAlias};
   __property AnsiString RealString[int Index] ==
      {read=GetRealString, write=SetRealString};
}
```

В этом примере компонент может хранить список строк вместе со списком псевдонимов. Свойство AliasString принимает значение RawString и возвращает псевдоним с помощью метода GetStringAlias(). При создании свойств-компонентов многие разработчики смущены тем, что в объявлении используется индексная нотация (т.е. квадратные скобки []), а в коде используется такой же способ записи, как и при вызове обычного метода. Например, свойство RealString имеет не только возвращаемое значение типа AnsiString, но и целое число в качестве индекса. Метод GetRealString() используется для извлечения строки списка на основе заданного значения индекса, как показано в листинге 9.13.

Листинг 9.13. Метод чтения свойства-массива

```
AnsiString __fastcall TStringAliasComponent::
↳GetRealString(int Index)
{
   if(Index > (RealList->Count - 1))
      return "";
   return RealList->Strings[Index];
}
```

В коде это свойство будет выглядеть следующим образом.

```
AnsiString str = StringAlias1->RealString[0];
```

Рассмотрим подробнее метод `SetRealString()`. На первый взгляд объявление этого метода выглядит несколько странно. Первый параметр является целочисленной переменной и используется как индекс, а второй — переменной типа `AnsiString`. Переменная `RealList` типа `TStringList` вставляет в список строку типа `AnsiString` в месте, указанном с помощью целочисленного параметра-индекса. В листинге 9.14 показан код определения метода `SetRealString()`.

Листинг 9.14. Метод записи значений свойства-массива

```
void __fastcall TStringAliasComponent::SetRealString(int Index,
                                        AnsiString Value)
{
   if((RealList->Count - 1)) < Index)
      RealList->Add(Value);
   else
      RealList->Insert(Index, Value);
}
```

В листинге 9.14 значение параметра `Index` сверяется с количеством строк, которые уже находятся в списке. Если значение параметра `Index` больше этого количества, то строка `Value` просто добавляется в конец списка. В противном случае вызывается метод `Insert()` объекта `TStringList` для вставки строки в месте, указанном значением индекса `Index`. Теперь в этот список можно вставить следующую строку.

```
StringAlias1->RealString[1] == "некоторая строка";
```

Рассмотрим еще один интересный момент. Метод `GetStringAlias()` используется для чтения значения свойства `AliasString`, в котором в качестве индекса используется строка. Список строк является массивом строк, поэтому каждая строка имеет индекс или определенное место в этом списке. Метод `IndexOf()` компонента `TStringList` используется для сравнения переданной строки со строками, которые уже находятся в списке. Этот метод возвращает целое значение, которое является индексом строки в этом списке, или значение `-1`, если такой строки нет. Теперь остается только вернуть из списка псевдонимов строку с индексом, указанным в вызове метода `IndexOf()`. Соответствующий код показан в листинге 9.15.

Листинг 9.15. Метод `GetStringAlias()`

```
AnsiString __fastcall TStringAliasComponent::GetStringAlias(
                        AnsiString RawString)
{
   int Index;
   Index = RealList->IndexOf(RawString);
   if((Index == -1) || (Index >(AliasList->Count-1)))
      return RawString;

   return AliasList->Strings [Index ];
}
```

Для применения этого свойства можно использовать следующий код.

```
AnsiString MyAliasString =
   StringAlias1->AliasString("The Raw String");
```

Другие способы работы со свойствами

В листингах 9.5–9.15 приведены примеры использования ключевых слов `read` и `write` в объявлении метода. В C++Builder также предусмотрено три дополнительных ключевых слова: `default`, `nodefault` и `stored`.

Ключевое слово `default` не задает для свойства предлагаемое по умолчанию значение, а сообщает C++Builder о том, какое значение по умолчанию будет присвоено этому свойству (разработчиком) в конструкторе компонента. IDE-среда впоследствии использует эту информацию для определения значения свойства, которое нужно передать в потоке в данную форму. Если этому свойству будет присвоено значение, эквивалентное значению по умолчанию, то такое значение свойства не будет сохранено как часть формы. Вот пример использования этого ключевого слова.

```
__property int IntegerProperty = {read = Finteger,
                                  write = Finteger,
                                  default = 10};
```

Ключевое слово `nodefault` сообщает IDE-среде о том, что свойство не связано ни с каким значением по умолчанию. Когда свойство объявляется в первый раз, нет необходимости использовать для него это ключевое слово, поскольку отсутствие значения по умолчанию означает то же самое. Ключевое слово `nodefault` в основном применяется в случае изменения определения унаследованного свойства, например, так, как показано ниже.

```
__property int DescendantInteger = {read = Finteger,
                                    write = Finteger,
                                    nodefault};
```

Учтите, что значение свойства с ключевым словом `nodefault` в объявлении передается потоком в форму только в тех случаях, когда значение присваивается свойству или базовой переменной с помощью одного из ее методов или с помощью окна Object Inspector.

Ключевое слово `stored` применяется для управления способом сохранения свойств. По умолчанию сохраняются все опубликованные свойства. Это поведение можно изменить, задав для ключевого слова `stored` значения `true` или `false` либо имя функции, которая возвращает логическое значение. В листинге 9.16 показан типичный пример использования ключевого слова `stored`.

Листинг 9.16. Использование ключевого слова `stored`

```
class PACKAGE LengthClass :public TComponent
{
protected:
   int FProp;
   bool StoreProperty(void);

__published:
   __property int AlwaysStore = {read = FProp, write = FProp,
                                 stored = true};
   __property int NeverStore = {read = FProp, write = FProp,
```

```
                                stored = false};
   __property int SimetimesStore = {read = FProp, write = FProp,
                                    stored = StoreProperty};
}
```

Порядок создания

Если компонент содержит свойства, которые зависят от значений других свойств во время передачи данных потоком, то можно управлять порядком их загрузки (а значит и инициализации), объявляя их в нужном порядке в заголовочном файле класса. Например, код в листинге 9.17 загружает свойства в следующем порядке: `PropA`, `PropB` и `PropC`.

Листинг 9.17. Зависимости между значениями свойств

```
class PACKAGE SampleComponent : public TComponent
{
private:
   int FPropA;
   bool FPropB;
   String FProC;
   void __fastcall SetPropB(bool pPropB);
   void __fastcall SetPropC(String pPropC);
public:
   __property int PropA = {read = FPropA, write = FPropA};
   __property bool PropB = {read = FPropB, write = SetPropB};
   __property String PropC = {read = FPropC, write = SetPropC};
}
```

Если при работе с взаимозависимыми свойствами возникают проблемы с их инициализацией, убедитесь в правильности порядка их объявления в определении класса.

События

Событием в компоненте является вызов необязательного метода в ответ на некоторое происшествие. Это происшествие может использоваться пользователем для выполнения каких-либо действий, прежде чем компонент продолжит работу, для обработки исключительной ситуации или для перехвата сообщения Windows.

Рассмотрим простой пример, в котором компонент просматривает каталоги, начиная с заданного корневого каталога. Если в компоненте предусмотрена возможность изменения текущего каталога, то это происшествие будет выглядеть для пользователя как *событие*. При возникновении такого события компонент попытается найти созданный для этого пользователем обработчик события (т.е. метод, связанный с этим событием) и вызовет соответствующий метод. Рассмотрим код, приведенный в листинге 9.18.

Листинг 9.18. Объявление свойства-события

```
class PACKAGE TTraverseDir : public TComponent
{
private:
   AnsiString FCurrentDir;
   TNotifyEvent *FOnDirChanged;
```

```
public:
   __fastcall TTraverseDir(TObject *Owner) : TComponent(Owner)
   {
      FOnDirChanged = 0;
   }
   __fastcall ~TTraverseDir(void){}
   __fastcall Execute();

__published:
   __property AnsiString CurrentDir = {read = FCurrentDir};
   __property TNotifyEvent OnDirChanged =
                              {read = FOnDirChanged,
                               write = FOnDirChanged};
}
```

В листинге 9.18 показан код объявления свойства, используемого только для чтения, а также код стандартного события. При выполнении этого компонента текущий каталог может измениться. Рассмотрим следующий фрагмент кода.

```
void __fastcall TTraverseDir::Execute(void)
{
   // Обход каталогов.

   // Здесь меняется текущий каталог,
   // происходит вызов обработчика события DirChanged,
   // если он задан.
   if(FOnDirChanged)
      FOnDirChanged(this);

   // Другой код компонента.
}
```

Переменная `FOnDirChanged` в предыдущем примере является указателем на компонент `TNotifyEvent`, который имеет следующее объявление.

```
typedef void __fastcall
   (__closure *TNotifyEvent)(System::TObject* Sender)
```

Как видите, это объявление означает. что используется только один параметр типа `TObject*`. При создании этого события (с помощью двойного щелчка в поле этого события в окне Object Inspector) IDE-среда создаст следующий код.

```
void __fastcall TTraverseDir::Traverse1DirChanged(
                                 TObject *Sender)
{
}
```

Теперь пользователь может добавить здесь свой код, который будет выполнен при вызове этого события. В этом случае событие является стандартным событием, которое просто передает указатель тому объекту, который вызвал появление события. Этот указатель позволяет различать несколько однотипных компонентов внутри проекта.

```
void __fastcall TTraverseDir::Traverse1DirChanged(
                                 TObject *Sender)
```

```
{
   if(Sender == Traverse1)
      // Код обработки события для компонента Traverse1
   else
      // Альтернативный код
}
```

Обработка события с дополнительными параметрами

Стандартное событие определяется так, как показано ниже.

```
typedef void __fastcall (__closure *TNotifyEvent)(
                                    System::TObject *Sender)
```

А пользовательское событие определяется немного иначе.

```
typedef void __fastcall (__closure *TDirChangedEvent)(
                                    System::TObject *Sender,
                                    bool &Abort)
```

В предыдущем коде сделано следующее:

- создан уникальный тип, т.е. тип `TNotifyEvent` теперь называется `TDirChangedEvent`;
- в список параметров добавлен еще один параметр.

Теперь определение класса будет выглядеть так, как показано в листинге 9.19.

Листинг 9.19. Код пользовательского события-свойства

```
typedef void __fastcall (__closure *TDirChangedEvent)(
                                    System::TObject *Sender,
                                    bool &Abort)
class PACKAGE TTraverseDir : public TComponent
{
private:
   TDirChangedEvent *FOnDirChanged;

__published:
   __property TDirChangedEvent OnDirChanged =
                                  {read = FOnDirChanged,
                                   write = FOnDirChanged};
}
```

Теперь, после создания этого события, IDE-среда автоматически добавит в код компонента следующий каркас.

```
void __fastcall TTraverseDir::Traverse1DirChanged(
                                TObject *Sender, bool &Abort)

{
}
```

Теперь, как показано в листинге 9.20, остается внести только одно изменение в исходный код, организовав вызов этого события.

Листинг 9.20. Вызов события

```
void __fastcall TTraverseDir::Execute(void)
{
   // Обход каталогов.

   bool &Abort = false;

   // Здесь меняется текущий каталог,
   // происходит вызов обработчика события DirChanged,
   // если он задан.
   if(FOnDirChanged)
      FOnDirChanged(this, Abort);

   if(Abort)
      // Обработка принудительного прекращения
      // работы компонента.

   // Остальной код компонента.
}
```

Этот код компонент существенно изменен, чтобы пользователь в случае необходимости мог прервать его выполнение.

Методы

Методами компонента называются вспомогательные процедуры, предназначенные для выполнения специальных задач. Принципиально они ничем не отличаются от методов класса. Создавая компоненты, разработчик задается целью сократить до минимума количество вызываемых в приложении методов. Ниже перечислено несколько простых правил, которых следует придерживаться при создании компонентов.

- Пользователь не должен вызывать методы, необходимые для нормальной работы компонента. Например, компонент сам должен позаботиться об инициализации всех переменных.
- Не должно быть никакой зависимости от порядка вызова методов. Компонент нужно спроектировать так, чтобы была учтена любая комбинация событий. Если пользователь вызывает зависящую от состояния процедуру (например, запрос к базе данных при отсутствии активного подключения к ней), то компонент должен адекватно обработать данную ситуацию. В этом случае потребуется либо подключиться к базе данных, либо инициировать соответствующую исключительную ситуацию на основе функции компонента.
- Пользователь не должен вызывать метод, который может изменить состояние компонента во время выполнения другой задачи.

Наилучшим способом решения этих проблем является создание собственных методов, которые смогут проверить текущее состояние компонента. Если все эти требования не будут выполняться, компонент должен попытаться исправить проблему. Спроектируйте компонент так, чтобы он был способен инициировать исключительную ситуацию, если его состояние не может быть исправлено. Рекомендуется создать пользовательские исключительные ситуации так, чтобы пользователь имел возможность обнаружить специфические условия их возникновения. Более подробно этот вопрос рассматривается в следующем разделе.

Вместо методов часто рекомендуется создавать свойства. Свойства позволяют скрывать реализацию от пользователя и, следовательно, способствуют созданию интуитивно понятного компонента. Например, компонент для работы с базой данных может иметь свойство `Active` и эквивалентные ему методы `Open()` и `Close()`. Рассмотрим приведенные ниже две эквивалентные по смыслу (база данных открыта) строки кода.

```
Database1->Active = true;
Database1->Open();
```

Аналогично эквивалентными по смыслу (база данных закрыта) являются и эти две строки кода.

```
Database1->Active = false;
Database1->Close();
```

По мнению автора, свойство `Active` следует сделать доступным для пользователя. Читатели могут возразить, что методы `Open()` и `Close()` лучше описывают текущее состояние компонента, но автор считает их излишними, поскольку оба они могут быть представлены всего одним свойством `Active`.

Методы компонентов обычно бывают открытыми или защищенными. Закрытые методы следует создавать только в особых случаях для внутренних целей компонента, а точнее, чтобы даже производные компоненты не использовали их.

Открытые методы

Открытыми (*public*) называются методы, необходимое для выполнения компонентом его функций. При этом важно убедиться в том, что они достаточно эффективны и не расходуют чрезмерно много ресурсов операционной системы. Если это неизбежно, рассмотрите возможность создания события (или функции обратного вызова), которое разработчик мог бы использовать для передачи пользователю сведений о выполняемых в данный момент действиях. Еще один способ заключается в предоставлении пользователю возможности прекратить работу компонента (с помощью передачи ссылки на событие или использования другого свойства).

Представьте себе компонент, который выполняет поиск файла в иерархической структуре каталогов. В зависимости от типа операционной системы этот процесс может продолжаться очень долго. Ожидая завершения процесса, пользователь может подумать, что приложение вообще прекратило работу и зависло. Поэтому лучше создать событие, которое будет возникать внутри метода. Именно это событие можно затем использовать для организации обратной связи с пользователем, например, сообщая ему имя просматриваемого в данный момент каталога.

Защищенные методы

Если компонент содержит методы, которые должны быть недоступны для разработчика приложений, но могут использоваться в производных компонентах, то эти методы следует объявить *защищенными* (*protected*). Это предотвратит вызов такого метода в неподходящем месте.

Если метод создается для реализации свойств, то его следует объявить *виртуальным защищенным* (*virtual protected*). Это позволяет расширить или переопределить реализацию этого метода.

Примером такого метода является виртуальный защищенный метод `Loaded()`, который вызывается после полной загрузки компонента (с помощью потокового вывода из формы).

В некоторых случаях компонент-наследник для выполнения каких-либо дополнительных задач должен “знать” о загрузке компонента и считывании значений всех свойств. Примером может служить компонент, который проверяет значение свойства, но не может это сделать до тех пор, пока не будут прочитаны значения всех свойств. В этом случае следует создать закрытую переменную `IsLoaded` и задать для нее значение `false` в конструкторе. (Хотя это

значение задается по умолчанию, явное кодирование такого присвоения позволяет повысить читабельность кода.) Затем нужно перегрузить метод Loaded() и задать значение true для переменной IsLoaded. После этого переменная может быть использована в методах реализации свойств для выполнения необходимой проверки.

В листингах 9.21 и 9.22 приведен код пользовательского компонента TAliasComboBox, являющийся частью бесплатного пакета MJFPack, который можно скопировать по адресу http://www.mjfreelancing.com. Этот пакет содержит другие компоненты, которые описанным выше способом могут быть связаны вместе.

Листинг 9.21. Заголовочный файл компонента TAliasComboBox

```
class PACKAGE TAliasComboBox : public TSmartComboBox
{
private:
   bool IsLoaded;

protected:
   virtual void __fastcall Loaded(void);
}
```

Листинг 9.22. Исходный код компонента TAliasComboBox

```
void __fastcall TAliasComboBox::Loaded(void)
{
   TComponent::Loaded();

   if(!ComponentState.Contains(csDesigning))
   {
      IsLoaded = true;
      GetAliases();
   }
}
```

В этом коде видно, что метод Loaded() перегружен в объявлении класса. В файле .CPP сначала вызывается прежний метод Loaded(), а потом вводится дополнительный код. В листинге 9.22 показано, что код этого компонента извлечет сведения о псевдонимах только в режиме выполнения приложения. Так как состояние некоторых переменных может зависеть от других свойств, то дополнительные методы этого компонента проверят значение переменной IsLoaded перед выполнением обработки, для которой могут потребоваться определенные значения этих свойств. При этом большая часть кода компонента будет функционировать только во время выполнения приложения.

Обработка исключительных ситуаций в коде компонента

Иногда обработку исключительной ситуации, возникшей в компоненте, можно передать дальше, что позволяет пользователю справиться с этой ситуацией. При возникновении исключительной ситуации программисту почти наверняка придется выполнить несколько важных операций очистки, а потом проделать следующее.

Во-первых, можно повторно инициировать исключительную ситуацию. Это довольно стандартная процедура для такой, например, исключительной ситуации, как Divide By Zero (Деление на ноль). Однако возможны ситуации, когда исключительную ситуацию лучше преобразовать в событие. Это позволяет предоставить пользователям более четкие методы управления этой ситуацией. Однако не следует преобразовывать все исключительные ситуации в события, так как это может затруднить использование таких компонентов при создании другими пользователями их собственных приложений.

Чтобы прояснить эту ситуацию, рассмотрим следующий пример. Представьте себе компонент, который последовательно выполняет несколько запросов к базе данных. Этот компонент мог бы содержать свойство TStrings со всеми запросами и метод Execute(), который выполняет их. Как пользователь мог бы использовать этот компонент? Вероятно, для этого он применил бы следующий код.

```
MultiQuery->Queries->Assign(Memo1->Lines);
MultiQuery1->Execute();
```

Такой простой код очень просто реализовать, но как в таком случае быть с возможными исключительными ситуациями? Должен ли пользователь самостоятельно обрабатывать исключительные ситуации? Это не совсем удачный способ. Лучше создать событие, которое бы вызывалось при возникновении исключительной ситуации. В этом событии пользователь должен иметь возможность принудительно прекратить процесс.

Давайте создадим пользовательский обработчик исключительной ситуации, вызываемый при попытке пользователя выполнить запрос, который выходит за рамки допустимого диапазона индексов. Здесь мы предположим, что имеется другой метод ExecuteItem(), который принимает в качестве параметра индекс для списка имеющихся запросов.

Во-первых, исключительную ситуацию нужно определить в заголовочном файле. Для этого нужно создать новый класс исключительной ситуации, производный от класса Exception, например, так, как показано в листинге 9.23.

Листинг 9.23. A Custom Exception Class

```
class EMultiQueryIndexOutOfBounds : public Exception
{
public:
  __fastcall EMultiQueryIndexOutOfBounds(const AnsiString Msg) :
    Exception(Msg){}
};
```

Вот и все. Теперь при попытке пользователя выполнить запрос, индекс которого выходит за рамки допустимого диапазона, возникает исключительная ситуация.

Код инициирования этой ситуации показан в листинге 9.24.

Листинг 9.24. Код пользовательской ситуации

```
void __fastcall TMultiQuery::ExecuteItem(int Index)
{
   if(Index < 0 || Index >Queries->Count)
      throw EmultiQueryIndexOutOfBounds;

   //... Код запроса
}
```

Как показано в листингах 9.23 и 9.24, пользовательская исключительная ситуация может быть достаточно просто создана и реализована. Если этот компонент должен выполнять запрос в процессе создания приложения, для пользователя должно отображаться соответствующее сообщение (а не генерироваться исключительная ситуация, инициируемая в IDE-среде). Для этого код нужно отредактировать так, как показано в листинге 9.25.

Листинг 9.25. Инициирование исключительной ситуации во время создания приложения

```
void __fastcall TMultiQuery::ExecuteItem(int Index)
{
   if(Index < 0 || Index > Queries->Count)
   {
      if(ComponentState.Contains(csDesigning))
         throw EmultiQueryIndexOutOfBounds(
                  "The Query index is out of range");
//       throw EmultiQueryIndexOutOfBounds(
//                  "Индекс запроса выходит за рамки допустимого
//                   диапазона");
      else
         throw EmultiQueryIndexOutOfBounds;
   }

   //... Код запроса
}
```

Ключевое слово namespace

При создании компонентов и присвоении им имен следует учитывать, что другие разработчики могут использовать такие же имена. Это может привести к возникновению конфликтов при использовании таких компонентов в одном проекте. Именно для решения этой проблемы и предназначено ключевое слово namespace.

При создании компонента с помощью программы-мастера создания компонентов ***New Component Wizard*** IDE-среда создает код, аналогичный показанному в коде из листинга 9.26.

Листинг 9.26. Код с использованием ключевого слова namespace

```
namespace Aliascombobox
{
   void __fastcall PACKAGE Register()
   {
      TComponentClass classes[1] == {__classid(TAliasComboBox)};
      RegisterComponents("MJF Pack", classes, 0);
   }
}
```

Ключевое слово namespace гарантирует, что компонент будет создан в собственной подсистеме. Рассмотрим теперь случай, когда ключевое слово namespace нужно использовать даже внутри пакета.

Предположим, что два разработчика создали компонент-часы с постоянной, которая обозначает принимаемый по умолчанию формат времени. При использовании в одном приложении обоих этих компонентов компилятор сообщит о дублировании имен.

```
// Компонент первого разработчика.
const bool Mode12; // 12 часовое представление времени
class PACKAGE TClock1 : public TComponent
{
}
// Компонент второго разработчика.
const bool Mode12; // 12 часовое представление времени
class PACKAGE TClock2 : public TComponent
{
}
```

Как видите, при создании пакетов компонентов следует иметь в виду возможность возникновения такого конфликта имен. Для решения этой проблемы можно использовать ключевое слово `namespace`. Для этого весь код после выражений `#include` в заголовочном файле следует заключить в скобки с указанием пространства имен, как показано на рис. 9.27.

Листинг 9.27. Указание пространства имен

```
namespace NClock1
{
   class PACKAGE TClock1 : public TComponent
   {
   }
}
```

Для всех создаваемых компонентов рекомендуется использовать соглашение об именах. Например, название пространства имен следует начинать с большой буквы *N*, за которой следует имя компонента. Если существует вероятность повторного использования такого же имени, то рекомендуется использовать какой-либо уникальный элемент, например ввести в качестве префикса аббревиатуру фирмы. Использование пространств имен гарантирует гладкую интеграцию ваших пакетов с пакетами других разработчиков.

Отклик на сообщения

Библиотека VCL способна выполнять огромную работу по обработке практически всех оконных сообщений, которые только могут потребоваться. Однако может возникнуть такая ситуация, когда для усовершенствования проекта необходимо организовать отклик на некоторое дополнительное событие.

Примером может служить поддержка операции перетаскивания имени файла из окна программы Windows Explorer в строковый компонент Grid. Создадим для этого компонент `TSuperStringGrid`, который является наследником компонента `TStringGrid`.

Операция перетаскивания управляется API-сообщением `WM_DROPFILES`. Вся необходимая для перетаскивания информация хранится в структуре `TWMDropFiles`.

Перехват оконных сообщений в компонентах выполняется так же, как и в других случаях. Единственное отличие заключается в том, что работа ведется с компонентом, а не с формой проекта. Следовательно, карту сообщений (message map) следует задать так, как показано в листинге 9.28.

Листинг 9.28. Перехват сообщений

```
BEGIN_MESSAGE_MAP
   MESSAGE_HANDLER(WM_DROPFILES, TWMDropFiles, WmDropFiles)
END_MESSAGE_MAP(TStringGrid)
```

На заметку

В объявлении карты сообщений точка с запятой не используется. Дело в том, что BEGIN_MESSAGE_MAP, MESSAGE_HANDLER и END_MESSAGE_MAP — это макросы, которые при компиляции заменяются кодом. Код этих макросов содержит все необходимые точки с запятой.

Код в листинге 9.28 создает карту сообщений для компонента (обратите внимание на компонент TStringGrid в макросе END_MESSAGE_MAP). Обработчик сообщений передаст все перехваченные сообщения WM_DROPFILES методу WmDropFiles() (который описывается ниже). Эта информация передается в структуре TWMDropFiles, которая определена в Windows.

Рассмотрим теперь метод обработки такого сообщения. В раздел защищенных элементов компонента следует вставить следующий код.

```
protected:
   void __fastcall WmDropFiles(TWMDropFiles &Message);
```

Обратите внимание, что в качестве параметра метода передается ссылка на структуру.

Перед использованием компонента его необходимо зарегистрировать в Windows, сообщая компоненту Grid о том, что он должен включать перетаскиваемые имена файлов. Эта задача выполняется с помощью функции DragAcceptFiles().

```
DragAcceptFiles(Handle, FCanDropFiles);
```

В предыдущей строке кода переменная FCanDropFiles используется компонентом для обозначения возможности включения имен файлов в операцию перетаскивания.

Теперь метод способен включать имена файлов при перехвате компонентом заданного сообщения Windows. В листинге 9.29 показан немного сокращенный код этой операции.

Листинг 9.29. Код включения в компонент перетаскиваемых имен файлов

```
void __fastcall TSuperStringGrid::WmDropFiles(TWMDropFiles &Message)
{
   char buff[MAX_PATH];
   HDROP hDrop = (HDROP)Message.Drop;
   POINT Point;
   int NumFiles = DragQueryFile(hDrop, -1, NULL, NULL);
   TStringList *DFiles = new TStringList;
   DFiles->Clear();
   DragQueryPoint(hDrop, &Point);
   for(int i = 0; i < NumFiles; i++)
   {
      DragQueryFile(hDrop, i, buff, sizeof(buff));
      DFiles->Add(buff);
   }
   DragFinish(hDrop);

   // Какие-то действия с файлами в списке DFiles

   delete DFiles;
}
```

Объяснение кода выходит за рамки этой главы, но подробное описание всех этих функций можно найти в справке C++Builder.

Как видите, обладая необходимыми знаниями о принципах перехвата сообщений и работы с Windows API-функциями, можно организовать перехват сообщений. Список всех имеющихся структур сообщений можно найти в файле `messages.hpp`.

Работа компонента при создании и выполнении приложения

Выше уже сравнивалась работа компонента во время создания и выполнения приложения. *Работа компонента при создании приложения* означает способ функционирования компонента при создании проекта в IDE-среде. А *работа компонента при выполнении приложения* означает способ функционирования компонента при запуске готового приложения.

Объект `TComponent` имеет свойство (типа `Set`) `ComponentState`, которое состоит из следующих констант: `csAncestor`, `csDesigning`, `csDesignInstance`, `csDestroying`, `csFixups`, `csFreeNotification`, `csInline`, `csLoading`, `csReading`, `csWriting` и `csUpdating`. В табл. 9.1 перечислены флаги структуры `ComponentState` с описанием их назначения.

Таблица 9.1. Флаги структуры `ComponentState`

Флаг	Назначение
`csAncestor`	Означает, что компонент введен в форме-предке. Задается, только если задан флаг `csDesigning`. Он задается или снимается с помощью метода `TComponent::SetAncestor()`
`csDesigning`	Означает, что компонент используется при создании приложения. Используется для обозначения текущего режима использования, т.е. во время создания или выполнения приложения. Он задается или снимается с помощью метода `TComponent::SetDesigning()`
`CsDesignInstance`	Означает, что компонент является корневым объектом. Например, он задается для создаваемого фрэйма, но не для фрэйма, который действует как компонент. Этот флаг всегда используется вместе с флагом `csDesigning`. Он задается или снимается с помощью метода `TComponent::SetDesignInstance()`. Впервые представлен в C++Builder 5
`csDestroying`	Означает удаление компонента. Он задается или снимается с помощью метода `TComponent::Destroying()`.
`csFixups`	Означает, что компонент связан с компонентом в другой, еще не загруженной форме. Этот флаг снимается после разрешения связей с помощью глобального метода `GlobalFixupReferences()`.
`csFreeNotification`	Означает, что компонент послал уведомления в другие формы о том, что он удаляется, но еще не удален. Задается или снимается с помощью метода `TComponent::FreeNotification()`. Впервые представлен в C++Builder 5
`csInline`	Означает, что компонент является компонентом верхнего уровня, который может быть изменен при выполнении приложения, а также включен в форму. Этот флаг используется для идентификации вложенных фрэймов во время их загрузки и сохранения. Он задается или снимается с помощью метода компонента `SetInline()`, а также метода `TReader::ReadComponent()`. Впервые представлен в C++Builder 5

Окончание табл. 9.2

Флаг	Назначение
CsLoading	Означает, что объект-файлер в момент загружает объект. Этот флаг задается при первой попытке создания компонента и не снимается до тех пор, пока не будут полностью загружены компонент и все его дочерние элементы (при вызове метода Loaded()). Задается с помощью методов TReader::ReadComponent() и TReader::ReadRootComponent(), а снимается с помощью метода TComponent::Loaded(). (Более подробные сведения об объектах-файлерах можно найти в разделе "TFiler" в интерактивной справке C++Builder.)
csReading	Означает, что компонент считывает значения свойств из потока. Обратите внимание: флаг csLoading всегда задается, если задан флаг csReading. То есть, флаг csReading задается на время чтения компонентом значений свойств при его загрузке. Задается и снимается с помощью методов TReader::ReadComponent() и Treader::ReadRootComponent()
csWriting	Означает, что данный компонент записывает значения свойств в поток. Задается и снимается с помощью метода TWriter::WriteComponent()
csUpdating	Означает, что компонент обновляется для учета изменений в форме-предке. Задается, только если задан флаг csAncestor. Задается с помощью метода TComponent::Updating(), снимается с помощью TComponent::Updated()

Наиболее интересным элементом этого набора здесь является флаг csDesigning. До тех пор, пока компонент существует в IDE-среде (как часть проекта), он будет содержать эту константу как часть этого набора для обозначения текущего использования его в режиме создания приложения. Для определения этого режима можно использовать приведенный ниже код.

```
if(ComponentState.Contains(csDesigning))
   // Здесь располагается код,
   // используемый при создании приложения.
else
   // Здесь располагается код,
   // используемый при выполнении приложения.
```

Почему некоторый код следует использовать только при выполнении приложения? Вот несколько причин для такого разделения кода:

- для указания свойства, которое используется только при выполнении приложения;
- для отображения предупредительного сообщения для пользователя, если он указал неправильное значение свойства;
- для отображения диалогового окна с выбором вариантов или редактора свойств, если введено неверное значение свойства.

Многие создатели компонентов не предоставляют пользователям такие возможности выбора типов предупреждений и диалоговых окон. Однако именно эти дополнительные элементы делают компонент более дружественным и интуитивно понятным пользователю.

Связывание компонентов

Связывание компонентов (*linking components*) означает способность компонента ссылаться на другой компонент или изменять компонент проекта. Например в C++Builder предусмотрен компонент TDriveComboBox со свойством DirList, который позволяет разработчику выбрать компонент TDirectoryListBox в той же форме. Этот тип связывания предоставляет

простой и быстрый метод автоматического обновления списка каталогов при каждом переходе к другому жесткому диску. Создание проекта для отображения списка каталогов и имен файлов заключается в размещении трех компонентов (TDriveComboBox, TDirectoryListBox и TFileListBox) в форме и указании значений для двух свойств. При этом, конечно, придется создать код обработчиков событий для выполнения некоторых действий, имеющих смысл и пользу для данного проекта. Но до этого момента разработчику не придется специально создавать ни единой строки кода.

Организация связи с другими компонентами начинается с создания свойства заданного типа. При создании свойства типа TLabel инспектор объектов Object Inspector покажет все доступные компоненты формы типа TLabel. Создадим простой компонент, который связан с компонентом TMemo или TRichEdit. При этом необходимо учитывать, что оба эти компонента являются наследниками компонента TCustomMemo.

Начнем с создания компонента-наследника для компонента TComponent, который имеет свойство LinkedEdit (листинг 9.30).

Листинг 9.30. Связанные компоненты.

```
class PACKAGE TMsgLog : public TComponent
{
private:
   TCustomMemo *FLinkedEdit;
   // TMemo или TRichEdit, или любой другой производный компонент

public:
   __fastcall TMsgLog(Tcomponent *Owner);
   __fastcall ~TMsgLog(void);
   void __fastcall OutputMsg(const AnsiString Message);

protected:
   virtual void __fastcall Notification(TComponent *AComponent,
                                        TOperation Operation);

__published:
   __property TCustomMemo *LinkedEdit = {read = FLinkedEdit,
                                         write = FLinkedEdit};
};
```

Код в листинге 9.30 создает компонент с одним свойством LinkedEdit. При этом следует иметь в виду следующее. Во-первых, сообщения нужно передавать связанному компоненту типа Memo или RichEdit (если таковые существуют). Кроме того, пользователь должен иметь возможность удалить связанное поле редактирования. Для передачи текстового сообщения в связанное поле редактирования используется метод OutputMsg(), а для уведомления о его удалении — метод Notification().

Для вывода сообщения используется следующий код.

```
void __fastcall TMsgLog::OutputMsg(const AnsiString Message)
{
   if(FLinkedEdit)
      FLinkedEdit->Lines->Add(Message);
}
```

Так как компонент TMemo и компонент TRichEdit имеют свойство Lines, то нет необходимости в явном приведении типов. Для выполнения специальной для данного компонента задачи (или специализированной обработки) следует использовать код, приведенный в листинге 9.31.

Листинг 9.31. Метод `OutputMsg()`

```
void __fastcall TMsgLog::OutputMsg(const AnsiString Message)
{
   TMemo *LinkedMemo = 0;
   TRichEdit *LinkedRichEdit = 0;

   LinkedMemo = dynamic_cast<TMemo *>(FLinkedEdit);
   LinkedRichEdit = dynamic_cast<TRichEdit *>(FLinkedEdit);

   if(FLinkedMemo)
      FLinkedMemo->Lines->Add(Message);
   else
   {
      FLinkedRichEdit->Font->Color = clRed;
      FLinkedRichEdit->Lines->Add(Message);
   }
}
```

Заключительная проверка удаления связанного поля редактирования выполняется с помощью перегрузки метода Notification() компонента TComponent, как показано в листинге 9.32.

Листинг 9.32. Метод `Notification()`

```
void __fastcall TMsgLog::Notification(TComponent *AComponent,
                                      TOperation Operation)
{
   // Добавленные элементы управления игнорируются.
   if(Operation != opRemove)
      return ;
   // Нужно проверять при каждом действии пользователя,
   // например, когда одна и та же надпись относится
   // к нескольким свойствам.
   if(AComponent == FLinkedEdit)
      FLinkedEdit = 0;
}
```

В листинге 9.32 показан способ обработки кода из другого удаленного компонента. В первых двух строках показано назначение параметра Operation.

Наиболее важная часть кода заключена в последних двух строках, где указатель AComponent сравнивается со свойством LinkedEdit (указатель на компонент-наследник компонента TCustomMemo). Если указатели совпадают, то указателю LinkedEdit присваивается значение NULL. Это позволяет удалить ссылку из окна Object Inspector и гарантирует, что код больше не указывает на адрес в памяти, которая уже утрачена вследствие фактического удаления компонента редактирования. Обратите внимание, что выражение LinkedEdit = 0 равносильно выражению LinkedEdit = NULL.

При связи двух компонентов, один из которых имеет зависимые компоненты (например, компоненты-наследники TDBDataSet, подключаемые к базе данных), разработчику придется позаботиться о проверке и соответствующей обработке этих зависимостей. Профессиональное проектирование компонентов означает сокращенный до минимума объем работы, который необходимо выполнить для должного поведения компонентов.

Связь событий из разных компонентов

До сих пор мы рассматривали связь между компонентами на основе событий. Например, свойство TMsgLog связано с другим компонентом таким образом, что обмен сообщениями выполняется автоматически без необходимости создания дополнительного кода.

В этом разделе рассматриваются способы связи событий между компонентами. Здесь будет показан способ перехвата события OnExit для связанного поля редактирования. (Учтите, что оба элемента управления, TMemo и TRichEdit, имеют свойство OnExit типа TNotifyEvent.) Это необходимо для дополнительной обработки после выполнения пользовательского кода. Предположим, что связанное поле редактирования используется не только в режиме для чтения. Это значит, что пользователь может вводить в него данные, которые должны быть зафиксированы как внесенные пользователем изменения. Рассмотрим способ перехвата такого события, а создание кода его дальнейшей обработки предоставим читателю.

События компонентов могут быть реализованы по-разному, в зависимости от природы события. Если компонент циклически используется в процессе, то он должен содержать код вызова обработчика события, если это указано. Рассмотрим следующий пример.

```
// Начало цикла.
if(FOnExit)
   FOnExit(this);
endif;
// ...
// Конец цикла.
```

Другие события могут возникнуть в результате получения какого-либо сообщения. В листинге 9.26 показан макрос для карты сообщений, предназначенной для приема имен файлов, которые перетащены в элемент управления из проводника Windows Explorer.

```
BEGIN_MESSAGE_MAP
   MESSAGE_HANDLER(WM_DROPFILES, TWMDropFiles, WmDropFiles)
END_MESSAGE_MAP(TStringGrid)
```

Если компонент имеет событие OnDrop, то он может быть реализован следующим образом.

```
void __fastcall TSuperStringGrid::WmDropFiles(
                                    TWMDropFiles &Message)
{
   if(FOnDrop)
      FOnDrop(this);
   endif;

   // ... Какой-то другой код.
}
```

Как видите компоненты могут иметь такой указатель на обработчик событий, как, например, FOnExit и FOnDrop из предыдущего примера. Это существенно упрощает создание указателя для обозначения пользовательского обработчика и перенаправления обработки пользо-

вательского события с вызовом внутреннего метода. Этот внутренний метод выполнит исходный код пользователя, а затем код самого компонента (или наоборот).

Особо важное значение имеет время перенаправления указателей. Логично было бы выполнить его в методе `Loaded()` компонента. Он вызывается при передаче в потоке всего компонента из формы, а следовательно, указываются все обработчики пользовательского события.

Для этого нужно определить метод `Loaded()` и указатель на стандартное событие в этом классе. (Это событие имеет тот же тип, что и перехватываемое событие. В нашем случае это событие `OnExit`, которое имеет тип `TNotifyEvent`.) Нужно также создать внутренний метод с тем же объявлением, что и перехватываемое событие. В нашем классе создадим метод `MsgLogOnExit`, который будет вызван до события `OnExit` связанного поля редактирования. В листинге 9.33 показано определение нового названия `Inherited` для типа `TComponent` с помощью ключевого слова `typedef`. Необходимость введения такого типа станет понятной после перехода к исходному коду.

Листинг 9.33. Заголовочный файл класса `TMsgLog`

```
class PACKAGE TMsgLog : public TComponent
{
   typedef TComponent Inherited;
private:
   TNotifyEvent *FonUsersExit;
   void __fastcall MsgLogOnExit(TObject *Sender);
protected:
   virtual void __fastcall Loaded(void);
   //... Какой-то другой код.
}
```

В исходном коде компонента может содержаться фрагмент, показанный в листинге 9.34.

Листинг 9.34. Исходный код класса `TMsgLog`

```
void __fastcall TMsgLog::TMsgLog(TComponent *Owner)
{
   FOnUsersExit = 0;
}

void __fastcall TMsgLog::Loaded(void)
{
   Inherited::Loaded();
   if(!ComponentState.Contains(csDesigning))
   {
      if(FlinkedEdit)
      {
         if(FlinkedEdit->OnExit)
         FOnUsersExit = FlinkedEdit->OnExit;
         FlinkedEdit->OnExit = MsgLogOnExit;
      }
   }
}

void __fastcall TMsgLog::MsgLogOnExit(TObject *Sender)
```

```
{
   if(FOnUsersExit)
      FOnUsersExit(this);

   // ... теперь выполняется некий дополнительный код.
}
```

При первой попытке создания компонента конструктор инициализирует переменную `FOnUsersExit` значением `NULL`. При полной загрузке формы с помощью потока вызывается событие `OnLoaded`. При этом сначала вызывается унаследованный метод (определение нового типа с помощью ключевого слова `typedef` улучшает читабельность кода). Далее следует проверить, не находится ли приложение в режиме создания. Если оно находится в режиме выполнения, следует проверить наличие связанного поля редактирования. Если это так, то следует найти пользовательский метод, связанный с событием `OnExit` этого элемента управления. Если эти тесты выполнены успешно, следует задать для указателя `FOnUsersExit` адрес пользовательского обработчика событий. Наконец, следует перенаправить обработчик события поля редактирования на внутренний метод `MsgLogOnExit()`. Это приведет к вызову метода `MsgLogOnExit()` при каждом выходе курсора за пределы поля редактирования, даже если пользователь не присвоил никакого обработчика события.

Метод `MsgLogOnExit()` начинается с поиска обработчика события. Если обработчик имеется, он будет немедленно вызван. Затем будет продолжено выполнение всех остальных задач. Необходимость вызова пользовательского события до или после вашего собственного кода определяется природой события, например необходимостью шифрования данных или проверки правильности ввода данных.

Создание визуальных компонентов

Как видите, компоненты могут быть любой частью программы, с которой взаимодействует разработчик. Компоненты могут быть невизуальными (как компонент-таблица `TTable`) или визуальными (как компонент-кнопка `TButton`). Наиболее очевидное различие между ними заключается в том, что визуальные компоненты имеют те же визуальные характеристики как во время создания, так и во время выполнения. При изменении свойств компонента, которые определяют его внешний вид, этот компонент должен быть перерисован или перекрашен с помощью окна Object Inspector для отражения этих изменений. Оконные элементы управления чаще всего являются всего лишь оболочками для универсальных элементов управления Windows (Common Controls), и операционная система Windows сама обычно заботится об их перерисовке. В некоторых ситуациях, когда некий компонент не связан с каким-либо существующим элементом управления, перерисовку компонента следует организовать самостоятельно. В любом случае будет полезно познакомиться с классами, которые предусмотрены в C++Builder для перерисовки экрана.

С чего начать

Один из наиболее важных аспектов создания компонентов заключается в выборе родительского класса. Для этого нужно еще раз просмотреть имеющиеся справочные файлы и исходный код библиотеки VCL. Для этого нельзя жалеть времени. Нет ничего более печального, чем потратить часы и даже дни на работу с компонентом и затем обнаружить, что он не обладает нужными качествами. При создании оконного компонента (который получает фокус

ввода и имеет дескриптор окна) сделайте его производным от классов TCustomControl или TWinControl. Если этот компонент чисто графический, как, например, кнопка TSpeedButton, то его следует сделать производным от класса TGraphicControl. Существует совсем мало (а порой их вообще нет) ограничений при создании визуальных компонентов. При этом в сети Internet предлагается огромное количество бесплатных и условно бесплатных компонентов и фрагментов исходного кода, которые можно использовать в работе.

Объект TCanvas

Объект TCanvas в C++Builder представляет собой оболочку для контекста устройства (Device Context). Он инкапсулирует разные инструменты рисования на экране сложных геометрических фигур и графических элементов. Доступ к объекту TCanvas в большинстве компонентов можно получить с помощью свойства Canvas. Хотя некоторые оконные элементы управления рисуются операционной системой Windows и потому не имеют свойства Canvas. В этом случае следует использовать другие методы, которые кратко рассматриваются ниже. Объект TCanvas также содержит несколько методов рисования на экране линий, геометрических фигур и сложных графических элементов.

В листинге 9.35 показан пример рисования линии по диагонали из верхнего левого угла в нижний правый угол канвы. Метод LineTo() рисует линию от текущей позиции пера до координат, указанных параметрами X и Y. Поэтому сначала следует указать начальное положение линии с помощью метода MoveTo().

Листинг 9.35. Рисование линии с помощью MoveTo()

```
Canvas->MoveTo(0,0);
int X = ClientRect.Right;
int Y = ClientRect.Bottom;
Canvas->LineTo (X,Y);
```

В листинге 9.36 для рисования в канве рамки, имеющей вид кнопки, используется метод Frame3D().

Листинг 9.36. Рисование рамки в виде кнопки

```
int PenWidth = 2;
TColor Top = clBtnHighlight;
TColor Bottom = clBtnShadow;
Frame3D(Canvas, ClientRect, Top, Bottom, PenWidth);
```

Для создания других эффектов в канве часто используются API-функции рисования. Некоторые из них используют контекст устройства (HDC) элемента управления, хотя для вызова API-функции рисования не всегда обязательно указывать контекст устройства (HDC) элемента управления. Для получения контекста устройства (HDC) элемента управления используется API-функция GetDC().

На заметку

Контекст устройства (HDC) элемента управления — это тип данных, который возвращается методом GetDC(), содержащим дескриптор контекста устройства DeviceContext. Он является синонимом свойства Handle объекта TCanvas.

```
HDC dc = GetDC(SomeComponent->Handle);
```

В листинге 9.37 используется форма с элементом управления TPaintBox (который используется потому, что свойство Canvas является опубликованным), а для рисования эллипса внутри него применяется API-функция RoundRect(). Элемент управления TPaintBox может располагаться в любом месте формы. Необходимый код рисования размещен в обработчике события OnPaint элемента управления TPaintBox. Полностью код этого проекта Project1.bpr можно найти в каталоге PaintBox1 на компакт-диске, который прилагается к этой книге.

Листинг 9.37. Использование API-функций рисования

```
void __fastcall TForm1::PaintBox1Paint(TObject *Sender)
{
   // Структура TRect используется
   // для сохранения введенных данных
   TRect Rect;
   int nLeftRect, nTopRect, nRightRect, nBottomRect,
      nWidth, nHeight;

   Rect = PaintBox1->ClientRect;
   nLeftRect = Rect.Left;
   nTopRect = Rect.Top;
   nRightRect = Rect.Right;
   nBottomRect = Rect.Bottom;
   nWidth = Rect.Right - Rect.Left;
   nHeight = Rect.Bottom - Rect.Top;

   if(RoundRect(
      PaintBox1->Canvas->Handle, // дескриптор контекста
                                 // устройства
      nLeftRect, // x-координата верхнего левого угла
                 // прямоугольника, ограничивающего рисуемый
                 // элемент управления
      nTopRect,  //y-координата верхнего левого угла
                 // прямоугольника, ограничивающего рисуемый
                 // элемент управления
      nRightRect, // x-координата нижнего правого угла
                  // прямоугольника, ограничивающего рисуемый
                  // элемент управления
      nBottomRect, // y-координата нижнего правого угла
                   // прямоугольника, ограничивающего рисуемый
                   // элемент управления
      nWidth, // ширина эллипса
      nHeight // высота эллипса
      ) == 0)
      ShowMessage("RoundRect failed...");
      // ShowMessage(
            "Нельзя нарисовать прямоугольник RoundRect...");
}
```

Попробуйте изменить значения переменных nWidth и nHeight. Начните с нулевых значений и вы обнаружите, что прямоугольник будет иметь четкие углы. При увеличении этих зна-

чений углы прямоугольника станут скругленными. Этот метод и некоторые другие можно использовать для создания скругленных и эллиптических кнопок или других элементов управления. Некоторые примеры создания таких объектов описываются ниже. Кроме того, более подробную информацию по этой теме можно найти в разделе “Painting and Drawing Functions” в справке по Win32, которая поставляется с C++Builder.

Использование в компонентах графических элементов

Графика в настоящее время стала неотъемлемой частью таких компонентов, как `TSpeedButton` и `TBitButton`. Она очень часто используется в бесплатных, условно бесплатных и коммерческих компонентах. Графика позволяет сделать компоненты более привлекательными, поэтому в C++Builder предусмотрено несколько классов для управления битовыми полями, пиктограммами, файлами в формате JPEG и GIF. Обычно рисование выполняется в битовом поле вне экрана, а затем это битовое поле целиком копируется в канву на экране. Это позволяет устранить мерцание экрана, так как канва перерисовывается только один раз. Это очень удобно при работе с изображением, состоящим из сложных графических элементов. Класс `TBitmap` имеет свойство `Canvas`, которое является синонимом объекта `TCanvas` и позволяет рисовать графические элементы в битовом поле вне экрана.

В следующем примере описывается форма с элементом управления `TPaintBox`. Сначала создается объект `TBitmap`, который используется для рисования изображения, аналогичного кнопке `TSpeedButton` со значением `true` свойства `Flat`. Затем битовое поле `TBitmap` с помощью одной операции полностью копируется на экран. В данном примере создается кнопка `TButton`, которая может изменять свой внешний вид и быть либо выступающей, либо утопленной. Полностью проект `Project1.bpr` этого примера находится в каталоге `PaintBox2` на прилагаемом к книге компакт-диске. Рассмотрим сначала заголовочный файл, код которого представлен в листинге 9.38.

Листинг 9.38. Создание кнопки, принимающей вид отжатой и утопленной

```
class TForm1 : public TForm
{
__published: // Компоненты IDE-среды
   TPaintBox *PaintBox1;
   TButton *Button1;
private:     // Объявления пользователя
   bool IsUp;
public:      // Объявления пользователя
   __fastcall TForm1(TComponent*Owner);
};
```

Логическая переменная `IsUp` используется для переключения яркой и темной окраски кнопки, а также для изменения надписи на кнопке. Если значение переменной `IsUp` равно `true`, то изображение кнопки будет в отжатом состоянии, а если значение переменной `IsUp` равно `false` — в утопленном. Так как переменная `IsUp` является переменной-членом, то в момент создания формы она будет инициализирована значением `false`. Для свойства `Caption` кнопки `Button1` можно задать значение `Up` с помощью окна Object Inspector.

Обработчик события OnClick кнопки имеет очень простую структуру. Он изменяет значение переменной IsUp и свойство Caption кнопки в соответствии с новым значением, а потом вызывает метод Repaint() компонента TPaintBox для перерисовки изображения. Код этого события показан в листинге 9.39.

Листинг 9.39. Метод Button1Click()

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
   IsUp = !IsUp;
   Button1->Caption = (IsUp) ? "Down" : "Up";
   PaintBox1->Repaint ();
}
```

Закрытый метод SwapColors() используется для изменения яркой и темной окраски кнопки на основе значения переменной IsUp. Код этого метода показан в листинге 9.40.

Листинг 9.40. Метод SwapColors()

```
void __fastcall TForm1::SwapColors(TColor &Top, TColor &Bottom)
{
   Top = (IsUp) ? clBtnHighlight : clBtnShadow;
   Bottom = (IsUp) ? clBtnShadow : clBtnHighlight;
}
```

На заключительном этапе следует создать обработчик события OnPaint кнопки TPaintBox. Его код показан в листинге 9.41.

Листинг 9.41. Код рисования кнопки

```
void __fastcall TForm1::PaintBox1Paint(TObject *Sender)
{
   TColor TopColor, BottomColor;
   TRect Rect;

   Rect = PaintBox1->ClientRect;

   Graphics::TBitmap *bit = new Graphics::TBitmap;
   bit->Width = PaintBox1->Width;
   bit->Height = PaintBox1->Height;
   bit->Canvas->Brush->Color = clBtnFace;
   bit->Canvas->FillRect(Rect);
   SwapColors(TopColor, BottomColor);
   Frame3D(bit->Canvas, Rect, TopColor, BottomColor, 2);
   PaintBox1->Canvas->Draw(0, 0, bit);
   delete bit;
}
```

В листинге 9.42 показан способ использования битовых файлов и рисования линий в форме. Большинство кнопок содержит не только линии и границы, которые образуют их форму, но

и пиктограммы, графические файлы и текст. Задача усложняется тем, что для загрузки графического файла потребуется создать второе битовое поле TBitmap, вычислить его расположение, скопировать в первое битовое поле, после чего скопировать в канву на экране. Полностью этот проект Project1.bpr находится в каталоге PaintBox3 на прилагаемом к книге компакт диске.

Листинг 9.42. Использование изображений и линий

```
void __fastcall TForm1::PaintBox1Paint(TObject *Sender)
{
   TColor TopColor, BottomColor;
   TRect Rect, gRect;

   Rect = PaintBox1->ClientRect;

   Graphics::TBitmap *bit = new Graphics::TBitmap;
   Graphics::TBitmap *bitFile = new Graphics::TBitmap;

   bitFile->LoadFromFile("geom1b.bmp");
   // Согласование размеров внеэкранного изображения
   // и размеров канвы на экране
   bit->Width = PaintBox1->Width;
   bit->Height = PaintBox1->Height;

   // Закрасить канву заданным цветом пера
   bit->Canvas->Brush->Color = clBtnFace;
   bit->Canvas->FillRect(Rect);

   // Расположение второго прямоугольника
   // по центру первого прямоугольника
   gRect.Left = ((Rect.Right - Rect.Left) / 2) -
                (bitFile->Width / 2);
   gRect.Top = ((Rect.Bottom - Rect.Top) / 2) -
               (bitFile->Height / 2);

   // Сдвиг внутреннего прямоугольника вверх и влево
   // для придания эффекта приподнятости панели
   gRect.Top += (IsUp) ? 0 : 1;
   gRect.Left += (IsUp) ? 0 : 1;

   gRect.Right = bitFile->Width + gRect.Left;;
   gRect.Bottom = bitFile->Height + gRect.Top;

   // Копирование изображения во внеэкранное битовое поле
   // с помощью эффекта прозрачности
   bit->Canvas->BrushCopy(gRect, bitFile,
      TRect(0, 0, bitFile->Width, bitFile->Height),
      bitFile->TransparentColor);

   // Рисование границ
   SwapColors(TopColor, BottomColor);
   Frame3D(bit->Canvas, Rect, TopColor, BottomColor, 2);
```

```
   // Копирование внеэкранного битового поля в канву на экране
   BitBlt(PaintBox1->Canvas->Handle, 0, 0,
          PaintBox1->ClientWidth,
          PaintBox1->ClientHeight,
          bit->Canvas->Handle, 0, 0, SRCCOPY);
   delete bitFile;
   delete bit;
}
```

Обработка сообщений мыши

Графические компоненты обычно являются производными от класса TGraphicControl, который содержит канву рисования и обрабатывает сообщения рисования WM_PAINT. Напомним, что неоконные компоненты не получают фокус ввода и не имеют оконного дескриптора. Хотя компоненты такого типа не получают фокус ввода, в библиотеке VCL предусмотрены пользовательские сообщения для перехвата и обработки событий мыши.

Например, при указании значения true для свойства Flat кнопки TSpeedButton эта кнопка приподнимется для отображения ее границ, если пользователь разместит указатель мыши на ней, и снова станет плоской при смещении указателя мыши за пределы этой кнопки. Этот эффект достигается обработкой двух сообщений CM_MOUSEENTER и CM_MOUSELEAVE так, как показано в листинге 9.43.

Листинг 9.43. Сообщения CM_MOUSEENTER и CM_MOUSELEAVE

```
void __fastcall CMMouseEnter(TMessage &Msg); //CM_MOUSEENTER
void __fastcall CMMouseLeave(TMessage &Msg); //CM_MOUSELEAVE

BEGIN_MESSAGE_MAP
   MESSAGE_HANDLER(CM_MOUSEENTER, TMessage, CMMouseEnter)
   MESSAGE_HANDLER(CM_MOUSELEAVE, TMessage, CMMouseLeave)
END_MESSAGE_MAP(TBaseComponentName)
```

Сообщение CM_ENABLEDCHANGED также имеет большое значение. Свойство Enabled элемента управления TGraphicControl объявлено как открытое, а метод задания его значения (setter method) просто передает элементу управления сообщение CM_ENABLECHANGED, чтобы выполнялись необходимые действия — отображался серым цветом текст или графика или отключалось событие. Для включения или отключения компонента это свойство можно объявить как опубликованное в заголовочном файле компонента, а также объявить метод и обработчик сообщения. Благодаря этому при выполнении приложения пользователи смогут присваивать значения true или false для свойства Enabled без нежелательных последствий. Сообщение CM_ENABLECHANGED показано в листинге 9.44.

Листинг 9.44. Сообщение CM_ENABLEDCHANGED

```
void __fastcall CMEnabledChanged(TMessage &Msg);
__published:
   __property Enabled ;

BEGIN_MESSAGE_MAP
   MESSAGE_HANDLER(CM_ENABLEDCHANGED,TMessage,CMEnabledChanged)
END_MESSAGE_MAP(TYourComponentName)
```

Другие события мыши, например OnMouseUp, OnMouseDown и OnMouseOver, объявлены в разделе защищенных членов класса TControl, поэтому разработчику потребуется только переопределить соответствующие методы. Напомним: для того чтобы наследники вашего компонента имели возможность переопределять эти события, их необходимо объявлять в разделе защищенных членов заголовочного файла данного компонента. Пример переопределения обработчиков событий мыши показаны в листинге 9.45.

Листинг 9.45. Переопределение обработчиков событий мыши класса TControl

```
private:
   TMmouseEvent FOnMouseUp;
   TMouseEvent FOnMouseDown;
   TMouseMoveEvent FOnMouseMove;

protected:
   void __fastcall MouseDown(TMouseButton Button,
                             TShiftState Shift,
                             int X, int Y);
   void __fastcall MouseMove(TshiftState Shift, int X, int Y);
   void __fastcall MouseUp(TMouseButton Button,
                           TShiftState Shift,
                           int X, int Y);

__published:
   __property TMouseEvent OnMouseUp = {read=FOnMouseUp,
                                       write=FOnMouseUp};
   __property TMouseEvent OnMouseDown = {read=FOnMouseDown,
                                         write=FOnMouseDown};
   __property TMouseMoveEvent OnMouseMove =
                                  {read=FOnMouseMove,
                                   write=FOnMouseMove};
```

В представленных примерах обработчик создавался для события OnPaint() элемента управления TPaintBox. Это событие возникает при получении им сообщения WM_PAINT. Компонент TGraphicControl перехватывает это сообщение и предоставляет виртуальный метод Paint(), который может быть переопределен в компонентах-наследниках для рисования элемента управления на экране так же, как это делается в компоненте TPaintBox с помощью обработчика события OnPaint().

Эти и другие сообщения определяются в файле messages.hpp. При наличии исходного кода для компонентов библиотеки VCL рекомендуется потратить некоторое время на изучение имеющихся сообщений или событий, а также доступных переопределяемых методов.

Комбинированный подход

В этом разделе описывается способ использования всех перечисленных выше методов в базовом компоненте, функциональность которого предполагается расширить и усовершенствовать. Хотя этот компонент не совсем закончен, его можно разместить в палитре компонентов C++Builder и использовать в других приложениях. При создании компонентов следует иметь в виду, что чем проще ваш компонент, тем выше вероятность, что он кому-то пригодится и будет использован. В листингах 9.46 и 9.47 приводится пример компонента Button,

который имеет внешний вид кнопки TSpeedButton с изображением и текстом. Исходный код этого компонента показан в листингах 9.46 и 9.47, а комментарии по поводу наиболее очевидных усовершенствований приводятся ниже. Полный исходный код этого примера находится в каталоге ExampleButton на прилагаемом к книге компакт-диске.

Листинг 9.46. Заголовочный файл ExampleButton.h для кнопки TExampleButton

```
//---------------------------------------------------------------
#ifndef ExampleButtonH
#define ExampleButtonH
//---------------------------------------------------------------
#include <SysUtils.hpp>
#include <Controls.hpp>
#include <Classes.hpp>
#include <Forms.hpp>
//---------------------------------------------------------------

enum TExButtonState {esUp, esDown, esFlat, esDisabled};

class PACKAGE TExampleButton : public TGraphicControl
{
private:
   Graphics::TBitmap *FGlyph;
   AnsiString FCaption;
   TImageList *FImage;
   TExButtonState FState;
   bool FMouseInControl;
   TNotifyEvent FOnClick;
   void __fastcall SetGlyph(Graphics::TBitmap *Value);
   void __fastcall SetCaption(AnsiString Value);
   void __fastcall BeforeDestruction(void);
   void __fastcall SwapColors(TColor &Top, TColor &Bottom);
   void __fastcall CalcGlyphLayout(TRect &r);
   void __fastcall CalcTextLayout(TRect &r);
   MESSAGE void __fastcall CMMouseEnter(TMessage &Msg);
   MESSAGE void __fastcall CMMouseLeave(TMessage &Msg);
   MESSAGE void __fastcall CMEnabledChanged(TMessage &Msg);

protected:
   void __fastcall Paint(void);
   void __fastcall MouseDown(TMouseButton Button,
                             TShiftState Shift,
                             int X, int Y);
   void __fastcall MouseUp(TMouseButton Button,
                           TShiftState Shift,
                           int X, int Y);

public:
   __fastcall TExampleButton(TComponent*Owner);

__published:
```

```
   __property AnsiString Caption =
                           {read=FCaption, write=SetCaption};
   __property Graphics::TBitmap *Glyph =
                           {read=FGlyph, write=SetGlyph};
   __property TNotifyEvent OnClick =
                           {read=FOnClick, write=FOnClick};

BEGIN_MESSAGE_MAP
   MESSAGE_HANDLER(CM_MOUSEENTER,TMessage,CMMouseEnter)
   MESSAGE_HANDLER(CM_MOUSELEAVE,TMessage,CMMouseLeave)
   MESSAGE_HANDLER(CM_ENABLEDCHANGED,TMessage,CMEnabledChanged)
END_MESSAGE_MAP(TGraphicControl)
};
//---------------------------------------------------------------
#endif
```

Листинг 9.47. Файл `ExampleButton.cpp` с исходным кодом кнопки `TExampleButton`

```
//---------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "ExampleButton.h"
#pragma package(smart_init)
//---------------------------------------------------------------
// ValidCtrCheck используется для того, чтобы гарантировать
// отсутствие чисто виртуальных функций в компоненте.
//

static inline void ValidCtrCheck(TExampleButton *)
{
   new TExampleButton(NULL);
}
//---------------------------------------------------------------
__fastcall TExampleButton::TExampleButton(TComponent*Owner)
             : TGraphicControl(Owner)
{
   SetBounds(0, 0, 50, 50);
   ControlStyle = ControlStyle << csReplicatable;
   FState = esFlat;
}
//---------------------------------------------------------------

namespace Examplebutton
{
   void __fastcall PACKAGE Register()
   {
      TComponentClass classes[1] == {__classid(TExampleButton)};
      RegisterComponents("Samples", classes, 0);
   }
}
```

```
//---------------------------------------------------------------
void __fastcall TExampleButton::CMMouseEnter(TMessage &Msg)
{
   if(Enabled)
   {
      FState = esUp;
      FMouseInControl = true;
      Invalidate();
   }
}
//---------------------------------------------------------------
void __fastcall TExampleButton::CMMouseLeave(TMessage &Msg)
{
   if(Enabled)
   {
      FState = esFlat;
      FMouseInControl = false;
      Invalidate();
   }
}
//---------------------------------------------------------------
void __fastcall TExampleButton::CMEnabledChanged(TMessage &Msg)
{
   FState = (Enabled) ? esFlat : esDisabled;
   Invalidate();
}
//---------------------------------------------------------------
void __fastcall TExampleButton::MouseDown(TMouseButton Button,
                                          TShiftState Shift,
                                          int X, int Y)
{
   if(Button == mbLeft)
   {
      if(Enabled && FMouseInControl)
      {
         FState = esDown;
         Invalidate();
      }
   }
}
//---------------------------------------------------------------
void __fastcall TExampleButton::MouseUp(TMouseButton Button,
                                        TShiftState Shift,
                                        int X, int Y)
{
   if(Button == mbLeft)
   {
      if(Enabled && FMouseInControl)
      {
         FState = esUp;
         Invalidate();
```

```
        if(FOnClick)
           FOnClick(this);
      }
   }
}
//---------------------------------------------------------------
void __fastcall TExampleButton::SetGlyph(
                                  Graphics::TBitmap *Value)
{
   if(Value == NULL)
      return;

   if(!FGlyph)
      FGlyph = new Graphics::TBitmap;

   FGlyph->Assign(Value);
   Invalidate();
}
//---------------------------------------------------------------
void __fastcall TExampleButton::SetCaption(AnsiString Value)
{
   FCaption = Value;
   Invalidate();
}
//---------------------------------------------------------------
void __fastcall TExampleButton::SwapColors(TColor &Top,
                                           TColor &Bottom)
{
   if(ComponentState.Contains(csDesigning))
   {
      FState = esUp;
   }

   Top = (FState == esUp) ? clBtnHighlight : clBtnShadow;
   Bottom = (FState == esDown) ? clBtnHighlight : clBtnShadow;
}
//---------------------------------------------------------------
void __fastcall TExampleButton::BeforeDestruction(void)
{
   if(FImage)
      delete FImage;

   if(FGlyph)
      delete FGlyph;
}
//---------------------------------------------------------------
void __fastcall TExampleButton::Paint(void)
{
   TRect cRect, tRect, gRect;
   TColor TopColor, BottomColor;
```

```
   Canvas->Brush->Color = clBtnFace;
   Canvas->FillRect(ClientRect);
   cRect = ClientRect;
   Graphics::TBitmap *bit = new Graphics::TBitmap;
   bit->Width = ClientWidth;
   bit->Height = ClientHeight;
   bit->Canvas->Brush->Color = clBtnFace;
   bit->Canvas->FillRect(cRect);

   if(FGlyph)
      if(!FGlyph->Empty)
      {
         CalcGlyphLayout(gRect);
         bit->Canvas->BrushCopy(gRect, FGlyph,
            Rect(0, 0, FGlyph->Width, FGlyph->Height),
            FGlyph->TransparentColor);
      }

      if(!FCaption.IsEmpty())
      {
         CalcTextLayout(tRect);
         bit->Canvas->TextRect(tRect, tRect.Left, tRect.Top,
                               FCaption);
      }

      if(FState == esUp || FState == esDown)
      {
         SwapColors(TopColor, BottomColor);
         Frame3D(bit->Canvas, cRect, TopColor, BottomColor, 1);
      }

      BitBlt(Canvas->Handle, 0, 0, ClientWidth, ClientHeight,
             bit->Canvas->Handle, 0, 0, SRCCOPY);
   delete bit;
}
//---------------------------------------------------------------
void __fastcall TExampleButton::CalcGlyphLayout(TRect &r)
{
   int TotalHeight=0;
   int TextHeight=0;

   if(!FCaption.IsEmpty())
      TextHeight = Canvas->TextHeight(FCaption);

   // Слагаемое 5 выполняет роль свойства 'Spacing',
   // но для простоты используется вместо него.
   TotalHeight = FGlyph->Height + TextHeight + 5;

   r = Rect((ClientWidth/2)-(FGlyph->Width/2),
           ((ClientHeight/2)-(TotalHeight/2)), FGlyph->Width +
            (ClientWidth/2)-(FGlyph->Width/2), FGlyph->Height +
```

```
          ((ClientHeight/2)-(TotalHeight/2)));
}
//---------------------------------------------------------------
void __fastcall TExampleButton::CalcTextLayout(TRect &r)
{
   int TotalHeight=0;
   int TextHeight=0;
   int TextWidth=0;
   TRect temp;

   if(FGlyph)
      TotalHeight = FGlyph->Height;

   TextHeight = Canvas->TextHeight(FCaption);
   TextWidth = Canvas->TextWidth(FCaption);

   TotalHeight += TextHeight + 5;

   temp.Left = 0;
   temp.Top = (ClientHeight/2)-(TotalHeight/2);
   temp.Bottom = temp.Top + TotalHeight;
   temp.Right = ClientWidth;

   r = Rect(((ClientWidth/2) - (TextWidth/2)),
             temp.Bottom-TextHeight,
            ((ClientWidth/2)-(TextWidth/2)) + TextWidth,
            temp.Bottom);
}
```

Здесь опубликовано только событие `OnClick`. В реальном рабочем компоненте, вероятно, потребуется опубликовать также события `OnMouseUp`, `OnMouseDown` и `OnMouseMove`. Кроме того, опубликованы свойства `Caption` и `Glyph`, но чтобы пользователи могли изменять шрифт текста, следует опубликовать свойство `Font`.

Рекомендуется также организовать перехват сообщения `CM_FONTCHANGED` для соответствующей перерисовки расположения надписи и рельефного изображения кнопки. При вычислении расположения изображения и текста между ними следует задать промежуток величиной в 5 пикселей. Кроме того, следует создать свойство, с помощью которого пользователь мог бы указать величину этого промежутка.

В листинге 9.47 обратите внимание на метод `SetGlyph()`, предназначенный для записи значения свойства `Glyph`. Если свойству `Glyph` присваивается указатель `NULL`, этот метод просто возвращается без выполнения каких-либо действий. Не первый взгляд может показаться, что это типичное поведение подобного свойства, но после указания изображения не существует способа избавиться от него. Другими словами, нельзя отобразить на экране одну только надпись, не удалив компонент целиком и не создав новый.

Напоследок рассмотрим логическую переменную `FMouseInControl`. Так как данный элемент управления реагирует на сообщения мыши, то на основе этой переменной отслеживается событие нахождения указателя мыши на элементе управления. Без этой переменной некоторые члены-функции были бы вызваны не там, где нужно, поскольку компонент будет получать сообщения мыши, даже если действия с мышью начались не над данным элементом управления. Предположим, например, что пользователь щелкает кнопкой мыши и, удерживая

ее, перемещает указатель мыши над элементом управления, а затем отпускает кнопку. В таком случае будет вызван метод `CMMouseUp()`, причем элементу управления не будет известно, что указатель мыши находится над ним. В результате будет перерисован весь компонент в состоянии `Up`, но в данном примере этого не произойдет до тех пор, пока указатель мыши не будет сдвинут и возвращен обратно или не будет произведен щелчок мышью на этой кнопке. Как раз для этого и нужна переменная `FMouseInControl`.

В листинге 9.47 форма кнопки создается с помощью метода `Frame3D()`. Включая заголовочный файл `Buttons.hpp` в исходный код, можно получить доступ к другому методу рисования геометрической формы в виде кнопки `DrawButtonFace()`, который показан в листинге 9.48.

Листинг 9.48. Метод `DrawButtonFace()`

```
TRect DrawButtonFace(TCanvas *Canvas, const TRect Client,
   int BevelWidth, TButtonStyle Style, bool IsRounded,
   bool IsDown, bool IsFocused);
```

Метод `DrawButtonFace()` перерисует кнопку с указанным размером прямоугольника `Client` и канвой `Canvas`. Некоторые параметры задаются в соответствии со значениями других параметров. Например, параметры `BevelWidth` и `IsRounded` имеют смысл только в том случае, если для свойства `Style` задано значение `bsWin31`.

Метод `DrawButtonFace()` использует API-функцию `DrawEdge()` (более подробные сведения о ней вы найдете в справке о Win32 в C++Builder) и может применяться в ваших подпрограммах.

Изменение оконных компонентов

Как уже говорилось выше, оконные компоненты являются оболочкой для стандартных элементов управления Windows. Эти компоненты могут перерисовываться автоматически, поэтому они не нуждаются в явном указании способа перерисовки. При изменении оконных элементов управления, вам, вероятно, потребуется изменить его способ функционирования, а не внешний вид. К счастью, в библиотеке VCL предусмотрено несколько защищенных методов для этих компонентов, которые могут быть переопределены как раз для выполнения этой задачи.

В последнем примере мы используем описанные в этой главе способы для более знакомой вам и надежной замены стандартного компонента `TFileListBox` в C++Builder. Прежде чем приступать к созданию кода, полезно ознакомимся с действиями, которые будут выполняться. Напомним, что нам хотелось бы создать максимально простой компонент и избавить пользователя от необходимости создания стандартного кода, который используется при работе с компонентом, содержащим список файлов. Ниже кратко перечислены эти действия.

- Отображение правильной пиктограммы для каждого файла.
- Создание для пользователя возможности запускать приложение или открывать документ после двойного щелчка на нем.
- Предоставление пользователю возможности добавить элемент в этот список.
- Организация возможности выбора элемента списка после щелчка на нем правой кнопкой мыши.
- Отображение горизонтальной полосы прокрутки, если длина элемента списка превышает длину окна списка.
- Организация совместимости с элементом управления `TDirectoryListBox`.

Теперь, определив назначение элемента управления, необходимо выбрать базовый класс для него. Как уже говорилось выше, в C++Builder предусмотрено несколько пользовательских классов, на основе которых можно создавать производные компоненты. Но этот способ не годится в нашем случае. Классы `TDirectoryListBox` и `TFileListBox` связаны посредством свойства `FileList` класса `TDirectoryListBox`. Это свойство объявляется как указатель на объект `TFileListBox`, поэтому компонент, производный от класса `TCustomListBox` или `TListBox`, не будет видим для этого свойства. Для поддержания совместимости с классом `TDirectoryListBox` необходимо сделать наш компонент производным от класса `TFileListBox`. К счастью, его методы чтения имен файлов являются защищенными, поэтому все, что нам потребуется сделать, — переопределить их в новом компоненте.

Рассмотрим изменения, которые необходимо внести в компонент, а также объявления новых свойств, методов и событий. Во-первых, пользователь должен иметь возможность запускать приложение или открывать документ из списка при двойном щелчке на нем мышью. Для включения и отключения этой возможности используется следующее логическое свойство `CanLaunch`.

```
__property bool CanLaunch = {read=FCanLaunch,
                             write=FCanLaunch,
                             default=true};
```

После двойного щелчка мышью в окне списка будет послано сообщение `WM_LBUTTONDBLCLK`. В классе `TCustomListBox` предусмотрен защищенный метод, который вызывается в ответ на это сообщение. В листинге 9.49 показан способ переопределения метода `DblClick()` для запуска приложения или открытия документа из этого списка.

Листинг 9.49. Метод `DblClick()`

```
void __fastcall TSHFileListBox::DblClick(void)
{
   if(FCanLaunch)
   {
      int ii=0;
      // Просмотр списка и поиска выбранного элемента
      for(ii=0; ii < Items->Count; ii++)
      {
         if(Selected[ii])
         {
            AnsiString str = Items->Strings[ii];
            ShellExecute(Handle, "open", str.c_str(), 0, 0,
                         SW_SHOWDEFAULT);
         }
      }
   }
   // Активизация события OnDblClick
   if(FOnDblClick)
      FOnDblClick(this);
}
```

Если переменная `FCanLaunch` имеет значение `true`, нужно сначала найти выбранный элемент, а потом использовать API-функцию `ShellExecute()` для запуска приложения. Этот метод также активизирует событие `OnDblClick`, которое имеет следующее объявление.

```
private:
   TNotifyEvent FOnDblClick;

__published:
   __property TNotifyEvent OnDblClick = {read=FOnDblClick,
                                         write=FOnDblClick};
```

Событие `OnDblClick` может не возвращать никакой информации, поэтому его можно объявить как переменную типа `TNotifyEvent`. Это поведение может быть изменено в случае необходимости, но в данном примере это нас устраивает. Теперь рассмотрим способ выбора элемента списка после щелчка правой кнопкой мыши. Во-первых, нужно объявить новое свойство так, как показано ниже.

```
__property bool RightBtnClick = {read=FRightBtnSelect,
                                 write=FRightBtnSelect,
                                 default=true};
```

Обратите внимание, что свойство ссылается на член-данные без какого-либо метода чтения и записи. Дело в том, что выбор элемента списка определяется с помощью события `MouseUp()`, код которого показан в листинге 9.50.

Листинг 9.50. Метод `MouseUp()`

```
//---------------------------------------------------------
void __fastcall TSHFileListBox::MouseUp(TMouseButton Button,
                                        TShiftState Shift,
                                        int X, int Y)
{
   if(!FRightBtnSel)
      return;

   TPoint ItemPos = Point(X, Y);
   // Расположен ли указатель мыши над элементом списка ?
   int Index = ItemAtPos(ItemPos, true);
   // Если нет - возврат
   if(Index == -1)
      return;
   // В противном случае - выбор этого элемента списка
   Perform(LB_SETCURSEL, (WPARAM)Index, 0);
}
```

Код в листинге 9.50 организован очень просто. Сначала проверяется переменная `FRightBtnSel`, чтобы убедиться в том, что выбран элемента списка. Затем текущие координаты указателя мыши преобразуются в структуру точки `TPoint`. Для поиска элемента списка, который находится под указателем мыши, используется метод `ItemAtPos()` класса `TCustomListBox`, принимающий значение созданной структуры `TPoint` и логическое значение, которое приводит к возврату значения `-1`, если координаты структуры `TPoint` выходят за пределы последнего элемента списка. Здесь логический параметр имеет значение `true`, и если возвращаемое значение равно `-1`, то метод просто возвращается. Заменив значение этого логического параметра на `false`, можно удалить конструкцию `if()`, которая проверяет возвращаемое значение. Наконец, метод `Perform()` используется для принудительной организации

такого поведения компонента, как если бы он получил оконное сообщение. Первый параметр метода Perform() содержит именно то сообщение, которое необходимо имитировать в данном случае. Сообщение LB_SETCURSEL указывает на то, что в списке с помощью указателя мыши изменен выбор элемента. Второй параметр содержит индекс выбранного элемента, а третий в данном случае не используется и равен нулю.

Далее попробуем организовать для пользователя возможность добавления элемента списка. Класс TFileListBox имеет свойство Mask, которое позволяет указать расширения для тех типов файлов, которые могут быть добавлены в список. Свойство Mask может быть переопределено с предоставлением методов чтения и записи, которые будут фильтровать имена файлов в соответствии со значением свойства Mask. В данном примере выбран простейший способ на основе организации события, которое позволяет пользователю применить собственный алгоритм фильтрации имен файлов. Это событие можно использовать наряду со свойством Mask для повышения функциональности.

Сначала объявим новое событие так, как показано ниже.

```
typedef void __fastcall (__closure *TAddItemEvent)(
                                    TObject *Sender,
                                    AnsiString Item,
                                    bool &CanAdd);
```

Как видите, это событие имеет три параметра. Параметр Sender — это сам список, Item —строковая переменная типа AnsiString, которая содержит имя файла, а CanAdd — логическая переменная, которая определяет возможность добавления элемента. Обратите внимание, что параметр CanAdd передается по ссылке, так что пользователь сможет изменить это значение на значение false в обработчике события, предотвратив добавление элемента Item в список.

Прежде чем приступить к рассмотрению способов получения имен файлов и вставки их в список, познакомимся со способом использования пиктограмм, которые применяются в окне программы Windows Explorer (листинг 9.51).

Листинг 9.51. Получение списка системных рисунков

```
SHFILEINFO shfi;
DWORD iHnd;
TImageList *Images;
Images = new TImageList(this);
Images->ShareImages = true;
iHnd = SHGetFileInfo("", 0, &shfi, sizeof(shfi),
                     SHGFI_SYSICONINDEX |
                     SHGFI_SHELLICONSIZE |
                     SHGFI_SMALLICON);
if(iHnd != 0)
   Images->Handle = iHnd;
```

Обратите внимание, что в листинге 9.51 для свойства ShareImages объекта Images задано значение true. Это очень важный момент. Таким образом список рисунков уведомляется о том, что его дескриптор не удаляется при удалении компонента. Дело в том, что дескриптор списка системных рисунков принадлежит системе, а если компонент удалит его, Windows не сможет отображать свои пиктограммы в других командах меню и ярлыках файлов. Эти нежелательные последствия можно устранить с помощью перезагрузки, позволив операционной системе Windows вновь получить дескриптор системных изображений.

На этом этапе мы переопределим метод `ReadFileNames()` класса `TFileListBox` для извлечения имен файлов несколько по-иному. В нашей версии для получения имен файлов будет использоваться оболочка с COM-интерфейсом. Поскольку просмотр всех идентификаторов элементов списка (itemid) может оказаться очень запутанным делом и его изложение выходит за рамки этой главы, мы не будем подробно рассматривать их. Создадим новый метод `AddItem()`, код которого показан в листинге 9.52. Он извлекает отображаемое имя файла и индекс его пиктограммы в списке системных изображений и приводит к возникновению созданного ранее события `OnAddItem`.

На заметку

itemid — альтернативное английское название термина "идентификатор элемента" (item identifier) или "идентификатор списка" (identifier list). Более подробные сведения о них можно найти в разделе "Item Identifiers and Identifier Lists" в справке о Win32, которая поставляется вместе с C++Builder.

Листинг 9.52. Метод `AddItem()`

```
int __fastcall TSHFileListBox::AddItem(LPITEMIDLIST pidl)
{
   SHFILEINFO shfi;
   int Index;

   SHGetFileInfo((char*)pidl, 0, &shfi, sizeof(shfi),
                 SHGFI_PIDL | SHGFI_SYSICONINDEX |
                 SHGFI_SMALLICON | SHGFI_DISPLAYNAME |
                 SHGFI_USEFILEATTRIBUTES);

   // Приводит к возникновению события OnAddItem, чтобы
   // предоставить или не предоставить пользователю возможность
   // добавления имени файла
   bool FCanAdd = true;

   if(FOnAddItem)
      FOnAddItem(this, AnsiString(shfi.szDisplayName), FCanAdd);

   if(FCanAdd)
   {
      TShellFileListItem *ShellInfo =
         new TShellFileListItem(pidl, shfi.iIcon);
      Index = Items->AddObject(AnsiString(shfi.szDisplayName),
                               (TObject*)ShellInfo);
      // Возвращение длин имени файла
      return Canvas->TextWidth(Items->Strings[Index]);
   }

   // Возвращает 0 для длины имени файла, если оно не вставлено
   return 0;
}
```

Метод `AddItem()` в качестве единственного параметра принимает идентификатор элемента списка (itemid) и возвращает целое значение. В листинге 9.52 для извлечения отображае-

мого имени файла и индекса пиктограммы используется API-функция SHGetFileInfo(). После получения отображаемого имени файла создается логическая переменная CanAdd для определения необходимости вставки этого имени в список, а потом инициируется событие OnAddItem. После этого можно проверить значение переменной CanAdd и если оно равно true, можно добавить новый элемент в окно списка. После вставки элемента для получения его ширины в пикселях применяется метод TextWidth() класса TCanvas. Метод возвращает значение ширины, если элемент вставлен, или значение 0 — в противном случае. Причины такого поведения будут рассмотрены ниже.

Рассмотрим теперь класс TShellFileListItem. Так как на самом деле в списке для каждого элемента следует нарисовать пиктограмму вместе с текстом, то потребуется отслеживать индекс пиктограммы для каждого элемента. Для каждого вставляемого в список элемента мы создадим экземпляр класса TShellFileListItem и присвоим его свойству Object свойства Items списка. Благодаря этому мы сможем впоследствии извлечь его при рисовании пиктограммы элемента списка. Класс TShellFileListItem также содержит копию идентификатора элемента списка (itemid) на случай возможных усовершенствований в будущем (например, для создания наследника TSHFileListBox и переопределения метода MouseUp() для отображения контекстного меню для данного имени файла).

При использовании свойства Object следует иметь в виду, что память, используемая для хранения экземпляра TShellFileListItem, должна освобождаться при удалении элемента из списка. Это можно сделать, переопределив метод DeleteString() так, как показано в листинге 9.55.

Как говорилось выше, возвращаемое методом AddItem() значение содержит длину в пикселях вставляемого элемента списка. Это значение используется для определения самого длинного элемента списка и отображения горизонтальной полосы прокрутки, если его длина превышает ширину окна списка. Рассмотрим следующий фрагмент кода из метода ReadFileNames().

```
while(Fetched > 0)
{
   // Добавление элемента в список
   int l = AddItem(rgelt);
   if(l > hExtent)
      hExtent = l;
   ppenumIDList->Next(celt, &rgelt, &Fetched);
}
```

В нем просматривается список идентификаторов для элементов-файлов, находящихся в каталоге, и извлекается идентификатор каждого файла. Метод AddItem() возвращает длину элемента и сравнивает ее с длиной предыдущего элемента. Если он оказывается длиннее, то переменной l присваивается значение новой длины, а процесс повторяется до тех пор, пока не будут просмотрены все файлы списка. В результате в конце цикла переменная l будет содержать значение длины самого длинного элемента. Затем для определения необходимости создания горизонтальной полосы прокрутки вызывается метод DoHorizontalScrollBar().

Метод DoHorizontalScrollBars(), код которого представлен в листинге 9.53, принимает в качестве параметра целочисленное значение, которое равно длине элемента (в пикселях), добавленного в список. Это значение увеличивается на 2 для создания левого края, если свойство ShowGlyphs имеет значение true, затем — на 18 для учета ширины пиктограммы и промежутка между ней и текстом. Наконец, метод Perform() вызывается для указания горизонтального размера элементов списка. Если значение WPARAM окажется больше, чем ширина этого элемента управления, появится горизонтальная полоса прокрутки.

Листинг 9.53. Создание горизонтальной полосы прокрутки

```
void __fastcall TSHFileListBox::DoHorizontalScrollBar(int he)
{
   he += 2;
   if(ShowGlyphs)
      he += 18;

   Perform(LB_SETHORIZONTALEXTENT, he, 0);
}
```

В листингах 9.54 и 9.55 показан весь код компонента TSHFileListBox, который в электронном виде находится в каталоге SHFileListBox на прилагаемом к книге компакт-диске.

Листинг 9.54. Заголовочный файл SHFileListBox.h компонента TSHFileListBox

```
//---------------------------------------------------------------
#ifndef SHFileListBoxH
#define SHFileListBoxH
//---------------------------------------------------------------
#include <SysUtils.hpp>
#include <Controls.hpp>
#include <Classes.hpp>
#include <Forms.hpp>
#include <FileCtrl.hpp>
#include <StdCtrls.hpp>
#include "ShlObj.h"

//---------------------------------------------------------------

class TShellFileListItem : public TObject
{
private:
   LPITEMIDLIST Fpidl;
   int FImageIndex;
public:
   __fastcall TShellFileListItem(LPITEMIDLIST lpidl, int Index);
   __fastcall ~TShellFileListItem(void);
   __property LPITEMIDLIST pidl = {read=Fpidl};
   __property int ImageIndex = {read=FImageIndex};
};

typedef void __fastcall
   (__closure *TAddItemEvent)(TObject *Sender,
                              AnsiString Item, bool &CanAdd);

class PACKAGE TSHFileListBox :public TFileListBox
{
private:
   TImageList *FImages;
   TNotifyEvent FOnDblClick;
```

```
   bool FCanLaunch;
   bool FRightBtnSel;
   TAddItemEvent FOnAddItem;
   int __fastcall AddItem(LPITEMIDLIST pidl);
   void __fastcall GetSysImages(void);

protected:
   void __fastcall DblClick(void);
   void __fastcall ReadFileNames(void);
   void __fastcall MouseUp(TMouseButton Button,
                           TShiftState Shift,
                           int X, int Y);
   void __fastcall DrawItem(int Index, const TRect &Rect,
                            TOwnerDrawState State);
   void __fastcall DoHorizontalScrollBar(int he);
   void __fastcall DeleteString(int Index);

public:
   __fastcall TSHFileListBox(TComponent*Owner);
   __fastcall ~TSHFileListBox(void);

__published:
   __property bool CanLaunch = {read=FCanLaunch,
                                write=FCanLaunch,
                                default=true};
   __property bool RightBtnSel = {read=FRightBtnSel,
                                  write=FRightBtnSel,
                                  default=true};
   __property TNotifyEvent OnDblClick = {read=FOnDblClick,
                                         write=FOnDblClick};
   __property TAddItemEvent OnAddItem = {read=FOnAddItem,
                                         write=FOnAddItem};
};
#endif
```

Листинг 9.55. Исходный код компонента `TSHFileListBox` из файла `SHFileListBox.cpp`

```
//---------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop
#include "SHFileListBox.h"
#pragma package(smart_init)
//---------------------------------------------------------------
__fastcall TShellFileListItem::TShellFileListItem(
             LPITEMIDLIST lpidl, int Index) : TObject()
{
   // Сохранение копии идентификатора (pidl) элемента списка
   Fpidl = CopyPIDL(lpidl);
   // и индекса его пиктограммы.
   FImageIndex = Index;
```

```
}
//---------------------------------------------------------------
   __fastcall TShellFileListItem::~TShellFileListItem(void)
{
   LPMALLOC lpMalloc = NULL;
   if(SUCCEEDED(SHGetMalloc(&lpMalloc)))
   {
      // Осовобождение памяти, связанной с
      // идентификатором (pidl) элемента списка
      lpMalloc->Free(Fpidl);
      lpMalloc->Release();
   }
}

//---------------------------------------------------------------
__fastcall TSHFileListBox::TSHFileListBox(TComponent*Owner) :
   TFileListBox(Owner)
{
   ItemHeight = 18;
   ShowGlyphs = true;
   FCanLaunch = true;
   FRightBtnSel = true;
}

//---------------------------------------------------------------
__fastcall TSHFileListBox::~TSHFileListBox(void)
{
   // Освобождение памяти рисунков
   if(FImages)
   delete FImages;
   FImages = NULL;
}

//---------------------------------------------------------------
void __fastcall TSHFileListBox::DeleteString(int Index)
{
// Этот метод вызывается в ответ на сообщение LB_DELETESTRING.
// Сначала удаляется TShellFileListItem по указателю строки
// в свойстве Object.
   TShellFileListItem *ShellItem =
      reinterpret_cast<TShellFileListItem*>(Items->Objects[Index]);
   delete ShellItem;
   ShellItem = NULL;

   // Теперь удаляется сам элемент списка.
   Items->Delete(Index);
}

//---------------------------------------------------------------
```

```
namespace Shfilelistbox
{
   void __fastcall PACKAGE Register()
   {
      TComponentClass classes[1] = {__classid(TSHFileListBox)};
      RegisterComponents("Samples", classes, 0);
   }
}

//---------------------------------------------------------------
void __fastcall TSHFileListBox::ReadFileNames(void)
{
   LPMALLOC g_pMalloc;
   LPSHELLFOLDER pisf;
   LPSHELLFOLDER sfChild;
   LPITEMIDLIST pidlDirectory;
   LPITEMIDLIST rgelt;
   LPENUMIDLIST ppenumIDList;
   int hExtent;

   try
   {
      try
      {
         if(HandleAllocated())
         {
            GetSysImages();
            // Запрет на обновления экрана.
            Items->BeginUpdate();
            // Удаление элементов, уже имеющихся в списке.
            Items->Clear();
            // Получение глобального указателя для
            // области памяти оболочки
            if(SHGetMalloc(&g_pMalloc) != NOERROR)
            {
               return;
            }
            // Получение интерфейса IShellFolder
            // рабочего стола
            if(SHGetDesktopFolder(&pisf) != NOERROR)
            {
               return;
            }
            // Преобразование строки с именем каталога
            // к формату WideChar
            WideChar oleStr[MAX_PATH];
            FDirectory.WideChar(oleStr, MAX_PATH);
            unsigned long pchEaten;
            unsigned long pdwAttributes;
```

```
            // Получение идентификатора pidl текущего каталога.
            pisf->ParseDisplayName(Handle, 0, oleStr, &pchEaten,
                                   &pidlDirectory,
                                   &pdwAttributes);
            // Получение интерфейса IShellFolder
            // текущего каталога.
            if(pisf->BindToObject(pidlDirectory, NULL,
                                  IID_IShellFolder,
                                  (void**)&sfChild) != NOERROR)
            {
               return;
            }
            // Перечисление объектов внутри каталога.
            sfChild->EnumObjects(Handle,
                                 SHCONTF_NONFOLDERS |
                                 SHCONTF_INCLUDEHIDDEN,
                                 &ppenumIDList);
            // Обход элементов списка.
            ULONG celt = 1;
            ULONG Fetched = 0;
            ppenumIDList->Next(celt, &rgelt, &Fetched);
            hExtent = 0;
            while(Fetched >0)
            {
               // Добавление элемента в список.
               int l = AddItem(rgelt);
               if(l >hExtent)
               hExtent = l;
               ppenumIDList->Next(celt, &rgelt, &Fetched);
            }
         }
      }
      catch(Exception &E)
      {
         throw(E);
         // Повторный перехват исключительных ситуаций.
      }
   }
   __finally
   {
   // Эти действия будут выполнены несмотря ни на что.
      g_pMalloc->Free(rgelt);
      g_pMalloc->Free(ppenumIDList);
      g_pMalloc->Free(pidlDirectory);
      pisf->Release();
      sfChild->Release();
      g_pMalloc->Release();
      Items->EndUpdate();
   }
```

```
   // В случае необходимости создать горизонтальную полосу прокрутки.
   DoHorizontalScrollBar(hExtent);
}

//---------------------------------------------------------------
void __fastcall TSHFileListBox::DoHorizontalScrollBar(int he)
{
   // Добавить пространство для границ.
   he += 2;

   // При использовании пиктограмм нужно зарезервировать место
   // для изображения и промежутка между ним и текстом.
   if(ShowGlyphs)
      he += 18;

   Perform(LB_SETHORIZONTALEXTENT, he, 0);
}

//---------------------------------------------------------------
void __fastcall TSHFileListBox::GetSysImages(void)
{
   SHFILEINFO shfi;
   DWORD iHnd;
   if(!FImages)
   {
      FImages = new TImageList(this);
      FImages->ShareImages = true;
      FImages->Height = 16;
      FImages->Width = 16;
      iHnd = SHGetFileInfo("", 0, &shfi, sizeof(shfi),
                           SHGFI_SYSICONINDEX |
                           SHGFI_SHELLICONSIZE |
                           SHGFI_SMALLICON);
      if(iHnd != 0)
         FImages->Handle = iHnd;
   }
}

//---------------------------------------------------------------
int __fastcall TSHFileListBox::AddItem(LPITEMIDLIST pidl)
{
   SHFILEINFO shfi;
   int Index;

   SHGetFileInfo((char*)pidl, 0, &shfi, sizeof(shfi),
                 SHGFI_PIDL | SHGFI_SYSICONINDEX |
                 SHGFI_SMALLICON | SHGFI_DISPLAYNAME |
                 SHGFI_USEFILEATTRIBUTES);
```

```
// Возникновение события OnAddItem для разрешения или
// запрещения пользователю вставить файл
   bool FCanAdd = true;
   if(FOnAddItem)
      FOnAddItem(this, AnsiString(shfi.szDisplayName), FCanAdd);

   if(FCanAdd)
   {
      TShellFileListItem *ShellInfo =
         new TShellFileListItem(pidl, shfi.iIcon);
      Index = Items->AddObject(AnsiString(shfi.szDisplayName),
                               (TObject*)ShellInfo);
      // Возвращение длины имени файла
      return Canvas->TextWidth(Items->Strings[Index]);
   }

   // Возврат 0 для длины имени файла, если он не был вставлен.
   return 0;
}

//---------------------------------------------------------------
void __fastcall TSHFileListBox::DrawItem(int Index,
                                         const TRect &Rect,
                                         TOwnerDrawState State)
{
   int Offset;

   Canvas->FillRect(Rect);
   Offset = 2;

   if(ShowGlyphs)
   {
      TShellFileListItem *ShellItem =
         reinterpret_cast<TShellFileListItem*>
↳(Items->Objects[Index]);
      // Рисование пиктограммы файла в списке.
      FImages->Draw(Canvas, Rect.Left+2, Rect.Top+2,
                    ShellItem->ImageIndex, true);
      Offset += 18;
   }

   int Texty = Canvas->TextHeight(Items->Strings[Index]);
   Texty = ((ItemHeight - Texty) / 2) + 1;
   // Теперь рисуется текст.
   Canvas->TextOut(Rect.Left +Offset, Rect.Top +Texty,
                   Items->Strings[Index]);
}

//---------------------------------------------------------------
```

```
void __fastcall TSHFileListBox::DblClick(void)
{
   if(FCanLaunch)
   {
      int ii = 0;
      // Обход списка и поиск выбранного элемента.
      for(ii = 0; ii < Items->Count; ii++)
      {
         if(Selected[ii])
         {
            AnsiString str = Items->Strings[ii];
            ShellExecute(Handle, "open", str.c_str(), 0, 0,
                         SW_SHOWDEFAULT);
         }
      }
   }

   // Возникновение события OnDblClick.
   if(FOnDblClick)
      FOnDblClick(this);
}

//---------------------------------------------------------------
void __fastcall TSHFileListBox::MouseUp(TMouseButton Button,
                                        TShiftState Shift,
                                        int X, int Y)
{
   if(!FRightBtnSel)
      return;

   TPoint ItemPos = Point(X, Y);
   // Есть ли элемент под указателеи мыши?
   int Index = ItemAtPos(ItemPos, true);
   // Если нет - возврат...
   if(Index = = -1)
      return;
   // ... в противном случае - выбор этого элемента.
   Perform(LB_SETCURSEL, (WPARAM)Index, 0);
}

//---------------------------------------------------------------
// Метод ValidCtrCheck используется для проверки того, не имеют ли
// созданные компоненты чисто виртуальных функций.
//
static inline void ValidCtrCheck(TSHFileListBox *)
{
   new TSHFileListBox (NULL);
}
```

Создание пользовательских компонентов, связанных с данными

При создании пользовательских компонентов, связанных с данными, очень важно правильно выбрать класс-предок. В этом разделе рассматривается расширение компонента редактирования `TMaskEdit` для чтения данных из источника данных и отображения их в соответствии с заданным форматом маски. Такой элемент управления называется *элементом просмотра данных* (*data-browsing control*). Мы расширим этот элемент управления до пользовательского компонента, связанного с данными. То есть изменения данных в поле и базе данных будут отражены в обоих направлениях.

Создание элемента только для чтения данных

Создаваемый элемент управления уже имеет свойство `ReadOnly`, регламентирующее только режим чтения, поэтому нам не придется создавать его вручную. В противном случае его следует создать так же, как и любое другое свойство, как показано в листинге 9.56 (учтите, что этот код не является обязательным для данного компонента).

Листинг 9.56. Создание свойства, регламентирующего только режим чтения

```
class PACKAGE TDBMaskEdit : public TMaskEdit
{
private:
   bool FReadOnly;
protected:
public:
   __fastcall TDBMaskEdit(TComponent*Owner);
__published:
   __property ReadOnly = {read = FReadOnly, write = FReadOnly,
                          default = true};
};
```

В конструкторе следует указать используемое по умолчанию значение этого свойства.

```
__fastcall TDBMaskEdit::TDBMaskEdit(TComponent*Owner)
   : TMaskEdit(Owner)
{
   FReadOnly = true;
}
```

Наконец, нужно организовать функционирование компонента только в режиме для чтения. Для этого нужно переопределить метод, который обычно связан с пользователем, осуществляющим доступ к этому элементу управления. При создании элемента управления в виде таблицы данных это — метод `SelectCell()`, в котором нужно организовать проверку значения свойства `ReadOnly` с соответствующей обработкой ситуации. Если свойство `ReadOnly` имеет значение `false`, следует вызвать унаследованный метод, в противном случае — метод просто возвращается.

Если элемент управления `TMaskEdit` имеет метод `SelectEdit()`, код будет выглядеть, как показано ниже.

```
bool __fastcall TDBMaskEdit::SelectEdit(void)
{
   if(FReadOnly)
      return(false);
   else
      return(TMaskEdit::SelectEdit());
}
```

В этом примере нам не придется создавать свойство `ReadOnly`, так как элемент управления `TMaskEdit` уже имеет его.

Установление связи с источником данных

Для связи элемента управления с данными необходимо установить связь с членом-элементом базы данных. Для установления связи с источником данных предусмотрен класс `TFieldDataLink`.

Связанный с данными элемент управления имеет собственный класс для установления связи с источником данных. Именно элемент управления отвечает за создание, инициализацию и удаление связи с источником данных.

Для установления этой связи требуется выполнить следующие действия.

1. Объявить класс установления связи с источником данных как член данного элемента управления.
2. Объявить соответствующим образом свойства чтения и записи.
3. Инициализировать связь с данными.

Объявление связи с источником данных

Связь с источником данных организуется на основе класса `TFieldDataLink` и файла `DBCTRLS.HPP`, который должен быть указан в заголовочном файле.

```
#include <DBCtrls.hpp>
class PACKAGE TDBMaskEdit : public TMaskEdit
{
private:
   TFieldDataLink *FDataLink;
...
};
```

Теперь в связанном с данными компоненте (как и в любых других связанных с данными компонентах) нужно создать свойства `DataSource` и `DataField`. Для доступа к свойствам класса связи с источником данных эти свойства используют методы "ретранслирования" ("pass-through"). Это позволяет использовать элемент управления и его соединение с данными для доступа к источнику данных наряду с другими элементами управления и пользователями.

Объявление доступа для чтения и записи

Способ доступа к данным в элементе управления определяется способом объявления его свойств. Попробуем предоставить компоненту полный доступ. Он имеет свойство `ReadOnly`, которое автоматически организует режим работы только с возможностью чтения. Однако это не исключает возможности создания дополнительного кода для организации прямой записи данных в связанное поле базы данных посредством этого элемента управления. Но если вам нужен доступ только в режиме для чтения, достаточно просто оставить параметр режима записи `write` таким, как он есть.

В листингах 9.57 и 9.58 показано объявление этих свойств и соответствующих методов чтения и записи.

Листинг 9.57. Объявление класса TDBMaskEdit в заголовочном файле

```
class PACKAGE TDBMaskEdit : public TMaskEdit
{
private:
...
   AnsiString __fastcall GetDataField(void);
   TDataSource*__fastcall GetDataSource(void);
   void __fastcall SetDataField(AnsiString pDataField);
   void __fastcall SetDataSource(TDataSource *pDataSource);
...
__published:
   __property AnsiString DataField = {read = GetDataField,
                                      write = SetDataField,
                                      nodefault};
   __property TDataSource *DataSource = {read = GetDataSource,
                                         write = SetDataSource,
                                         nodefault};
};
```

Листинг 9.58. Методы компонента TDBMaskEdit

```
AnsiString __fastcall TDBMaskEdit::GetDataField(void)
{
   return(FDataLink->FieldName);
}

TDataSource *__fastcall TDBMaskEdit::GetDataSource(void)
{
   return(FDataLink->DataSource);
}

void __fastcall TDBMaskEdit::SetDataField(AnsiString pDataField)
{
   FDataLink->FieldName = pDataField;
}

void __fastcall TDBMaskEdit::SetDataSource(TDataSource *pDataSource)
{
   if(pDataSource != NULL)
   pDataSource->FreeNotification(this);
   FDataLink->DataSource = pDataSource;
}
```

Здесь потребуется разъяснить только назначение метода FreeNotification() класса pDataSource. В C++Builder поддерживается внутренний список объектов, чтобы остальные объекты могли быть уведомлены о его удалении. Метод FreeNotification() вызывается ав-

томатически для компонентов той же формы, но в данном случае существует вероятность, что компоненты другой формы (например модуль данных) также может быть связан с ним. Поэтому метод FreeNotification() нужен для гарантии того, что объект добавлен во внутренние списки других форм.

Инициализация связи с источником данных

Здесь может сложиться ложное впечатление о том, что компонент уже полностью готов. При попытке компиляции этого компонента и добавления его в форму разработчик сможет обнаружить сообщения о нарушениях доступа в окне Object Inspector для свойств DataField и DataSource. Причина в том, что не был инициализирован внутренний объект FieldDataLink.

В раздел открытых переменных заголовочного класса файла следует добавить следующую строку.

```
__fastcall ~TDBMaskEdit(void);
```

А в конструктор и деструктор этого компонента следует добавить приведенный ниже код.

```
__fastcall TDBMaskEdit::TDBMaskEdit(TComponent*Owner)
   : TMaskEdit(Owner)
{
   FDataLink = new TFieldDataLink();
   FDataLink->Control = this;
}

__fastcall TDBMaskEdit::~TDBMaskEdit(void)
{
   if(FDataLink)
   {
      FDataLink->Control = 0;
      FDataLink->OnUpdateData = 0;
      delete FDataLink;
   }
}
```

Свойство Control класса FDataLink имеет тип TComponent. Это свойство может иметь в качестве значения компонент, который использует объект TFieldDataLink для установления связи с объектом TField. Для свойства Control следует указать значение this, чтобы указать, что именно этот компонент отвечает за связь.

Доступ к объекту TObject достигается за счет создания свойства, используемого только для чтения. Добавьте это свойство в раздел открытых членов в определении класса.

```
__property TField *Field = {read = GetField};
```

Добавьте объявление метода GetField в раздел закрытых членов класса.

```
TField *__fastcall GetField(void);
```

В исходный код добавьте следующий код.

```
TField *__fastcall TDBMaskEdit::GetField(void)
{
   return(FDataLink->Field);
}
```

Использование события OnDataChange

Итак, мы создали компонент, который может связаться с источником данных, но не может реагировать на изменения данных. Попробуем создать код, который позволит элементу управления реагировать на изменения в поле, например на перемещение к новой записи.

Классы соединения с данными имеют событие OnDataChange, которое вызывается при изменении данных. Для того чтобы компонент мог отвечать на эти изменения, создадим метод и свяжем его с событием OnDataChange.

На заметку

TDataLink — это вспомогательный класс, который используется объектами, связанными с данными. Более подробные сведения об использовании его свойств, методов и событий можно найти в справке C++Builder.

Событие OnDataChange имеет тип TNotifyEvent, поэтому метод нужно создать с таким же прототипом. В раздел закрытых членов в объявлении класса TDBMaskEdit в заголовочном файле компонента добавим следующий фрагмент кода.

```
class PACKAGE TDBMaskEdit : public TMaskEdit
{
private:
   //...
   void __fastcall DataChange(TObject *Sender);
}
```

В конструкторе метод DataChange() нужно связать с событием OnDataChange, а также удалить эту связь в деструкторе компонента.

```
__fastcall TDBMaskEdit::TDBMaskEdit(TComponent*Owner)
   : TMaskEdit(Owner)
{
   FDataLink = new TFieldDataLink();
   FDataLink->Control = this;
   FDataLink->OnDataChange = DataChange;
}

__fastcall TDBMaskEdit::~TDBMaskEdit(void)
{
   if(FDataLink)
   {
      FDataLink->Control = 0;
      FDataLink->OnUpdateData = 0;
      FDataLink->OnDataChange = 0;
      delete FDataLink;
   }
}
```

Наконец, определим метод DataChange() так, как показано ниже.

```
void __fastcall TDBMaskEdit::DataChange(TObject *Sender)
{
   if(!FDataLink->Field)
   {
      if(ComponentState.Contains(csDesigning))
```

```
        Text = Name;
      else
        Text = "";
   }
   else
      Text = FDataLink->Field->AsString;
}
```

Метод `DataChange()` сначала проверяет наличие связи с источником данных (и полем). Если указатель связи равен нулю, то для свойства `Text` (член унаследованного компонента) при выполнении задается пустая строка, а при создании — имя элемента управления. Если поле с данными указано правильно, для свойства `Text` в качестве значения задается содержимое этого поля с помощью свойства `AsString` объекта `TField`.

Таким образом, мы создали так называемый элемент управления для просмотра данных, потому что он способен только отображать изменения данных, которые происходят в источнике данных. Приступим теперь к превращению этого компонента в элемент управления с возможностью редактирования данных.

Создание элемента управления с возможностью редактирования данных

Нам придется создать код для обработки событий, связанных с манипуляциями клавишами и мышью. Это позволит переносить любые изменения данных из элемента управления в базовое поле в связанной базе данных.

Свойство `ReadOnly`

Если пользователь располагает элементом управления с возможностью редактирования данных, то он, конечно же, захочет применить его НЕ только в режиме для чтения. По умолчанию свойство `ReadOnly` класса `TMaskEdit` (класса-наследника) имеет значение `false`, поэтому нам не придется изменять его. Однако при создании компонента с пользовательским свойством `ReadOnly` следует убедиться в том, что принимаемое по умолчанию значение этого свойства равно `false`.

События клавиатуры и мыши

В файле `controls.hpp` в коде класса `TMaskEdit` можно найти защищенные методы `KeyDown()` и `MouseDown()`. Эти методы отвечают на соответствующие оконные сообщения (`KEYDOWN`, `WM_LBUTTONDOWN`, `WM_MBUTTONDOWN` и `WM_RBUTTONDOWN`) и вызывают соответствующее событие, если оно определено пользователем.

Для переопределения этих методов в `TDBMaskEdit` следует создать методы `KeyDown()` и `MouseDown()` на основе следующих объявлений в файле `controls.hpp`.

```
virtual void __fastcall MouseDown(TMouseButton,
                                  TShiftState Shift,
                                  int X, int Y);
virtual void __fastcall KeyDown(unsigned short &Key,
                                TShiftState Shift);
```

Их исходное объявление можно найти в файле `controls.hpp` (или файлах справки).

После этого создадим код, представленный в листинге 9.59.

Листинг 9.59. Методы `MouseDown()` и `KeyDown()`

```
void __fastcall TDBMaskEdit::MouseDown(TMouseButton Button,
                                       TShiftState Shift,
                                       int X, int Y)
{
   if(!ReadOnly &&FDataLink->Edit())
      TMaskEdit::MouseDown(Button, Shift, X, Y);
   else
   {
      if(OnMouseDown)
         OnMouseDown(this, Button, Shift, X, Y);
   }
}

void __fastcall TDBMaskEdit::KeyDown(unsigned short &Key,
                                     TShiftState Shift)
{
   Set<unsigned short, VK_PRIOR, VK_DOWN> Keys;
   Keys = Keys << VK_PRIOR << VK_NEXT << VK_END << VK_HOME <<
                  VK_LEFT << VK_UP << VK_RIGHT << VK_DOWN;

   if(!ReadOnly && (Keys.Contains(Key)) && FDataLink->Edit())
      TMaskEdit::KeyDown(Key, Shift);
   else
   {
      if(OnKeyDown)
         OnKeyDown(this, Key, Shift);
   }
}
```

В обоих случаях нужно убедиться в том, что компонент используется не только в режиме для чтения и `FieldDataLink` может использоваться для редактирования. Метод `KeyDown()` также обрабатывает действия клавиш курсора (определенные в файле `winuser.h`). Когда все проверки закончены, поле может редактироваться; и для этого вызывается унаследованный метод. Этот метод автоматически вызовет появление связанного события, если таковое определено. Если поле не может редактироваться, то выполняется пользовательское событие (если таковое существует).

Обновление набора данных

Изменение содержимого элемента управления, связанного с данными, должно быть отражено в поле. Аналогично, при изменении значения поля, оно также должно быть отражено в связанном с данными элементе управления.

Элемент управления `TDBMaskEdit` уже имеет метод `DataChange()`, который вызывается событием `OnDataChange` типа `TFieldDataLink`. Именно этот метод отражает изменение значения поля в элементе управления `TDBMaskEdit`, т.е. воплощает первый из указанных сценариев.

Теперь нужно создать код для обновления значения в поле после изменения содержимого элемента управления. Класс `TFieldDataLink` имеет событие `OnUpdateData`, с помощью которого связанный с данными элемент управления может записывать в набор данных любые из-

менения. Создадим метод UpdateData() в классе TDBMaskEdit и свяжем его с событием OnUpdateData класса TFieldDataLink.

Добавьте объявление метода UpdateData() в элемент управления TDBMaskEdit, как показано в следующем примере кода.

```
void __fastcall UpdateData(TObject *Sender);
```

В конструкторе класса присвойте этот метод событию OnUpdateData типа TFieldDataLink, как показано ниже.

```
__fastcall TDBMaskEdit::TDBMaskEdit(TComponent*Owner)
   : TMaskEdit(Owner)
{
   FDataLink = new TFieldDataLink();
   FDataLink->Control = this;
   FDataLink->OnUpdateData = UpdateData;
   FDataLink->OnDataChange = DataChange;
}
```

В качестве значения поля зададим содержимое элемента управления TDBMaskEdit:

```
void __fastcall TDBMaskEdit::UpdateData(TObject *Sender)
{
   if(FDataLink->CanModify)
      FDataLink->Field->AsString = Text;
}
```

Элемент управления TDBMaskEdit — наследник класса TMaskEdit, который, в свою очередь, является наследником класса TCustomEdit. Этот класс имеет защищенный метод Change(), который вызывается системными событиями Windows. Он, в свою очередь, возбуждает событие OnChange.

Переопределим метод Change() так, чтобы он обновлял набор данных перед вызовом унаследованного метода. В разделе защищенных членов класса TDBMaskEdit добавим объявление следующего метода.

```
DYNAMIC void __fastcall Change(void);
```

А также добавим его код, который показан в листинге 9.60.

Листинг 9.60. Метод Change()

```
void __fastcall TDBMaskEdit::Change(void)
{
   if(FDataLink)
   {
      // Следует определить, не находится ли источник данных
      // в режиме редактирования. Если - нет, то нужно сохранить
      // текущее значение, потому что перевод в этот режим
      // приведет к замене текущего значения тем значением,
      // которое в данный момент хранится в таблице.
      AnsiString ChangedValue = Text;

      // Получение координат курсора.
      int CursorPosition = SelStart;
```

```
      // Переход в режим редактирования.
      if(FDataLink->CanModify && FDataLink->Edit())
      {
         Text = ChangedValue; // Если до этого не были в режиме
                              // редактирования
         SelStart = CursorPosition;
         FDataLink->Modified(); // Внесение изменения (источник
                                // данных не переводится в режим
                                // редактирования)
      }
   }

   TMaskEdit::Change ();
}
```

Это изменение уведомляет класс `TFieldDataLink` о том, что внесены изменения, а затем вызывает унаследованный метод `Change()`.

Заключительный этап заключается в обработке ситуации, когда фокус ввода покидает элемент управления. Класс `TWinControl` отвечает на сообщение `CM_EXIT`, генерируя событие `OnExit`.

В ответ на это сообщение может быть вызван метод обновления записи в связанном наборе данных. Для этого создается карта сообщений в классе `TDBMaskEdit`. В разделе закрытых членов добавьте следующий код.

```
void __fastcall CMExit(TWMNoParams Message);

BEGIN_MESSAGE_MAP
   MESSAGE_HANDLER(CM_EXIT, TWMNoParams, CMExit)
END_MESSAGE_MAP(TMaskEdit)
```

Согласно карте сообщений, метод `CMExit()` будет вызван в ответ на сообщение `CM_EXIT` с соответствующей информацией, переданной в структуре `TWMNoParams`.

Ниже показан код метода `CMExit()`.

```
void __fastcall TDBMaskEdit::CMExit(void)
{
   try
   {
      ValidateEdit();
      if(FDataLink && FDataLink->CanModify)
         FDataLink->UpdateRecord();
   }

   catch(...)
   {
      SetFocus();
      throw;
   }
}
```

При этом содержимое поля будет проверено по отношению к заданной маске. Если источник данных может быть обновлен, в наборе записей будет обновлена соответствующая запись. При возникновении исключительной ситуации курсор будет возвращен в тот элемент управления, который вызвал ее появление, а сама она будет обработана приложением.

Создание заключительного сообщения

В C++Builder предусмотрен компонент TDBCtrlGrid, в котором записи из источника данных отображаются в виде бланка произвольного формата. При попытке обновления источника данных со стороны этого компонента он пошлет сообщение CM_GETDATALINK. Поискав это сообщение в заголовочных файлах C++Builder, его можно обнаружить в картах сообщений, заданных для всех элементов управления, которые предназначены для работы с базой данных. Изучив соответствующий файл с расширением .pas, в нем можно найти следующий обработчик сообщения.

```
procedure TDBEdit.CMGetDataLink(var Message:TMessage);
begin
   Message.Result := Integer(FDataLink);
end;
```

Для организации такой поддержки в нашем компоненте нужно создать карту сообщений, объявить метод и реализовать обработчик сообщения.

Объявление разместим в разделе закрытых членов класса.

```
void __fastcall CMGetDataLink(TMessage Message);
```

В разделе отрытых членов класса следующим образом изменим карту сообщений.

```
BEGIN_MESSAGE_MAP
   MESSAGE_HANDLER(CM_EXIT, TWMNoParams, CMExit)
   MESSAGE_HANDLER(CM_GETDATALINK, TMessage, CMGetDataLink)
END_MESSAGE_MAP(TMaskEdit)
```

Наконец, реализуем этот метод в исходном коде.

```
void __fastcall TDBMaskEdit::CMGetDataLink(TMessage Message)
{
   Message.Result = (int)FDataLink;
}
```

Вот и все. Мы получили элемент управления, который связан с данными и функционирует так же, как и любой другой элемент управления.

Регистрация компонентов

Регистрация компонентов представляет собой достаточно простой и многостадийный процесс. На первом этапе необходимо убедиться в том, что компонент, инсталлируемый в палитре компонентов Component Palette, не содержит виртуальных (или динамических DYNAMIC) функций. Иначе говоря, не содержит функций следующего вида.

```
virtual ВозвращаемыйТип __fastcall
   ИмяФункции (СписокПараметров) = 0;
```

Обратите внимание, что ключевое слово __fastcall не требуется для виртуальных функций, но оно используется в членах-функциях компонентов. Вот почему оно здесь показано.

Наличие виртуальных функций можно проверить вручную, проверив определение класса компонента, или вызвав функцию ValidCtrCheck() и передав ей в качестве аргумента указатель на это компонент. Функция ValidCtrCheck() при этом размещается в каком-то месте исходного кода. Например, для компонента TCustomComponent она может иметь следующий вид.

```
static inline void ValidCtrCheck(TCustomComponent *)
{
   new TCustomComponent(NULL);
}
```

Единственное назначение этой функции заключается в создании экземпляра класса TCustomComponent. Так как не допускается создание экземпляра для класса с виртуальной функцией, компилятор выдаст следующие сообщения об ошибках компиляции: E2352 Cannot create instance of abstract class 'TCustomComponent' (E2352 Нельзя создать экземпляр абстрактного класса) и E2353 Class 'TCustomComponent' is abstract because of 'function' (E2353 Класс 'TCustomComponent' является абстрактным из-за наличия в нем функции 'функция').

Во втором сообщении будет указано имя виртуальной функции. Обе эти ошибки связаны с наличием следующей строки кода.

```
new TCustomComponent(NULL);
```

Разработчику не всегда нужна эта функция, поскольку он вряд ли случайно создаст виртуальную функцию. Однако при использовании IDE-среды для создания нового компонента эта функция автоматически добавляется в исходный код. В таком случае ее рекомендуется оставить.

Определив, что компонент не является абстрактным базовым классом и для него можно создать экземпляр, можно приступать к созданию кода регистрации. Для этого нужно создать функцию Register(), которая *должна* быть заключена в пространство имен с тем же названием, что и имя файла, в котором она находится. При этом следует строго соблюдать следующее соглашение об именах: первый символ имени пространства имен должен быть прописным, а все остальные — строчными. Таким образом, функция Register() будет иметь следующий формат.

```
namespace Имя_файла_в_котором_она_находится
{
   void __fastcall PACKAGE Register()
   {
      // Код регистрации.
   }
}
```

Не забудьте указать макрос PACKAGE перед именем функции Register(). Теперь остается только создать код регистрации одного или нескольких компонентов. Для этого нужно использовать функцию RegisterComponents(), которая имеет следующее объявление в файле $(BCB)\Include \Vcl \Classes.hpp.

```
extern PACKAGE void __fastcall
   RegisterComponents(const AnsiString Page,
                      TMetaClass* const *ComponentClasses,
                      const int ComponentClasses_Size);
```

Аргумент типа const AnsiString функции RegisterComponents() содержит имя вкладки палитры, когда должен быть инсталлирован компонент, а аргумент типа TMetaClass* const * — указатель на массив инсталлируемых компонентов. Если значение первого аргумента не соответствует ни одной вкладке палитры компонентов Component Palette, то в ней будет создана новая вкладка с этим именем. В случае необходимости значение аргумента может быть получено на основе строкового ресурса, что позволяет использовать разные строки для разных локализованных настроек операционной системы.

Открытый массив TMetaClass* имеет более сложную структуру. Его можно создать с помощью макроса OPENARRAY либо вручную. Рассмотрим оба подхода.

Допустим, требуется зарегистрировать компоненты TCustomComponent1, TCustomComponent2 и TCustomComponent3 в новой вкладке MyCustomComponents. Для этого сначала нужно получить для них указатели TMetaClass*. Это делается с помощью оператора __classid следующим образом: __classid(TCustomComponent1).

С помощью макроса OPENARRAY запишем функцию RegisterComponents() следующим образом.

```
RegisterComponents("MyCustomComponents",
                   OPENARRAY(TMetaClass*,
                   (__classid(TCustomComponent1),
                   __classid(TCustomComponent2),
                   __classid(TCustomComponent3))));
```

При этом тип TComponentClass можно использовать вместо TMetaClass*, потому что TMetaClass* является новым определением типа TComponentClass, которое в файле $(BCB)\Include \Vcl \Classes.hpp имеет следующий вид.

```
typedef TMetaClass* TComponentClass;
```

Макрос OPENARRAY более подробно рассматривается в разделе, посвященном регистрации свойств в категории, главы 10. Учтите, что в одном вызове функции RegisterComponents() допускается регистрация не более 19 аргументов (т.е. компонентов). Это вызвано тем, что аналогичные ограничения наложены на макрос OPENARRAY. Но это ограничение можно легко преодолеть.

Другой подход основан на объявлении и инициализации массива типа TMetaClass* (или TComponentClass) вручную:

```
TMetaClass Components[3] == {__classid(TCustomComponent1),
                             __classid(TCustomComponent2),
                             __classid(TCustomComponent3)};
```

Затем он передается функции RegisterComponents(), но при этом нужно также передать значение индекса последнего фактического члена массива, в данном случае — 2.

```
RegisterComponents("MyCustomComponents", Components, 2);
```

Сам вызов функции выглядит проще, но при этом велика вероятность передачи ошибочного значения для последнего аргумента.

Вот конечный вид кода функции Register().

```
namespace Имя_файла_в_котором_она_находится
{
   void __fastcall PACKAGE Register()
   {
      RegisterComponents("MyCustomComponents",
                         OPENARRAY(TMetaClass*,
                         (__classid(TCustomComponent1),
                         __classid(TCustomComponent2),
                         __classid(TCustomComponent3))));
   }
}
```

Напомним, что в функции Register() может находиться любое количество функций RegisterComponents(). Они могут понадобиться, например, для регистрации свойств и редакторов компонентов. Однако это уже тема следующей главы. Код регистрации компонента можно поместить в исходный код компонента, но обычно его стараются размещать отдельно. Более подробные сведения об этом можно найти в разделе, посвященном распространению компонентов, главы 11.

Резюме

Эта глава содержит достаточно обширный материал, связанный с пользовательскими компонентами. При создании пользовательских компонентов следует просмотреть диаграмму библиотеки VCL Chart и исходный код VCL, чтобы выбор базового класса для нового компонента был оптимальным. Здесь представлен простой пример расширения функциональных возможностей существующих компонентов на основе компонента `TStyleLabel` и способы создания невизуальных компонентов. Представлены также способы создания связанных с данными компонентов и взаимосвязи между ними.

Нетрудно заметить, что на создание собственного пользовательского компонента требуется затратить гораздо больше труда, чем на модификацию уже имеющегося компонента. Начните с самого простого и постарайтесь создать классы для обработки наиболее сложных частей.

Вы познакомились со способами перехвата системных сообщений Windows вашим компонентом, модификации предлагаемого по умолчанию обработчика базового класса, создания пользовательских событий, а также вызова этих событий внутри компонента. Постарайтесь создать максимально возможное количество свойств, методов и событий, которые могут понадобиться пользователям вашего компонента.

Независимо от того, создаете ли вы только готовые приложения, только компоненты или то и другое, для создания пользовательских компонентов требуется совершенно другой профессиональный уровень владения инструментами C++Builder. Чем сложнее создаваемые компоненты, тем более высокой квалификацией должны обладать их создатели. Создавая пользовательские компоненты, можно не только повысить уровень своих знаний об архитектуре библиотеки VCL, но и приобрести навыки работы с C++Builder и программирования на языке C++. Это позволит вам добиться полного контроля над пользовательским интерфейсом ваших приложений и сократить время их создания.

Глава

10

Создание редакторов свойств и компонентов

Джэйми Оллсоп

Как уже говорилось в главе 9, компоненты являются строительными блоками C++Builder и неотъемлемыми элементами любой C++Builder-программы. Большую часть своего времени разработчики посвящают работе с компонентами, изучению их свойств, а также попыткам их наиболее эффективного использования. В связи с этим усовершенствование интерфейса компонента, который используется при создании приложения, т.е. интерфейса времени создания (designtime interface) — один из наиболее оптимальных путей повышения эффективности использования компонента. Разработчики приложений прилагают значительные усилия для создания максимально удобного пользовательского интерфейса. Создатели компонентов также должны учитывать возможность применения созданного ими интерфейса другими пользователями.

В этой главе рассматриваются методы, необходимые для успешной реализации редакторов свойств и компонентов. Некоторые наиболее значительные усовершенствования IDE-среды C++Builder относятся к интерфейсу свойств и компонентов времени создания приложения, которые повышают производительность труда разработчиков. Все эти новые элементы и функциональные возможности среды C++Builder 5 подробно рассматриваются здесь, а также приводятся рекомендации наиболее эффективного их использования.

Эта глава посвящена вопросу, которым часто пренебрегают авторы других книг, — созданию компонентов. Несомненно, создатели компонентов по достоинству оценят предлагаемый ценный учебный материал. Эта глава разбита на четыре части. В первой части описываются все аспекты редакторов свойств, включая новые возможности работы с изображениями. Затем рассматриваются редакторы компонентов. Далее излагается логическая последовательность действий при их создании. В третьей части описываются способы использования ресурсов в редакторах. Эта тема часто упоминается, но редко сопровождается действительно полезными рекомендациями. В заключение описываются категории свойств, способ регистрации отдельных категорий, а также подробно рассматривается создание пользовательских категорий. Эта глава, несомненно, очень полезна для создателей компонентов.

Код всех редакторов свойств и редакторов компонентов, которые рассматриваются в этой главе, содержится на прилагаемом к книге компакт-диске. Изучая его, можно разобраться во всех аспектах создания пользовательских редакторов компонентов. Представленный в этой главе код содержится в двух пакетах: пакете времени создания приложения `EnhancedEditors` и пакетах времени создания и времени выполнения приложения `NewAdditionalComponents`. Пакет `NewAdditionalComponentsDTP.bpk` времени создания приложения содержит код редакторов свойств и компонентов, а также код регистрации. А пакет `NewAdditionalComponentsRTP.bpk` времени выполнения приложения содержит код компонентов.

Прежде чем приступить к изучению этой главы, рекомендуется установить пакеты в используемой вами среде C++Builder 5, чтобы поближе познакомиться с результатами их работы. Перед инсталляцией пакета скопируйте на жесткий диск каталог `Chapter10Packages`, который содержит нужные нам файлы. Новому каталогу можно присвоить более информативное имя. Затем скопируйте файлы времени выполнения из каталога `Chapter10Packages\System` в системный каталог операционной системы Windows, например в каталог `Windows\System` на компьютере под управлением операционной системы Windows 9x или `WINNT\System32` на компьютерах под управлением операционной системы Windows NT и Windows 2000. Эти файлы нужны для работы двух пакетов времени создания. Для обоих пакетов времени создания потребуется файл с пакетом времени выполнения `TNonVCLTypeInfoPackage.bpl`, а для пакета времени создания `NewAdditionalComponentsDTP.bpl` также требуется файл с пакетом времени выполнения `NewAdditionalComponentsRTP.bpl`.

Для инсталляции пакета `EnhancedEditors` запустите среду C++Builder 5 и выберите команду меню Component⇨Install Packages. Щелкните на кнопке Add и укажите файл `EnhancedEditors.bpl`. После щелчка на кнопке Open диалоговое окно Add Design Package закроется и пакет будет представлен в списке Design packages под названием `Enhanced`

Property and Component Editors. Для завершения работы с этим диалоговым окном щелкните на кнопке OK. В табл. 10.1 перечислены редакторы свойств и компонентов, которые содержатся в пакете EnhancedEditors вместе с информацией об их регистрации (т.е. инсталляции) в IDE-среде, где инсталлирован данный компонент.

Таблица 10.1. Редакторы свойств и компонентов, зарегистрированные в пакете EnhancedEditors

Редакторы	Регистрация
TShapeTypePropertyEditor	Есть
TImageListPropertyEditor	Есть
TImageIndexProperty	Нет — абстрактный базовый класс
TPersistentDerivedImageIndexProperty	Есть
TComponentDerivedImageIndexProperty	Есть
TMenuItemImageIndexProperty	Есть
TTabSheetImageIndexProperty	Не требуется
TToolButtonImageIndexProperty	Есть
TCoolBandImageIndexProperty	Не требуется
TListColumnImageIndexProperty	Есть
TCustomActionImageIndexProperty	Не требуется
THeaderSectionImageIndexProperty	Не требуется
TDisplayCursorProperty	Есть
TDisplayFontNameProperty	Есть
TUnsignedProperty	Есть
TCharPropertyEditor	Есть
TSignedCharProperty	Есть
TUnsignedCharProperty	Есть
TimageComponentEditor	Есть

Пакет NewAdditionalComponents инсталлируется точно так же, как и пакет EnhancedEditors. Выберите команду меню Component⇨Install Packages. Щелкните на кнопке Add и укажите файл NewAdditionalComponentsDTP.bpl. После щелчка на кнопке Open диалоговое окно Add Design Package закроется и пакет будет представлен в списке Design packages под названием New Components for the Additional Palette Page. Для завершения работы с этим диалоговым окном щелкните на кнопке OK. При этом в IDE-среде будут зарегистрированы компоненты TEnhancedImage и TFilterEdit этого пакета.

Кроме того, будет зарегистрирован редактор свойств TImageIndexPropertyEditor этого пакета и включен пакет только времени выполнения TNonVCLTypeInfoPackage.bpk. Последний содержит код, представленный в листингах 10.7 и 10.8 ниже в этой главе в разделе о получении информации о типе на основе заданного свойства и класса для объектов, тип которых отличен от типов библиотеки VCL. Этот пакет нужен для регистрации пакетов EnhancedEditors.bpk и NewAdditionalComponentsDTP.bpk. Следовательно, для перекомпиляции пакета IDE-среда во время компиляции и компоновки должна найти заголовочные файлы (.h) и файлы импорта (.bpi).

Создание пользовательских редакторов свойств

Один из лучших способов усовершенствования интерфейса компонента времени создания — создание удобных и интуитивно понятных редакторов свойств. В этом разделе рассматриваются основные принципы создания редакторов свойств. Все пользовательские редакторы свойств происходят от класса `TPropertyEditor`, который предоставляет основные функции, необходимые этому редактору для работы в IDE-среде. В листинге 10.1 показано определение класса `TPropertyEditor` из файла `$(BCB)\Include\Vcl\DsgnIntf.hpp`, где `$(BCB)` — каталог, в котором установлен C++Builder 5.

Листинг 10.1. Определение класса `TPropertyEditor`

```
class DELPHICLASS TPropertyEditor;
typedef void __fastcall (__closure
   *TGetPropEditProc)(TPropertyEditor*Prop);

class PASCALIMPLEMENTATION TPropertyEditor :
   public System::TObject
{
   typedef System::TObject inherited;
private:
   _di_IFormDesigner FDesigner;
   TInstProp *FPropList;
   int FPropCount;
   AnsiString __fastcall GetPrivateDirectory();
   void __fastcall SetPropEntry(int Index,
                                Classes::TPersistent* AInstance,
                                Typinfo::PPropInfo APropInfo);

protected:
   __fastcall virtual TPropertyEditor(
        const _di_IFormDesigner ADesigner,
        int APropCount);
   Typinfo::PPropInfo __fastcall GetPropInfo(void);
   Extended __fastcall GetFloatValue(void);
   Extended __fastcall GetFloatValueAt(int Index);
   __int64 __fastcall GetInt64Value(void);
   __int64 __fastcall GetInt64ValueAt(int Index);
   Sysutils::TMethod __fastcall GetMethodValue();
   Sysutils::TMethod __fastcall GetMethodValueAt(int Index);
   int __fastcall GetOrdValue(void);
   int __fastcall GetOrdValueAt(int Index);
   AnsiString __fastcall GetStrValue();
   AnsiString __fastcall GetStrValueAt(int Index);
   Variant __fastcall GetVarValue();
   Variant __fastcall GetVarValueAt(int Index);
   void __fastcall Modified(void);
   void __fastcall SetFloatValue(Extended Value);
   void __fastcall SetMethodValue(const Sysutils::TMethod &Value);
```

```
   void __fastcall SetInt64Value(__int64 Value);
   void __fastcall SetOrdValue(int Value);
   void __fastcall SetStrValue(const AnsiString Value);
   void __fastcall SetVarValue(const Variant &Value);
public:
   __fastcall virtual ~TPropertyEditor(void);
   virtual void __fastcall Activate(void);
   virtual bool __fastcall AllEqual(void);
   virtual bool __fastcall AutoFill(void);
   virtual void __fastcall Edit(void);
   virtual TPropertyAttributes __fastcall GetAttributes(void);
   Classes::TPersistent* __fastcall GetComponent(int Index);
   virtual int __fastcall GetEditLimit(void);
   virtual AnsiString __fastcall GetName();
   virtual void __fastcall GetProperties(TGetPropEditProc Proc);
   Typinfo::PTypeInfo __fastcall GetPropType(void);
   virtual AnsiString __fastcall GetValue();
   AnsiString __fastcall GetVisualValue();
   virtual void __fastcall GetValues(Classes::TGetStrProc Proc);
   virtual void __fastcall Initialize(void);
   void __fastcall Revert(void);
   virtual void __fastcall SetValue(const AnsiString Value);
   bool __fastcall ValueAvailable(void);
   DYNAMIC void __fastcall ListMeasureWidth(
      const AnsiString Value,
      Graphics::TCanvas* ACanvas,
      int& AWidth);
   DYNAMIC void __fastcall ListMeasureHeight(
      const AnsiString Value,
      Graphics::TCanvas* ACanvas,
      int& AHeight);
   DYNAMIC void __fastcall ListDrawValue(
      const AnsiString Value,
      Graphics::TCanvas* ACanvas,
      const Windows::TRect& ARect,
      bool ASelected);
   DYNAMIC void __fastcall PropDrawName(
      Graphics::TCanvas* ACanvas,
      const Windows::TRect& ARect,
      bool ASelected);
   DYNAMIC void __fastcall PropDrawValue(
      Graphics::TCanvas* ACanvas,
      const Windows::TRect& ARect,
      bool ASelected);
   __property _di_IFormDesigner Designer = {read=FDesigner};
   __property AnsiString PrivateDirectory =
      {read=GetPrivateDirectory};
   __property int PropCount = {read=FPropCount, nodefault};
   __property AnsiString Value = {read=GetValue,
                                  write=SetValue};
   };
```

Для настройки этого редактора нужно переопределить одну или несколько виртуальных или динамических (DYNAMIC) функций класса TPropertyEditor. Сэкономить время и усилия можно, создав пользовательский редактор свойств на основе наиболее подходящего класса редакторов свойств. На рис. 10.1 показана иерархия наследников класса TPropertyEditor. В серых прямоугольниках показаны классы-наследники, в которых переопределены способы перерисовки класса TPropertyEditor. Более подробно мы поговорим об этом в разделе, посвященном использованию изображений в редакторах свойств.

Иерархическую структуру, показанную на рис. 10.1, удобно использовать для выбора базового редактора свойств. Назначение каждого из них, за некоторыми исключениями, вполне очевидно из названия. В табл. 10.2 кратко описаны наиболее часто встречающиеся редакторы свойств.

Таблица 10.2. Краткое описание наиболее часто встречающихся редакторов свойств

Класс редактора свойств	Назначение
TCaptionProperty	Редактор всех именованных свойств Caption и Text типа AnsiString. Например, свойство Caption класса TForm и свойство Text класса TEdit. Разница между этим редактором свойств и редактором TStringProperty, от которого он происходит, заключается в том, что редактируемый компонент постоянно обновляется при редактировании свойства. А при использовании редактора TStringProperty обновление происходит только по окончании редактирования
TCharProperty	Используемый по умолчанию редактор всех свойств типа char и подчиненных типов char. Отображает символ значения свойства или само значение с префиксом #. Пример: свойство PasswordChar(char) класса TMaskEdit
TClassProperty	Используемый по умолчанию редактор всех свойств классов, производных от класса Tpersistent. Опубликованные свойства этого класса отображаются как подчиненные свойства после щелчка на символе + перед именем свойства. Пример: свойство Constraints(TSize Constraints*) класса Tform
TColorProperty	Используемый по умолчанию редактор свойств типа TColor. Отображает цвет в виде значения cl*XXX* (если оно существует) или в шестнадцатеричном формате (т.е. в BGR-формате: 0x00*BBGGRR*). Это значение может быть введено в формате cl*XXX* либо в числовом формате. Значение cl*XXX* можно выбрать из списка. После двойного щелчка на имени свойства на экране откроется диалоговое окно Color. Пример: свойство Color(TColor) класса TForm
TComponentProperty	Используемый по умолчанию редактор для указателей на объекты, производные от класса TComponent. Редактор отображает раскрывающийся список объектов совместимого типа, которые появляются в той же форме, что и редактируемый компонент. Пример: свойство Images(TcustomImageList*) класса TToolBar
TCursorProperty	Используется для свойств типа TCursor. Позволяет из списка имен курсоров выбрать курсор с его изображением. Пример: свойство Cursor(TCursor) класса TForm

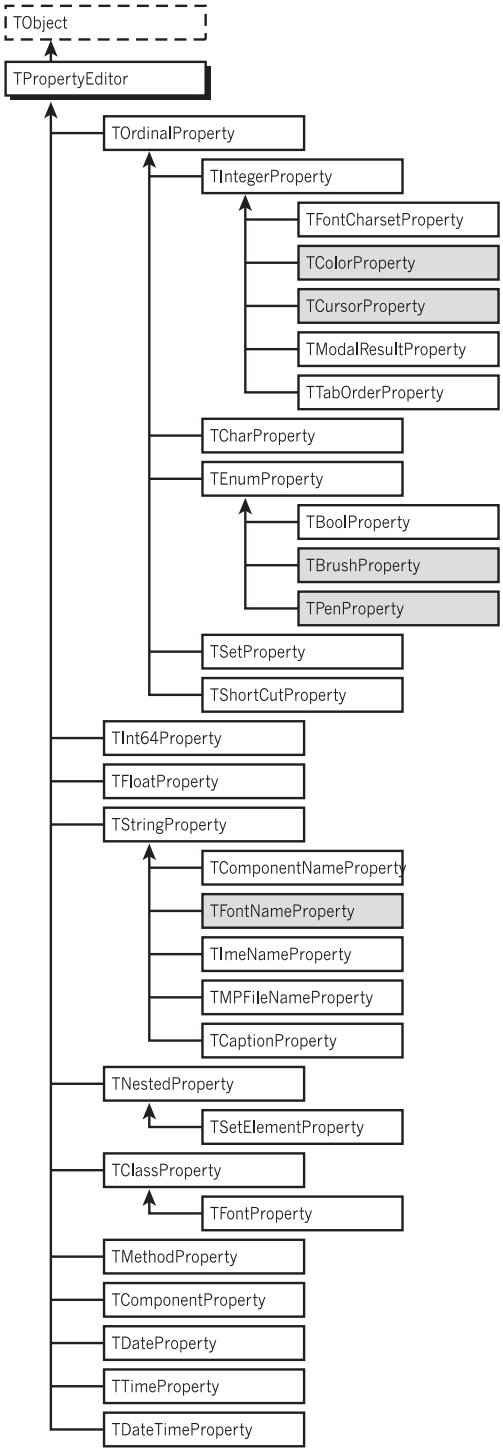


Рис. 10.1. Иерархия наследников класса `TPropertyEditor`

Окончание табл. 10.2

Класс редактора свойств	Назначение
TEnumProperty	Используемый по умолчанию редактор для всех пронумерованных свойств. В раскрывающемся списке отображаются значения, которые может иметь данное свойство. Пример: свойства Align(TAlign) и BorderStyle(TFormBorderStyle) класса TForm
TFloatProperty	Используемый по умолчанию редактор для свойств числового типа с плавающей запятой, а именно double, long double и float. Примеры: PrintLeftMargin(double) и PrintRightMargin(double) класса TF1Book
TFontProperty	Для свойств типа TFont. Этот редактор позволяет редактировать параметры шрифта с помощью диалогового окна Font (которое открывается после щелчка на кнопке с многоточием) или с помощью раскрывающегося списка подчиненных свойств. Пример: свойство Font(TFont) класса TForm
TIntegerProperty	Используемый по умолчанию редактор для всех свойств типа int. Примеры: свойства Height(int) и Width(int) класса TForm
TMethodProperty	Используемый по умолчанию редактор для свойств типа указатель-на-метод (функцию-член), то есть для событий. Этот редактор отображает раскрывающийся список обработчиков событий для всех событий того же типа, что и заданное свойство. Примеры: события OnClick и OnClose класса TForm
TOrdinalProperty	Все редакторы свойств порядкового типа, например TIntegerProperty, TCharProperty, TEnumProperty и TSetProperty, являются наследниками этого редактора
TPropertyEditor	Базовый класс для всех редакторов свойств
TSetElementProperty	Используется для редактирования отдельных элементов класса Set. Для этого свойства может быть задано значение true, если элемент принадлежит набору Set, или значение false в противном случае
TSetProperty	Используемый по умолчанию редактор для всех свойств типа Set. Каждый элемент набора Set отображается в виде подчиненного свойства для свойства Set, что позволяет легко удалять или добавлять в этот набор нужные элементы. Примеры: свойства Anchors(TAnchors) и BorderIcons(TBorderIcons) класса TForm
TStringProperty	Используемый по умолчанию редактор для всех свойств типа AnsiString. Примеры: свойства Hint и Name класса TForm

Выбор оптимального редактора определяется предъявляемыми к нему требованиями. Действительно, самой трудной частью в процессе создания пользовательского редактора свойств является определение его поведения. Этот вопрос подробно рассматривается в следующем разделе.

Процесс создания нового редактора свойств состоит из следующих этапов.

1. Точное определение предполагаемого поведения редактора свойств. Редакторы свойств часто создаются таким образом, чтобы предложить пользователю ограниченный набор

вариантов, который гарантирует соответствующее поведение компонента и интуитивно понятный интерфейс. На этом этапе должны быть определены рамки таких ограничений, например, допустить ли выбор только дискретных и заданных значений.

2. Определение необходимости создания пользовательского редактора свойств. Изменив способ использования свойства, можно добиться желаемого эффекта без необходимости создания пользовательского редактора свойств. В этой связи важно знать, какие редакторы свойств зарегистрированы для имеющихся типов свойств (см. табл. 10.2). Так как этот раздел посвящен созданию пользовательских редакторов свойств, то эта сторона процесса будет описана очень кратко. Вряд ли стоит еще раз подчеркивать, что никогда не помешают знания как можно большего количества существующих редакторов свойств и способов их работы. Прекрасным источником такой информации является файл `$(BCB)\Source\ToolsApi\DsgnIntf.pas`.
3. Тщательный выбор базового редактора свойств для создания пользовательского редактора свойств, который избавит разработчика от излишнего кодирования.
4. Выбор атрибутов свойств для данного редактора свойств.
5. Определение в родительском редакторе свойств функций, которые требуется переопределить.
6. Наконец, создание необходимого кода и его тестирование.

Сразу после принятия решения о создании пользовательского редактора свойств и выбора базового редактора свойств следует приступить к выбору атрибутов свойств. Каждый редактор свойств имеет метод `GetAttributes()`, который возвращает набор типа `TPropertyAttributes`. Он сообщает окну Object Inspector о способе использования свойства. Например, если свойство отображает раскрывающийся список значений, то разработчику следует убедиться в том, что значение `paValueList` содержится в наборе типа `TPropertyAttributes`, который возвращается методом `GetAttributes()`. Этот метод базового редактора свойств следует переопределить, если он не соответствует требованиям пользовательского редактора свойств. В табл. 10.3 показаны различные значения, которые могут содержаться в наборе `TPropertyAttributes`. В ней также указаны методы, которые следует переопределить для использования того или иного атрибута свойства.

Таблица 10.3. Значения, которые могут содержаться в наборе `TPropertyAttributes`

Значение	Описание
`paAutoUpdate`	Свойства с таким атрибутом автоматически обновляются при их изменении в окне Object Inspector; например, свойство `Caption` класса `TLabel`. Обычно свойство не обновляется до тех пор, пока не нажата клавиша <Enter> или фокус ввода не перемещен за пределы этого свойства. Для преобразования типа `AnsiString` к заданному типу и выполнения проверки корректности значения вызывается метод `SetValue()`. Для этого, вероятно, придется переопределить метод `SetValue(const AnsiString Value)`
`paDialog`	Свойства с этим атрибутом имеют кнопку с многоточием (`...`) с правой стороны от области расположения значения свойства. После щелчка на этой кнопке на экране появится диалоговое окно редактирования этого свойства. Это происходит вследствие вызова метода `Edit()` после щелчка на кнопке с многоточием. Для этого, вероятно, придется переопределить метод `Edit()`

Окончание табл. 10.3

Значение	Описание
paMultiSelect	Свойства с таким атрибутом могут редактироваться при выборе в форме нескольких компонентов. Например, редактор свойства Caption для компонентов TLabel и TButton имеет этот атрибут. При выборе в форме нескольких компонентов TLabel и TButton их свойства Caption могут редактироваться одновременно. В окне Object Inspector отображаются все свойства, редакторы которых имеют атрибут paMultiSelect, а имена и типы совпадают
paReadOnly	Свойства с таким атрибутом не могут редактироваться в окне Object Inspector
paRevertable	Свойства с таким атрибутом позволяют использовать команду Revert to inherited в контекстном меню окна Object Inspector, благодаря которой редактор свойств преобразует текущее значение свойства к значению, используемому по умолчанию
paSortList	Свойства с этим атрибутом содержат отсортированные списки значений в окне Object Inspector
paSubProperties	Свойства с этим атрибутом сообщают окну Object Inspector о том, что редактор свойств имеет подчиненные и редактируемые свойства. Перед именем такого свойства имеется символ +. Примером может служить редактор свойства TFont. Для того чтобы сообщить окну Object Inspector о наличии подчиненных свойств, следует переопределить метод GetProperties(TGetPropEditProc Proc)
paValueList	Свойства с этим атрибутом отображают раскрывающийся список возможных значений этого свойства. Кроме того, значение может быть введено вручную в пределах области допустимых значений. Например, так ведут себя свойства типа TColor. Для отображения списка значений в окне Object Inspector нужно переопределить метод GetValues(Classes::TGetStrProc Proc)

После выбора атрибутов редактора свойства сразу становится ясно, какие методы родительского редактора свойств должны быть переопределены. В зависимости от спецификации редактора свойств, может также потребоваться переопределить другие методы. В табл. 10.4 перечислены виртуальные и динамические (DYNAMIC) методы класса TPropertyEditor. Они сгруппированы и упорядочены в соответствии с их назначением.

Таблица 10.4. Виртуальные и динамические (DYNAMIC) методы класса TPropertyEditor

Метод	Описание
GetAttributes()	virtual TPropertyAttributes __fastcall GetAttributes(void); Возвращает набор типа TPropertyAttributes. Вызывается для указания атрибутов редактора свойств
GetValue()	virtual AnsiString __fastcall GetValue(); Возвращает строку типа AnsiString, которая представляет значение свойства. По умолчанию (т.е. в классе TPropertyEditor) возвращается значение (unknown). Следовательно, при создании прямого наследника класса TPropertyEditor необходимо переопределить этот метод для возвращения корректного значения

Продолжение табл. 10.4

Метод	Описание
`SetValue()`	`virtual void __fastcall SetValue(const AnsiString Value);` Вызывается для указания значения свойства. Этот метод может преобразовать строку типа `AnsiString` для значения свойства к нужному формату. При вводе некорректного значения метод `SetValue()` инициирует исключительную ситуацию с описанием этой ошибки. Обратите внимание, что метод `SetValue()` принимает в качестве параметра значение типа `const AnsiString` и не возвращает значения. Следовательно, обработка исключительных ситуаций является единственным способом работы с некорректными значениями
`Edit()`	`virtual void __fastcall Edit(void);` Вызывается после щелчка на кнопке с многоточием или двойного щелчка на самом свойстве (метод `GetAttributes()` должен возвращать атрибут `paDialog`). Он обычно используется для отображения формы, которая имеет интуитивно понятный интерфейс для редактирования значений свойств. Метод `Edit()` может вызывать методы `GetValue()` и `SetValue()` либо непосредственно считывать и записывать значение свойства. В этом случае должна быть организована проверка вводимых значений. Если будет введено неверное значение, то это приведет к возникновению исключительной ситуации с описанием данной ошибки

`GetValues()`	`virtual void __fastcall GetValues (Classes::TGetStrProc Proc);`
	Этот метод вызывается только в том случае, если методом `GetAttributes()` возвращается атрибут `paValueList`. Единственный параметр `Proc` имеет тип `TGetStrProc` (указатель на функцию-член экземпляра), который в файле `$(BCB)\Include\Vcl\Classes.hpp` имеет следующее объявление `typedef void __fastcall (__closure *TGetStrProc) (const AnsiString S);`
	Параметр `Proc` фактически является адресом метода с единственным параметром `S` типа `const AnsiString`, который передает строку типа `AnsiString` в раскрывающийся список этого редактора свойств. Метод `Proc(const AnsiString S)` следует вызвать один раз для каждого значения, которое должно быть отображено в раскрывающемся списке редактора свойств, например:
	`Proc(value1); // value1 имеет тип AnsiString`
	`Proc(value2); // value2 имеет тип AnsiString`
	и т.д.

`Activate()`	`virtual void __fastcall Activate(void);` Вызывается при выборе свойства в окне Object Inspector. Позволяет определить атрибуты только после выбора свойства (за исключением `paSubProperties` и `paMultiSelect`)
`AllEqual()`	`virtual bool __fastcall AllEqual(void);` Возвращает логическое значение. Вызывается только в том случае, когда атрибут `paMultiSelect` входит в состав атрибутов данного редактора свойств (когда он возвращается методом `GetAttributes()`). Он определяет равенство (и в этом случае возвращает `true`) нескольких выбранных свойств с одинаковыми именами и типами, для которых зарегистрирован данный редактор. В случае равенства для отображения этого значения вызывается метод `GetValue()`;в противном случае диапазон значений будет пуст

Продолжение табл. 10.4

Метод	Описание
`AutoFill()`	`virtual bool __fastcall AutoFill(void);` Возвращает логическое значение. Вызывается только в том случае, когда `paValueList` возвращается методом `GetAttributes()` и определяет возможность или невозможность (возвращает `true` или `false`) инкрементного выбора значений, возвращаемых методом `GetValues()`, в окне Object Inspector. По умолчанию возвращается значение `true`
`GetEditLimit()`	`virtual int __fastcall GetEditLimit(void);` Возвращает значение типа `int`, представляющее максимальное количество символов ввода, которые допускаются для этого свойства в окне Object Inspector. Переопределяя этот метод можно изменить это количество, которое по умолчанию равно 255
`GetName()`	`virtual AnsiString __fastcall GetName();` Возвращает значение типа `AnsiString`, которое используется окном Object Inspector для отображения имени свойства. Его следует переопределить, только если вместо имени свойства в окне Object Inspector требуется указать другое имя
`GetProperties()`	`virtual void __fastcall GetProperties(TGetPropEditProc Proc);` Этот метод необходимо переопределить, если нужно отобразить подчиненные свойства. Единственный параметр `Proc` типа `TGetPropEditProc` объявлен в файле `$(BCB)\Include\Vcl\DsgnIntf.hpp` в виде `typedef void __fastcall (__closure *TGetPropEditProc)(TPropertyEditor*Prop);` Параметр `Proc`, следовательно, является адресом метода с указателем на редактор свойств, производный от `TPropertyEditor`. Для каждого подчиненного свойства следует вызвать `Proc(TPropertyEditor* Prop)`, передавая в качестве аргумента указатель на редактор свойств, производный от `TPropertyEditor`. Например, `TSetProperty` переопределяет этот метод и передает указатель `TSetElementProperty` для каждого элемента набора. `TClassProperty` переопределяет метод `GetProperties()` с отображением подчиненных свойств для каждого опубликованного свойства класса
`Initialize()`	`virtual void __fastcall Initialize(void);` Вызывается окном Object Inspector после создания редактора свойств, но перед его использованием. При выборе сразу нескольких компонентов редакторы свойств создаются, но затем часто удаляются, поскольку они не используются. Этот метод позволяет задержать выполнение операций до тех пор, пока они не будут востребованы
`ListMeasureWidth()`	`DYNAMIC void __fastcall ListMeasureWidth(const AnsiString Value, Graphics::TCanvas* Acanvas, int& AWidth);` Вызывается во время вычисления ширины раскрывающегося списка свойства. Если наряду с текстом в списке размещаются изображения, этот метод следует переписать, чтобы гарантировать, что он достаточно широк

Окончание табл. 10.4

Метод	Описание
`ListMeasureHeight()`	`DYNAMIC void __fastcall ListMeasureHeight(const AnsiString Value, Graphics::TCanvas* ACanvas, int& AHeight);` Вызывается при вычислении длины раскрывающегося списка свойства. Если высота изображения больше высоты текста, этот метод следует переопределить для предотвращения обрезания рисунка
`ListDrawValue()`	`DYNAMIC void __fastcall ListDrawValue(const AnsiString Value, Graphics::TCanvas* ACanvas, const Windows::TRect& ARect, bool Aselected);` Вызывается для перерисовки текущего элемента списка в раскрывающемся списке свойства. Для организации перерисовки изображения метод придется переопределить. По умолчанию этот метод перерисовывает только текст текущего элемента списка

`PropDrawValue()`	`DYNAMIC void __fastcall PropDrawValue(Graphics::Tcanvas* ACanvas, const Windows::TRect& ARect, bool ASelected);`
	Вызывается в тех случаях, когда в окне Object Inspector нужно перерисовать значение свойства. Для организации перерисовки изображения этот метод придется переопределить
	`DYNAMIC void __fastcall PropDrawName(Graphics::Tcanvas* ACanvas, const Windows::TRect &ARect, bool ASelected);`
	Вызывается в тех случаях, когда в окне Object Inspector нужно перерисовать имя свойства. Для организации перерисовки изображения этот метод придется переопределить, но такая необходимость возникает крайне редко

Теперь читатель может легко представить себе возможности пользовательского редактора свойств. В следующих разделах мы рассмотрим наиболее важные методы и дадим рекомендации по их кодированию. Последние пять методов `ListMeasureWidth()`, `ListMeasureHeight()`, `ListDrawValue()`, `PropDrawValue()` и `PropDrawName()` связаны с перерисовкой изображений в окне Object Inspector и подробно рассматриваются в разделе, посвященном использованию изображений в редакторах свойств.

Наиболее часто переопределяются первые пять методов, приведенных в табл. 10.6, — `GetAttributes()`, `GetValue()`, `SetValue()`, `Edit()` и `GetValues()`. В листинге 10.2 показано определение класса пользовательского редактора свойств, производного от класса `TPropertyEditor`.

Ключевым фрагментом кода в листинге 10.2 является строка

```
typedef : typedef TPropertyEditor inherited;
```

Она позволяет использовать новый тип `inherited` в указании пространства имен (`namespace`) вместо типа `TPropertyEditor`. Такой способ часто используется при работе с библиотекой VCL, упрощая явный вызов родительских методов (в данном случае класса `TPropertyEditor`). При этом сохраняется простота его сопровождения. При изменении имени родительского класса разработчик должен будет изменить код только в одном месте.

Например, в код метода `GetAttributes()` редактора свойств можно включить следующую строку:

```
return inherited::GetAttributes() << paValueList >> paMultiSelect
```

В ней вызывается метод `GetAttributes()` базового класса редактора свойств, возвращающий множество значений `TPropertyAttributes`. При этом `paValueList` добавляется в это множество, `paMultiSelect` удаляется из него, а полученное в результате этих действий множество возвращается.

Листинг 10.2. Определение кода для пользовательского редактора свойств

```
class TCustomPropertyEditor : public TPropertyEditor
{
   typedef TPropertyEditor inherited;

public:
   virtual TPropertyAttributes __fastcall GetAttributes(void);
   virtual AnsiString __fastcall GetValue();
   virtual void __fastcall SetValue(const AnsiString Value);
   virtual void __fastcall Edit(void);
   virtual void __fastcall GetValues(Classes::TGetStrProc Proc);

protected:
   #pragma option push -w-inl
   inline __fastcall virtual
      TCustomPropertyEditor(const _di_IFormDesigner ADesigner,
                            int APropCount)
         : TPropertyEditor(ADesigner, APropCount)
   {}
   #pragma option pop

public:
   #pragma option push -w-inl
   inline __fastcall virtual ~TCustomProperty(void){}
   #pragma option pop
};
```

Метод `GetAttributes()`

Метод `GetAttributes()` реализуется достаточно просто, поскольку в нем придется заменить только те атрибуты, которые возвращаются родительским классом и непосредственно влияют на код. Остальные атрибуты следует оставить неизменными, поэтому можно создать действительно необходимые атрибуты и удалить все ненужные. При этом следует тщательно проверить атрибуты родительского класса. Возможно, их не придется изменять вообще. Допустим, редактор свойств, производный от класса `TPropertyEditor` необходим для отображения раскрывающегося списка значений, и его не следует использовать при выборе нескольких компонентов. В таком случае код метода `GetAttributes()` может иметь следующий вид.

```
TPropertyAttributes __fastcall
   TCustomPropertyEditor::GetAttributes(void)
{
   return
      inherited::GetAttributes() << paValueList >>
         paMultiSelect;
}
```

Если метод TPropertyEditor::GetAttributes() возвращает paRevertable, то код метода GetAttributes() может иметь следующий вид.

```
TPropertyAttributes __fastcall
   TCustomPropertyEditor::GetAttributes(void)
{
   return
      TPropertyAttributes() << paValueList <<
         paRevertable >> paMultiSelect;
}
```

Метод GetValue()

Метод GetValue() используется для возвращения строкового представления типа AnsiString для значения редактируемого свойства. Для этого следует использовать методы GetXxxValue() класса TPropertyEditor, где Xxx — Float, Int64, Method, Ord, Str или Var. Эти методы кратко описываются в табл. 10.5.

Таблица 10.5. Методы GetXxxValue() класса TPropertyEditor

Метод	Описание
GetFloatValue()	Возвращает значение типа Extended, т.е. long double. Используется для извлечения значений с плавающей запятой, например float, double и long double
GetInt64Value()	Возвращает значение типа __int64. Используется для извлечения значений свойства типа Int64(_int64)

GetMethodValue()	Возвращает структуру типа Tmethod:
	struct Tmethod
	{
	void *Code;
	void *Data;
	};
	Используется для извлечения значений свойства типа Closure, иначе говоря, события

GetOrdValue()	Возвращает значение типа int. Используется для извлечения значений свойства типа Ordinal, например char, signed char, unsigned char, int, unsigned, short и long. Может также использоваться для извлечения указателя; при этом тип int должен быть приведен к соответствующему типу указателя с помощью оператора reinterpret_cast
GetStrValue()	Возвращает значение типа AnsiString. Используется для извлечения значений свойства типа AnsiString
GetVarValue()	Возвращает значение типа Variant. Используется для извлечения значений свойства типа Variant. Класс Variant моделирует поведение вариантного типа Object Pascal. Более подробное описание этого типа можно найти в оперативной справке

Приведенный ниже код содержит реализацию метода GetValue() для извлечения значения свойства типа char с помощью метода GetOrdValue().

```
AnsiString __fastcall TCustomPropertyEditor::GetValue()
{
   char ch = static_cast<char>(GetOrdValue());
   if(ch >32 && ch < 128) return ch;
   else return AnsiString().sprintf("#%d", ch);

   // Обратите внимание, что символ # применяется здесь для
   // отображения специальных символов, которые не могут быть
   // отображены непосредственно. Таким образом в библиотеке
   // VCL отображаются непечатаемые символьные значения;
   // например, #8 означает знак возврата на одну позицию (\b).
}
```

Обратите внимание на использование ключевого слова static_cast для приведения возвращаемого типа int к типу char. Операторы явного приведения типа часто используются для переопределения методов GetValue() и SetValue() класса TPropertyEditor. Поэтому разработчику следует четко понимать, когда и зачем их использовать.

Метод SetValue()

Метод SetValue() используется для указания фактического значения свойства с преобразованием его строкового формата AnsiString к наиболее приемлемому. Для этого следует использовать один из методов SetXxxValue() класса TPropertyEditor, где Xxx — Float, Int64, Method, Ord, Str или Var. Эти методы кратко описаны в табл. 10.6.

Таблица 10.6. Методы SetXxxValue() класса TPropertyEditor

Метод	Описание
SetFloatValue()	Передает в качестве аргумента значение типа Extended, т.е. long double. Используется для установки значений с плавающей запятой, например float, double и long double
SetInt64Value()	Передает в качестве аргумента значение типа __int64. Используется для установки значений свойства типа Int64(_int64)
SetMethodValue()	Передает в качестве аргумента структуру типа Tmethod. Используется для установки значений свойства типа Closure, т.е. события
SetOrdValue()	Передает в качестве аргумента значение типа int. Используется для установки значений свойства типа Ordinal, например: char, signed char, unsigned char, int, unsigned, short и long. Может также использоваться для установки указателя; при этом тип int должен быть приведен к соответствующему типу указателя с помощью оператора reinterpret_cast
SetStrValue()	Передает в качестве аргумента значение типа AnsiString. Используется для установки значений свойства типа AnsiString
SetVarValue()	Передает в качестве аргумента значение типа Variant. Используется для установки значений свойства типа Variant

Прежде чем вызвать методы SetXxxValue(), метод SetValue() должен убедиться в корректности переданных ему значений; в противном случае генерируется исключительная ситуация. Класс исключения EPropertyError имеет смысл использовать в качестве базового класса для создания собственных производных классов исключений. Ниже приводится фрагмент кода для свойства типа int, в котором не допускается наличие отрицательных значений:

```
void __fastcall
   TCustomPropertyEditor::SetValue(const AnsiString Value)
{
   if(Value.ToInt()<0)
   {
      throw EPropertyError("The value must be greater than 0");
   }
   else SetOrdValue(Value.ToInt());
}
```

Метод Edit()

Метод Edit() обычно используется для создания более удобного интерфейса. Часто этот интерфейс имеет вид формы, функционирующей подобно диалоговому окну. Метод Edit() может также вызывать методы GetValue() и SetValue(), а также методы GetXxxValue() и SetXxxValue(). Здесь следует отметить, что класс TPropertyEditor и его производные классы имеют свойство Value, методы чтения и записи значений которых называются GetValue() и SetValue(). Вот как выглядит объявление этого свойства.

```
__property AnsiString Value = {read=GetValue, write=SetValue};
```

Это свойство может использоваться вместо явного вызова методов GetValue() и SetValue(). Независимо от способа вызова методов GetValue() и SetValue(), метод Edit() должен отобразить на экране соответствующую форму для интуитивно понятного редактирования значения данного свойства.

Для этого можно применить следующие два основных способа. В первом случае можно организовать автоматическое обновление значения свойства при его отображении в форме. В втором случае форму можно использовать как диалоговое окно для извлечения одного или нескольких значений, заданных пользователем, и последующей окончательной установки значения свойства после закрытия формы (т.е. после возвращения модального результата mrOK). Выбор одного из этих двух способов определяет вид метода Edit().

В первом случае используются два основных типа значений свойств: одно для представления единственного значения, например типа int; а другое — для представления набора значений, например класса TFont (хотя редактор свойства TFont ведет себя так, как во втором случае). Разница между ними заключается в способе использования класса Value для обновления данного свойства. Value является указателем в свойстве класса. Чтобы форма могла обновлять это свойство, она должна иметь адрес Value или адрес того объекта, на который Value указывает. Для свойства класса это можно выполнить достаточно просто; указатель класса считывается из значения Value, а значения класса редактируются с помощью этого указателя. Удобный способ организации такого поведения заключается в объявлении свойства того же типа, что и редактируемое свойство. Перед отображением формы его значение сравнивается со значением Value, что позволяет отображать и сохранять исходные значения.

Для одного редактируемого элемента свойства ссылка на Value должна передаваться в конструкторе формы. Использование ссылки на Value гарантирует, что при каждом изме-

нении Value будут вызываться методы GetValue() и SetValue(). При использовании этого подхода рекомендуется также сохранять значение свойства, которое оно имело при исходном отображении формы. Это позволяет отменить операцию редактирования с восстановлением исходного значения. Пример кода обработки такой ситуации приводится в листингах 10.3 и 10.4 для свойства класса и одного элемента редактируемого свойства.

Листинг 10.3. Код пользовательской формы для редактирования свойства-класса

```
// Сначала идет код формы TMyPropertyForm

// В ЗАГОЛОВОЧНОМ ФАЙЛЕ
//---------------------------------------------------------//
#ifndef MyPropertyFormH
#define MyPropertyFormH
//---------------------------------------------------------//
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include "HeaderDeclaringTPropertyClass"
//---------------------------------------------------------//
class TMyPropertyForm : public TForm
{
__published: // Компоненты, управляемые IDE-средой
private:
   TPropertyClass* FPropertyClass;
   // Другие объявленные здесь переменные используются для
   // восстановления исходных значений при нажатии кнопки Cancel

protected:
   void __fastcall SetPropertyClass(TPropertyClass* Pointer);
public:
   __fastcall TMyPropertyForm(TComponent* Owner);
   __property TPropertyClass*PropertyClass =
                {read=FPropertyClass, write=SetPropertyClass};

   // Другие объявления
};
//---------------------------------------------------------//
#endif

// В ФАЙЛЕ РЕАЛИЗАЦИИ
//---------------------------------------------------------//
#include <vcl.h>
#pragma hdrstop
#include "MyPropertyForm.h"
//---------------------------------------------------------//
#pragma package(smart_init)
#pragma resource "*.dfm"
//---------------------------------------------------------//
__fastcall TMyPropertyForm::TMyPropertyForm(TComponent* Owner)
```

```
             : TForm(Owner)
{
}
//---------------------------------------------------------//
void __fastcall
   TMyPropertyForm::SetPropertyClass(TPropertyClass* Pointer)
{
   FPropertyClass = Pointer;
   if(FPropertyClass != 0)
   {
      // Сохранение текущих значений свойства
   }
}
//---------------------------------------------------------//

// МЕТОД Edit()
#include "MyPropertyForm.h" // Запомните этот файл
void __fastcall TCustomPropertyEditor::Edit(void)
{
   // Создание формы
   std::auto_ptr<TMyPropertyForm*>
      MyPropertyForm(new TMyPropertyForm(0));
   // Связь со свойством
   MyPropertyForm->PropertyClass
      = reinterpret_cast<TPropertyClass*>(GetOrdValue());
   // Отображение формы, которая выполняет остальную работу.
   MyPropertyForm->ShowModal();
}
//---------------------------------------------------------//
```

Обратите внимание на использование ключевого слова `reinterpret_cast` для приведения числового представления указателя на класс (типа `int`) к типу действительного указателя на класс. Код в листинге 10.4 короче, чем в листинге 10.3 потому что в нем показаны только имеющиеся между ними различия.

Листинг 10.4. Код пользовательской формы для редактирования свойства типа `int`

```
// Сначала идет код формы TMyPropertyForm

// В ЗАГОЛОВОЧНОМ ФАЙЛЕ ИЗМЕНИМ ОПРЕДЕЛЕНИЕ
//---------------------------------------------------------//
class TMyPropertyForm : public TForm
{
__published: // Компоненты, управляемые IDE-средой
private:
   AnsiString& Value;
   int OldValue;
   // Другие объявления

public:
```

```
   __fastcall TMyPropertyForm(TComponent* Owner,
                              AnsiString& PropertyValue);
   // Другие объявления
};
//---------------------------------------------------------//

#endif
//---------------------------------------------------------//
// В ФАЙЛЕ РЕАЛИЗАЦИИ ИЗМЕНИМ КОНСТРУКТОР
__fastcall
   TMyPropertyForm::TMyPropertyForm(TComponent* Owner,
                                    AnsiString& PropertyValue)
                        : TForm(Owner), Value(PropertyValue)
{
   // Сохранение текущего значения свойства.
   // В этом случае оно имеет тип int,
   // поэтому нужно применить следующий код.
   OldValue = Value.ToInt();
}
//---------------------------------------------------------//

// МЕТОД Edit(), почти такой же...
#include "MyPropertyForm.h" // Запомним это файл
void __fastcall TCustomPropertyEditor::Edit(void)
{
   // Эта форма создается
   // с дополнительными параметрами!
   std::auto_ptr<TMyPropertyForm*>
      MyPropertyForm(new TMyPropertyForm(0, Value));
   // Отображение формы, которая выполняет остальную работу.
   MyPropertyForm->ShowModal();
}
//---------------------------------------------------------//
```

Различие между первым и вторым подходом заключается в том, что во втором подходе значение изменяется после возвращения модальной формы, а не сразу же при ее отображении на экране. Это наиболее распространенный способ использования формы для редактирования значения свойства. В листинге 10.5 показан основной код метода `Edit()`.

Листинг 10.5. Код пользовательской формы с обновлением значения после закрытия формы

```
#include "MyPropertyDialog.h" // Этот файл нужен для определения
                              // диалогового окна типа
                              // TMyPropertyDialog.
void __fastcall TCustomPropertyEditor::Edit(void)
{
   // Создание формы
   std::auto_ptr<TMyPropertyDialog*>
      MyPropertyDialog(new TMyPropertyDialog(0));
```

```
   // Указание текущих значений свойства в диалоговом окне
   // MyPropertyDialog->value1 = GetValue();
   // MyPropertyDialog->value2 = GetXxxValue();
   // и т.д. ...

   // Отображение формы с указанием результата.
   if(MyPropertyDialog->ShowModal() == IDOK)
   {
      // Установка новых значений свойства
   }
}
```

Обратите внимание, что класс `TMyPropertyDialog` может быть не диалоговым окном, а всего лишь его оболочкой, аналогичной компонентам стандартных диалоговых окон. В этом случае диалоговое окно будет отображаться с помощью метода `Execute()`. Более подробные сведения об этом диалоговом окне можно найти в разделе "Making a Dialog Box a Component" в интерактивной справке C++Builder. В этом случае оболочка диалогового окна должна быть наследницей класса `TObject`, а не класса `TComponent`.

Метод `GetValues()`

Метод `GetValues()` используется для наполнения значениями раскрывающегося списка свойства. Это делается с помощью последовательных вызовов функции `Proc()` и передачи строкового представления (`AnsiString`) значения. Например, если нужно использовать ряд значений, которые характеризуют интенсивность передачи данных между коммуникационным портом компьютера и внешним модемом, то при наличии в редакторе свойства атрибута `paValueList` код метода `GetValues()` может иметь следующий вид.

```
void __fastcall GetValues(Classes::TGetStrProc Proc)
{
   Proc("300");
   Proc("9600");
   Proc("57600");
   // и т.д. ...
}
```

Свойства класса `TPropertyEditor`

Класс `TPropertyEditor` имеет четыре свойства, которые могут использоваться при создании пользовательских редакторов свойств. Одно из них, `Value`, уже рассматривалось в двух предыдущих разделах. Остальные три свойства используются крайне редко. Вот их краткая характеристика.

- `Designer`. Это свойство используется только для чтения и возвращает указатель на интерфейс `IFormDesigner` IDE-среды. Оно используется для информирования IDE-среды о возникновении определенных событий или для указания IDE-среде на необходимость выполнения определенных действий. Например при создании собственного кода для методов `SetXxxValue()` нужно сообщить IDE-среде о модификации данного свойства. Это делается с помощью вызова функции `Designer->Modifed();`. На самом деле

вы вызываете метод `Modified()` класса `TPropertyEditor`, который вызывает точно такой же код. Метод `Revert()` класса `TPropertyEditor` также использует это свойство. Вряд ли вам потребуется вообще использовать это свойство. Здесь оно описывается лишь для полноты представления информации обо всех свойствах класса `TPropertyEditor`.

- `PrivateDirectory`. Это свойство содержит имя каталога в формате строки `AnsiString`, возвращаемой методом `GetPrivateDirectory()`, который получает имя этого каталога от метода `Designer->GetPrivateDirectory()`. Следовательно этот каталог задается IDE-средой. Если вашему редактору свойства нужен отдельный каталог для хранения дополнительных файлов, то его можно задать с помощью этого свойства, которое используется только для чтения.
- `PropCount`. Это свойство используется только для чтения и возвращает количество редактируемых свойств при выборе сразу нескольких компонентов. Оно используется только в том случае, когда метод `GetAttributes()` возвращает атрибут `paMultiSelect`.

Выбор оптимального редактора свойств

Рассмотрим свойство в компоненте, который является оболочкой для коммуникационного Windows API-интерфейса и позволяет задавать разные значения интенсивности обмена данными в баудах. Выбор значений допускается не только среди определенных значений. Пользователь может указать собственное значение интенсивности обмена данными в бодах. Как лучше всего организовать способ указания этих значений?

Можно использовать раскрывающийся список значений. При этом желательно, чтобы значения в списке имели числовой, а не перечислимый тип. Проще всего для этого было бы использовать пользовательский редактор свойств, который является наследником класса `TIntegerProperty`, но отображает раскрывающийся список задаваемых значений. Пользователь может ввести нужное значение в области редактирования значения свойства в окне Object Inspector. Этот способ достаточно просто реализовать, причем он работает вполне удовлетворительно.

Но можно ли назвать этот способ оптимальным? Он хорошо работает при выборе значения из списка, но при этом также следует учесть возможность ввода собственного значения пользователем. Это относительно просто, но для этого требуется, чтобы все значения в списке сравнивались со значением, возвращаемым этим свойством. Если оно отличается от списочных значений, то будет воспринято как пользовательское значение интенсивности обмена данными. В таком случае компонент должен послать введенное значение коммуникационному API-интерфейсу для проверки. Введенные значения могут оказаться очень разными по величине. Поэтому при каждом вводе пользовательского значения необходимо выполнить проверку его соответствия заданному диапазону значений. Хотя сам по себе редактор свойств очень прост, но код его сопровождения может оказаться очень большим. При создании рабочего кода может возникнуть не только эта, но и многие другие проблемы.

Переопределяя метод `SetValue()` и создавая два отдельных свойства: одно — для ввода пользовательского значения интенсивности обмена данными и другое, логического типа, — для указания используемого значения (пользовательского или хранящегося в списке), ограничимся только теми значениями, которые находятся в раскрывающемся списке

Похоже, нам придется создать огромное количество кода только для организации ввода простого целочисленного значения. Вернемся к самому началу и рассмотрим исходные требования еще раз.

Нам хотелось бы вводить значение из списка заданных значений, а также задавать пользовательское значение, которое может оказаться неприемлемым. Первоначально нам хотелось бы использовать для этих значений перечислимый тип, но использование целочисленных значений кажется более привлекательным. Рассмотрим внимательней способ с использованием перечислимого типа. В таком случае набор значений генерируется достаточно просто, причем мы даже можем указать порядок их расположения с помощью символов подчеркивания между инициалами перечисления и значением. Например, для переменной перечислимого типа `TBaudRate` с инициалами `br` значения интенсивности обмена 9600 и 115200 можно было бы представить в виде `br___9600` и `br_115200`.

В состав элементов перечислимого типа `enum` можно даже включить пользовательское значение `brUserDefined`. При выборе `brUserDefined` можно организовать чтение и проверку свойства `UserDefined` типа `int`. Так что это свойство тоже нам потребуется. Для этого совсем не нужно создавать пользовательский редактор свойства, так как уже предусмотрен класс `TEnumProperty`, который является редактором перечислимых свойств. Здесь, однако, возникает проблема. Каждый раз при указании или получении значения во время выполнения нужно использовать перечислимый тип, а это часто очень неудобно. Необходимо сделать перечисление доступным для пользователя компонента. Для поддержания "чистоты" глобального пространства имен перечислимый тип можно заключить в другое пространство имен, но это может еще больше усложнить применение значений перечислимого типа, что не рекомендуется. Действительно, при создании большинства компонентов стараются избежать этих дополнительных сложностей. Вот почему для значений перечислимого типа стремятся применять специальные буквенные обозначения. Более подробную информацию об именовании значений перечислимого типа можно найти в главе 3 в разделе о выборе типов имен.

Какой же способ является оптимальным? Это *целиком* зависит от назначения свойства и всего компонента в целом. Так как наши рассуждения имеют гипотетический характер, то в этой ситуации довольно сложно сказать, какой метод лучше. Создаваемые компоненты должны надежными и простыми. Следует избегать создания чересчур сложного кода, потому что он может скрыть некоторые тонкости работы компонента. Подход с использованием значений перечислимого типа может показаться немного запутанным при преобразованиях их к типу `int`, но в таком случае всем известно что с ними можно сделать и что нельзя. Избегая создания пользовательского редактора свойств можно существенно сэкономить время и другие ресурсы. Чтобы организовать чтение значения, можно создать свойство, используемое только для чтения, причем так, что, например, значение типа `int` для указания интенсивности обмена может быть сохранено при успешной установке значения свойства типа `enum`. Впоследствии оно может быть считано как значение типа `int` свойства, которое используется только для чтения.

При создании редакторов свойств и компонентов всегда следует тщательно рассматривать весь процесс целиком и возможные последствия принятых решений.

Свойства и исключительные ситуации

При изменении значения свойства всегда существует вероятность ввода неправильного значения и в этом случае функция-получатель значения должна иметь возможность обнаруживать некорректное значение и вызывать исключительную ситуацию для того, чтобы пользователь мог ввести правильное значение. Где именно происходит изменение значения свойства? В диалоговом окне редактирования свойства, в редакторе свойств, а также в самом свойстве во время выполнения. Их взаимосвязь схематически представлена на рис. 10.2. Обратите внимание, что параметром метода `SetValue()` является значение типа `const AnsiString`, даже если оно передается по значению (более подробно ключевое слово `const`

рассматривается в главе 3). Это позволяет предотвратить изменение значения свойства `Value` внутри метода `SetValue()`. Этот способ применения ключевого слова `const` отличается от общепринятого способа, а именно для указания неизменности передаваемого аргумента функции. При передаче аргумента по значению он копируется, а потому никак не может быть изменен. При возникновении ошибки единственным способом информирования пользователя является вызов исключительной ситуации. С помощью этого способа, т.е. передавая аргументы по значению с указанием ключевого слова `const`, можно также создать другие методы установки значения.

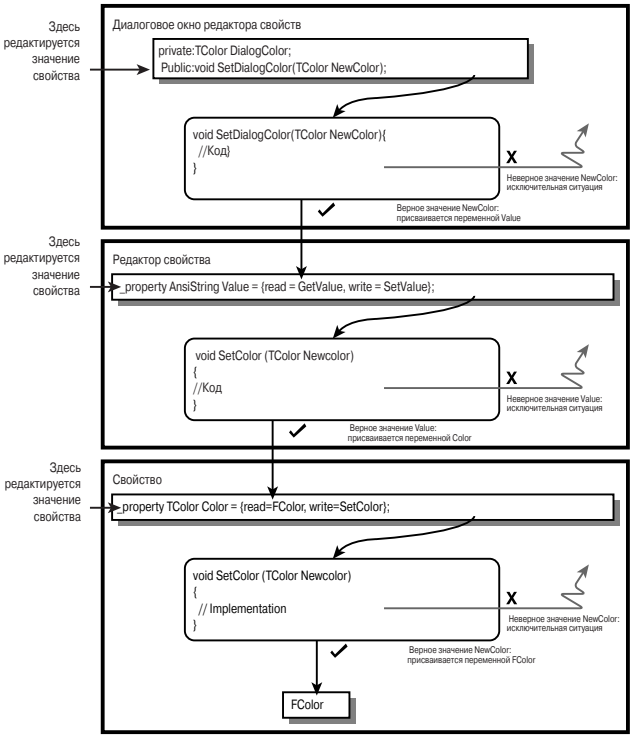


Рис. 10.2. Исключительные ситуации, возникающие при редактировании свойства

На рис. 10.2 показано, что для установки значения свойства вызывается метод `SetColor()` (если не возникнет исключительная ситуация). Именно здесь можно было бы проверить корректность значения и вызвать исключительную ситуацию. Напомним, что назначение исключительной ситуации состоит в обнаружении ошибки и предоставлении пользователю возможности ввести новое значение. При вызове обработчика исключительной ситуации из метода установки значения свойства операция редактирования значения пользователем, скорей всего, будет завершена. Это может быть связано с необходимостью повторного отображения диалогового окна и возникновением других неудобств. Вызов исключительной ситуации из редактора свойств (или диалогового окна редактора свойств) также не имеет смысла, потому что редактор свойств не используется при выполнении приложения, а потому не будет препятствовать появлению ошибочных значений.

Наиболее приемлемым вариантом является вызов исключительной ситуации в месте возникновения ошибки. Это не самое простое, но самое надежное решение.

Регистрация пользовательских редакторов свойств

Пользовательские редакторы свойств регистрируются почти элементарно. *"Почти"* потому, что для этого достаточно применить метод `RegisterPropertyEditor()`, но параметры этого метода не всегда просты. Как и другие функции регистрации, `RegisterPropertyEditor()` должна располагаться внутри функции `Register()` пакета. Ниже приведен пример объявления функции регистрации `RegisterPropertyEditor()`.

```
extern PACKAGE
void __fastcall RegisterPropertyEditor(
        Typinfo::PTypeInfo PropertyType,
        TMetaClass* ComponentClass,
        const AnsiString PropertyName,
        TMetaClass* EditorClass);
```

В табл. 10.7 приведен перечень параметров с описанием их назначения. Параметры `PropertyType` и `PropertyName` используются для указания критериев, которым должно удовлетворять свойство, чтобы его можно было использовать в данном редакторе свойств.

Таблица 10.7. Параметры функции регистрации редактора свойств `RegisterPropertyEditor()`

Имя параметра	Назначение
`PropertyType`	Этот параметр содержит указатель на структуру `TTypeInfo`, где хранится информация о типах свойства, для которого используется редактор. Этот параметр *обязательно* должен быть указан. Если это свойство имеет тип класса, производного от VCL-класса, то его следует получить с помощью макроса `__typeinfo(`*`TVCL-класс`*`)`. В противном случае нужно получить информацию о типе (`typeinfo`) аналогичного уже существующего свойства или за счет создания его вручную. Оба метода рассматриваются в этом разделе
`ComponentClass`	Этот параметр позволяет указать, будет ли редактор использоваться для всех совпадающих свойств во всех компонентах или только для совпадающих свойств в компонентах заданного типа. Для указания компонента некоторого типа следует использовать оператор `__classid` (который возвращает `TMetaClass*`) с указанием имени класса: `__classid(TComponentClassName)`. В противном случае все компоненты указываются передачей в качестве параметра значения 0
`PropertyName`	Этот параметр используется для указания имени свойства в формате `AnsiString`. Он используется для ограничения спецификации свойства. Если требуется использовать все свойства с совпадающей информацией о типах, то следует передать пустую строку `AnsiString ("")`. Если значение свойства `ComponentClass` равно 0, то этот параметр игнорируется
`EditorClass`	Этот параметр является обязательным. Он сообщает IDE-среде о том, какой редактор свойства нужно зарегистрировать. Как и параметр `ComponentClass`, он получает указатель класса `TMetaClass`. Следовательно, имя класса редактора свойства передается с помощью оператора `__classid(TPropertyEditorClassName)`

В табл. 10.7 показано, что параметрам ComponentClass и PropertyName можно присвоить значение, поэтому редактор свойства *не* ограничивается связью только с особым классом компонента или именем свойства. Это противоречит их обычному способу употребления. Единственный параметр, который следует прокомментировать, — PropertyType. Как уже говорилось ранее, макрос __typeinfo может использоваться для извлечения информации о типе, если тип свойства является классом библиотеки VCL (по крайней мере производным типом от TObject). Макрос __typeinfo имеет следующее определение в файле $(BCB)\Include\Vcl\Sysmac.h.

```
#define __typeinfo(type)
   (PTypeInfo)TObject::ClassInfo(__classid(type))
```

Если свойство не является классом библиотеки VCL, то информация о типе должна быть получена другим путем — либо на основании имени свойства и члена типа PTypeInfo, который принадлежит данному классу, либо сгенерирована вручную.

Тип PTypeInfo является указателем на структуру TTypeInfo и получается в результате следующего переопределения типа.

```
typedef TTypeInfo* PTypeInfo;
```

Структура TTypeInfo содержит следующее объявление в файле $(BCB)\Include\Vcl\Typinfo.hpp.

```
struct TTypeInfo
{
   TTypeKind Kind;
   System::ShortString Name;
};
```

Перечислимый тип TTypeKind объявлен в том же файле и содержит следующие типы.

```
enum TTypeKind { tkUnknown, tkInteger, tkChar,
                 tkEnumeration, tkFloat, tkString,
                 tkSet, tkClass, tkMethod,
                 tkWChar, tkLString, tkWString,
                 tkVariant, tkArray, tkRecord,
                 tkInterface, tkInt64, tkDynArray };
```

Переменная Name содержит строковое представление для названия соответствующего типа. Например, для типа int оно равно "int", а для типа AnsiString — "AnsiString". В следующих двух разделах рассматриваются способы получения указателя TTypeInfo* для типов свойств, которые не входят в состав библиотеки VCL.

Получение указателя TTypeInfo* на основе существующего свойства и класса для библиотечного типа

Для этого нужно иметь класс библиотеки VCL с уже определенным свойством, к которому предоставлен доступ. Тогда информацию о типе этого свойства можно получить с помощью функции GetPropInfo(), которая объявлена в файле $(BCB)\Include\Vcl\Typinfo.hpp.

Функция GetPropInfo() возвращает указатель на структуру TPropInfo для свойства данного класса с заданным именем свойства, а также (по желанию) с указанием типа TTypeKind. Тип PPropInfo имеет тип указателя на структуру TPropInfo и задается с помощью оператора typedef.

```
typedef TPropInfo* PPropInfo;
```

Функция `GetPropInfo()` имеет четыре перегруженные версии:

```
extern PACKAGE PPropInfo __fastcall
              GetPropInfo(PTypeInfo TypeInfo,
                          const AnsiString PropName);

extern PACKAGE PPropInfo __fastcall
              GetPropInfo(PTypeInfo TypeInfo,
                          const AnsiString PropName,
                          TTypeKinds AKinds);

extern PACKAGE PPropInfo __fastcall
              GetPropInfo(TMetaClass*AClass,
                          const AnsiString PropName,
                          TTypeKinds AKinds);

extern PACKAGE PPropInfo __fastcall
              GetPropInfo(System::TObject* Instance,
                          const AnsiString PropName,
                          TTypeKinds AKinds);
```

Эти перегруженные версии в конечном итоге вызывают первую версию перегруженного метода.

```
extern PACKAGE PPropInfo __fastcall
              GetPropInfo(PTypeInfo TypeInfo,
                          const AnsiString PropName);
```

Именно она представляет для нас наибольший интерес. В других версиях в качестве параметра можно указать перечислимый набор типов `TTypeKinds`. Он используется для указания одного (`TypeKind`) или нескольких (`TypeKinds`) типов, которым должно соответствовать свойство. На основании возвращенного указателя `PPropInfo` можно получить указатель `PTypeInfo` на структуру с информацией о типе свойства, которая хранится в поле `PropType` структуры `TPropInfo`. Объявление структуры `TPropInfo` содержится в файле `$(BCB)\Include\Vcl\Typinfo.hpp` и имеет следующий вид.

```
struct TPropInfo
{
   PTypeInfo* PropType;
   void* GetProc;
   void* SetProc;
   void* StoredProc;
   int Index;
   int Default;
   short NameIndex;
   System::ShortString Name;
};
```

Например, указатель `PTypeInfo` для свойства `Name` класса `TFont` можно получить с помощью указателя `PPropInfo`.

```
PPropInfo FontNamePropInfo =
          Typinfo::GetPropInfo(__typeinfo(TFont), "Name");
```

А затем с его помощью можно получить указатель PTypeInfo для заданного свойства.

```
PTypeInfo FontNameTypeInfo = *FontNamePropInfo->PropType;
```

Теперь это значение указателя PTypeInfo можно передать функции RegisterPropertyEditor(). Таким образом, мы получили указатель на структуру TTypeInfo для свойства типа AnsiString. Следовательно, указатель PTypeInfo можно получить и использовать как параметр типа PTypeInfo всякий раз, когда требуется получить информацию о типе AnsiString. Кроме того, указатель PTypeInfo для пользовательского свойства пользовательского компонента можно получить аналогичным образом.

```
PPropInfo CustomPropInfo =
          Typinfo::GetPropInfo(__typeinfo(TCustomComponent),
                               "CustomPropertyName");
PTypeInfo CustomTypeInfo = *CustomPropInfo->PropType;
```

Обратите внимание, что ситуация становится более понятной при использовании типов TTypeInfo* и TPropInfo* вместо их новых пользовательских определений (PTypeInfo и PPropInfo). Ключевое слово typedef использовано здесь для более простого сравнения с объявлениями функции GetPropInfo().

При этом представленные здесь промежуточные шаги получения указателя PTypeInfo могут игнорироваться. Например, приведенный ниже фрагмент можно использовать в качестве аргумента функции RegisterPropertyEditor() для пользовательского свойств пользовательского компонента.

```
*(Typinfo::GetPropInfo(__typeinfo(TCustomComponent),
                       "CustomPropertyName"))->PropType
```

Этот метод получения указателя TTypeInfo* основан на существовании опубликованного свойства заданного типа, который уже используется библиотекой VCL. Это не всегда возможно. Кроме того, иногда может показаться, что используемый тип совпадает с требуемым типом, но на самом деле это не так. Примером этого является свойство Interval компонента TTimer. Оно имеет тип Cardinal, который является переопределением типа unsigned int с помощью ключевого слова typedef в файле $(BCB)\Include\Vcl\Sysmac.h. Разумно было бы предположить, что, извлекая указатель TypeInfo* для этого свойства, можно регистрировать редакторы свойств для свойств типа unsigned int. Однако это неверно. Свойство с типом unsigned int должно быть реализовано в классе C++. В этом заключается следующий важный урок. Указатель TTypeInfo* для небиблиотечного класса не обязательно будет таким же, если это свойство относится к классу, реализованному на Object Pascal, а не языке C++. Для решения этой проблемы есть очень простой и эффективный способ, который заключается в создании класса, содержащего опубликованные свойства нужных типов. Затем для извлечения подходящего указателя TTypeInfo* следует использовать описанные выше методы, которые применяются для регистрации редактора свойств. Пример такого класса показан в листинге 10.6.

Листинг 10.6. Пример использования небиблиотечных типов свойств в одном классе

```
class PACKAGE TNonVCLTypesClass : public TObject
{
public:
__published:
```

```
   // Фундаментальные целочисленные типы
   __property int IntProperty = {};
   __property unsigned int UnsignedIntProperty = {};
   __property short int ShortIntProperty = {};
   __property unsigned short int UnsignedShortIntProperty = {};
   __property long int LongIntProperty = {};
   __property unsigned long int UnsignedLongIntProperty = {};
   __property char CharProperty = {};
   __property unsigned char UnsignedCharProperty = {};
   __property signed char SignedCharProperty = {};

   // Фундаментальные типы для чисел с плавающей запятой
   __property double DoubleProperty = {};
   __property long double LongDoubleProperty = {};
   __property float FloatProperty = {};

   // Фундаментальный логический тип
   __property bool BoolProperty = {};

   // Класс AnsiString
   __property AnsiString AnsiStringProperty = {};

private:
   // Закрытый конструктор, на основании которого нельзя
   // создать экземпляр класса
   inline __fastcall TNonVCLTypesClass() : TObject()
   {}
};
```

При создании компонента `TTestComponent` со свойством `Size` типа `unsigned int` для регистрации пользовательского редактора свойств следует использовать следующий код.

```
RegisterPropertyEditor(*(Typinfo::GetPropInfo
                                (__typeinfo(TNonVCLTypesClass),
                                 "UnsignedIntProperty")
                         )->PropType,
                        __classid(TTestComponent),
                        "Size",
                        __classid(TUnsignedProperty));
```

Ниже еще раз представлен первый параметр, который обычно вызывает наибольшие затруднения у разработчиков.

```
*(Typinfo::GetPropInfo(__typeinfo(TNonVCLTypesClass),
                       "UnsignedIntProperty"))->PropType
```

Он выглядит очень громоздким и сложным. Для его упрощения можно создать класс, содержащий статичные члены-функции, которые возвращают правильный указатель `TTypeInfo*` для каждого типа. Определение такого класса показано в листинге 10.7.

Листинг 10.7. Код из файла NonVCLTypeInfo.h

```
//---------------------------------------------------------------//
#ifndef NonVCLTypeInfoH
#define NonVCLTypeInfoH
//---------------------------------------------------------------//
#ifndef TypInfoHPP
#include <TypInfo.hpp>
#endif
//---------------------------------------------------------------//
class PACKAGE TNonVCLTypeInfo : public TObject
{
public:
   // Фундаментальные целочисленные типы
   static PTypeInfo __fastcall Int();
   static PTypeInfo __fastcall UnsignedInt();
   static PTypeInfo __fastcall ShortInt();
   static PTypeInfo __fastcall UnsignedShortInt();
   static PTypeInfo __fastcall LongInt();
   static PTypeInfo __fastcall UnsignedLongInt();
   static PTypeInfo __fastcall Char();
   static PTypeInfo __fastcall UnsignedChar();
   static PTypeInfo __fastcall SignedChar();

   // Фундаментальные типы для чисел с плавающей запятой
   static PTypeInfo __fastcall Double();
   static PTypeInfo __fastcall LongDouble();
   static PTypeInfo __fastcall Float();

   // Фундаментальный логический тип
   static PTypeInfo __fastcall Bool();

   // Класс AnsiString
   static PTypeInfo __fastcall AnsiString();

private:
   // Закрытый конструктор, на основании которого нельзя
   // создать экземпляр класса
   inline __fastcall TNonVCLTypeInfo():TObject()
   {}

};

// Далее следует определение класса TNonVCLTypesClass
// (см. листинг 10.6)

//---------------------------------------------------------------//
#endif
```

Код реализации этого класса показан в листинге 10.8.

Листинг 10.8. Код из файла NonVCLTypeInfo.cpp

```
#include <vcl.h>
#pragma hdrstop
#include "NonVCLTypeInfo.h"
//---------------------------------------------------------------//
#pragma package(smart_init)
//---------------------------------------------------------------//

PTypeInfo __fastcall TNonVCLTypeInfo::Int()
{
   return *(Typinfo::GetPropInfo(__typeinfo(TNonVCLTypesClass),
                                 "IntProperty"))->PropType;
}

PTypeInfo __fastcall TNonVCLTypeInfo::UnsignedInt()
{
   return *(Typinfo::GetPropInfo(__typeinfo(TNonVCLTypesClass),
                                 "UnsignedIntProperty"))->PropType;
}
//---------------------------------------------------------------//

PTypeInfo __fastcall TNonVCLTypeInfo::ShortInt()
{
   return *(Typinfo::GetPropInfo(__typeinfo(TNonVCLTypesClass),
                                 "ShortIntProperty"))->PropType;
}

PTypeInfo __fastcall TNonVCLTypeInfo::UnsignedShortInt()
{
   return *(Typinfo::GetPropInfo(__typeinfo(TNonVCLTypesClass),
                                 "UnsignedShortIntProperty")
                                )->PropType;
}
//---------------------------------------------------------------//

PTypeInfo __fastcall TNonVCLTypeInfo::LongInt()
{
   return *(Typinfo::GetPropInfo(__typeinfo(TNonVCLTypesClass),
                                 "LongIntProperty"))->PropType;
}

PTypeInfo __fastcall TNonVCLTypeInfo::UnsignedLongInt()
{
   return *(Typinfo::GetPropInfo(__typeinfo(TNonVCLTypesClass),
                                 "UnsignedLongIntProperty")
                                )->PropType;
}
//---------------------------------------------------------------//
```

```
PTypeInfo __fastcall TNonVCLTypeInfo::Char()
{
   return *(Typinfo::GetPropInfo(__typeinfo(TNonVCLTypesClass),
                                 "CharProperty"))->PropType;
}

PTypeInfo __fastcall TNonVCLTypeInfo::UnsignedChar()
{
   return *(Typinfo::GetPropInfo(__typeinfo(TNonVCLTypesClass),
                                 "UnsignedCharProperty")
                                )->PropType;
}

PTypeInfo __fastcall TNonVCLTypeInfo::SignedChar()
{
   return *(Typinfo::GetPropInfo(__typeinfo(TNonVCLTypesClass),
                                 "SignedCharProperty")
                                )->PropType;
}
//---------------------------------------------------------//

PTypeInfo __fastcall TNonVCLTypeInfo::Double()
{
   return *(Typinfo::GetPropInfo(__typeinfo(TNonVCLTypesClass),
                                 "DoubleProperty"))->PropType;
}

PTypeInfo __fastcall TNonVCLTypeInfo::LongDouble()
{
   return *(Typinfo::GetPropInfo(__typeinfo(TNonVCLTypesClass),
                                 "LongDoubleProperty")
                                )->PropType;
}

PTypeInfo __fastcall TNonVCLTypeInfo::Float()
{
   return *(Typinfo::GetPropInfo(__typeinfo(TNonVCLTypesClass),
                                 "FloatProperty"))->PropType;
}
//---------------------------------------------------------//

PTypeInfo __fastcall TNonVCLTypeInfo::Bool()
{
   return *(Typinfo::GetPropInfo(__typeinfo(TNonVCLTypesClass),
                                 "BoolProperty"))->PropType;
}
//---------------------------------------------------------//
```

```
PTypeInfo __fastcall TNonVCLTypeInfo::AnsiString()
{
   return *(Typinfo::GetPropInfo(__typeinfo(TNonVCLTypesClass),
                                  "AnsiStringProperty")
                                 )->PropType;
}
//---------------------------------------------------------//
```

В предыдущем примере регистрации редактора свойства `Size` типа `unsigned int` компонента `TTestComponent` функция регистрации будет выглядеть так, как показано ниже.

```
RegisterPropertyEditor(TNonVCLTypeInfo::UnsignedInt(),
                       __classid(TTestComponent),
                       "Size",
                       __classid(TUnsignedProperty));
```

Этот код гораздо проще и понятнее. Именно этот способ следует использовать для регистрации редакторов свойств небиблиотечных типов, т.е. типов, которые не являются наследниками классов библиотеки VCL.

Как уже говорилось выше, определение указателя `TTypeInfo*` для свойства небиблиотечного типа, реализованного на Object Pascal, отличается от определения указателя для свойства, реализованного на C++. Примером этого является свойство `PasswordChar` класса `TMaskEdit`. Для того чтобы зарегистрировать редактор свойства для всех символьных типов, следует проделать это на Object Pascal и на C++. Предыдущий подход на основе особого класса, содержащего соответствующие свойства небиблиотечного типа, прекрасно подходит для реализаций на C++. Однако для получения корректного указателя `TTypeInfo*` для реализаций на Object Pascal указатель `TTypeInfo*` должен быть определен непосредственно на основе класса библиотеки VCL, т.е. на основе свойства `PasswordChar` класса `TMaskEdit`. Именно этот способ был использован сначала для получения указателя `TTypeInfo*`. Для регистрации нового редактора `TCharPropertyEditor` для всех компонентов и свойств типа `char` нужно использовать следующий код.

```
TPropInfo*VCLCharPropInfo =
   Typinfo::GetPropInfo(__typeinfo(TMaskEdit),
                        "PasswordChar");
// Регистрация редактора свойств для компонентов библиотеки VCL,
// реализованных на Object Pascal

RegisterPropertyEditor(*VCLCharPropInfo->PropType,
                       0,
                       "",
                       __classid(TCharPropertyEditor));

// Регистрация редактора свойств для компонентов библиотеки VCL,
// реализованных на C++
RegisterPropertyEditor(TNonVCLTypeInfo::Char(),
                       0,
                       "",
                       __classid(TCharPropertyEditor));
```

Получение указателя TTypeInfo* вручную для небиблиотечного типа

Ручной способ получения указателя TTypeInfo* является альтернативным вариантом для способа на основе использования класса библиотеки VCL. Он представлен здесь лишь для сравнения и в силу его широкого распространения. Тем не менее автор рекомендует избегать его, используя вместо него первый способ. Указатель PTypeInfo должен быть создан перед вызовом функции RegisterPropertyEditor() либо код создания этого указателя должен располагаться в функции, которая возвращает указатель.

Существует два способа ручного создания такого кода. Один из них заключается в локальном объявлении статической структуры static TTypeInfo, присвоении соответствующих значений и использовании ссылки на нее с помощью указателя PTypeInfo в качестве аргумента. Другой способ основан на динамическом выделении памяти для структуры TTypeInfo, присвоении соответствующих значений и использовании ссылки на нее с помощью указателя PTypeInfo в качестве аргумента. Оба эти метода генерации соответствующего указателя PTypeInfo для свойства AnsiString показаны в листинге 10.9. Этот код и другие аналогичные функции находятся в модуле GetTypeInfo на прилагаемом к книге компакт-диске.

Листинг 10.9. Ручной способ создания указателя TTypeInfo*

```
//-------------------------------------------------------------//
//                      В виде функции                         //
//-------------------------------------------------------------//
TTypeInfo* AnsiStringTypeInfo(void)
{
   static TTypeInfo TypeInfo;
   TypeInfo.Name = "AnsiString";
   TypeInfo.Kind = tkLString;
   return &TypeInfo;
}

// или

TTypeInfo* AnsiStringTypeInfo(void)
{
   TTypeInfo*TypeInfo = new TTypeInfo;
   TypeInfo->Name = "AnsiString";
   TypeInfo->Kind = tkLString;
   return TypeInfo;
}

//------------------ Код регистрации -------------------------//
RegisterPropertyEditor(AnsiStringTypeInfo(),
                       0,
                       "",
                       __classid(TAnsiStringPropertyEditor));

//-------------------------------------------------------------//
//              Перед функцией RegisterPropertyEditor()      //
//-------------------------------------------------------------//
```

```
static TTypeInfo AnsiStringTypeInfo;
TypeInfo.Name = "AnsiString";
TypeInfo.Kind = tkLString;
RegisterPropertyEditor(&AnsiStringTypeInfo,
                       0,
                       "",
                       __classid(TAnsiStringPropertyEditor));

// или

TTypeInfo*AnsiStringTypeInfo = new TTypeInfo;
TypeInfo->Name = "AnsiString";
TypeInfo->Kind = tkLString;
RegisterPropertyEditor(AnsiStringTypeInfo,
                       0,
                       "",
                       __classid(TAnsiStringPropertyEditor));
```

Обратите внимание, что при динамическом создании структуры `TTypeInfo` (с помощью оператора `new`) она не удаляется после вызова функции `RegisterPropertyEditor()`, так как в противном случае регистрация может не состояться. Причина этого поясняется в следующем разделе.

Получение указателя `TTypeInfo*` для небиблиотечного типа

Какой из двух подходов следует использовать для получения указателя `TTypeInfo*` для небиблиотечного типа — определить его на основе класса библиотеки VCL или создать вручную? Ответ очень прост: если возможно, используйте первый способ. В частности, первый способ необходимо использовать в том случае, когда редактор свойства создается для переопределения существующего редактора свойства, который уже зарегистрирован библиотекой VCL (а не определен динамически), или для редактора свойств, который зарегистрирован предварительно с помощью первого способа. Вообще, первый подход является более надежным, потому что в нем используется представление библиотеки VCL для указателя `TTypeInfo*` заданного свойства. Тут следует указать на необходимость использования первого способа для переопределения зарегистрированного редактора свойств. Создание класса со статическими членами-функциями для возвращения подходящего указателя `TTypeInfo*` существенно упрощает первый способ, а потому его следует рассматривать как более предпочтительный.

Следует отметить важный факт: создание функции для особого указателя `PTypeInfo` (второй способ) — это *не* одно и то же, что и получение указателя `PTypeInfo` от библиотеки VCL (первый способ). Причина этого заключается в том, что реализация класса `TPropertyClassRec`, используемая внутри функции `RegisterPropertyEditor()`, содержит только переменную типа `PTypeInfo`, а не фактические значения полей `Name` и `Kind` структуры `TTypeInfo`. Поэтому не может использоваться ссылка на локально объявленную нестатичную структуру `TTypeInfo`, и не допускается удаление динамически созданной структуры `TTypeInfo` (она остается в свободной области памяти).

Регистрация редакторов свойств выполняется сравнительно просто. Однако при этом нужно позаботиться о точности передаваемых параметров. Часто бывает так, что редактор

свойств успешно компилируется и инсталлируется, но его нет в заданной функции, потому что в коде регистрации неправильно указан параметр PropertyName, хотя в целом редактор свойств работает без ошибок.

Правила переопределения редакторов свойств

Теперь, когда вы знаете, как регистрируются пользовательские редакторы свойств, и что существует возможность переопределения ранее инсталлированных редакторов свойств, возникает вопрос о правилах переопределения редакторов свойств. Переопределяя редакторы свойств, следует иметь в виду следующее.

- При наличии нескольких вариантов редактора свойств используется тот, который инсталлирован последним.
- Вновь зарегистрированный редактор свойств переопределяет существующий редактор свойств только в том случае, если использованная для регистрации спецификация *так же специальна*, как и спецификация, использованная для регистрации существующего редактора. Например, если редактор свойств зарегистрирован для свойства Shape (типа TShapeType) компонента TShape, то инсталляция нового редактора для свойств типа TShapeType *для всех* компонентов (ComponentClass = 0) *не* будет переопределением редактора свойства Shape компонента TShape.

При переопределении редактора свойств следует правильно выбрать метод получения информации о типе (PTypeInfo). Практический пример переопределения редактора свойств можно найти в пакете EnhancedEditors на прилагаемом к книге компакт-диске.

Использование изображений в редакторах свойств

В этом разделе описываются способы создания изображений в окне Object Inspector для пользовательских редакторов свойств. Некоторые редакторы свойств, обладающие возможностями рисования изображений в окне Object Inspector, уже были перечислены в табл. 2.4 в главе 2. Если редактор свойств наследуется от одного из редакторов свойств, перечисленных в табл. 2.4, или используется тип, зарегистрированный для одного из этих реакторов свойств, можно воспользоваться уже имеющимся кодом рисования изображений. Например, свойство типа TColor будет автоматически представлено в окне Object Inspector, как и другие свойства TColor. Однако существуют другие типы свойств, для которых полезно было бы использовать изображения. Для этого в базовом классе всех редакторов свойств TPropertyEditor предусмотрено шесть новых методов, пять из которых могут быть переопределены. Вот как выглядят их объявления.

```
DYNAMIC void __fastcall ListMeasureWidth(
                        const AnsiString Value,
                        Graphics::TCanvas* ACanvas,
                        int& AWidth);
DYNAMIC void __fastcall ListMeasureHeight(
                        const AnsiString Value,
                        Graphics::TCanvas* ACanvas,
                        int& AHeight);
```

```
DYNAMIC void __fastcall ListDrawValue(
                          const AnsiString Value,
                          Graphics::TCanvas* ACanvas,
                          const Windows::TRect& ARect,
                          bool ASelected);
DYNAMIC void __fastcall PropDrawValue(
                          Graphics::TCanvas* ACanvas,
                          const Windows::TRect& ARect,
                          bool ASelected);
DYNAMIC void __fastcall PropDrawName(
                          Graphics::TCanvas* ACanvas,
                          const Windows::TRect& ARect,
                          bool ASelected);
```

Шестой метод используется вместе с методами `XxxxDrawValue` и имеет следующее объявление.

```
AnsiString __fastcall GetVisualValue();
```

В табл. 10.8 кратко описано их назначение.

Таблица 10.8. Новые методы класса `TPropertyEditor`, предназначенные для рисования изображений

Метод	Назначение
`ListMeasureWidth()`	Используется для указания изменяемой ширины элемента раскрывающегося списка. Так как значение общей ширины раскрывающегося списка задается равным самому широкому элементу (или даже шире), то именно это значение и является минимальной шириной раскрывающегося списка
`ListMeasureHeight()`	Используется для указания изменяемой высоты элемента раскрывающегося списка. Если для элементов списка не используются большие изображения (как, например, для свойств `TCursor`), этот метод обычно нет необходимости переопределять
`ListDrawValue()`	Этот метод вызывается для рисования значения каждого свойства в раскрывающемся списке
`PropDrawValue()`	Этот метод вызывается для рисования выбранного значения свойства, если он не имеет фокуса ввода. Если свойство содержит фокус ввода, то текущее значение свойства будет представлено как редактируемое значение типа `AnsiString`
`PropDrawName()`	Этот метод вызывается для рисования имени свойства в окне Object Inspector. Он используется очень редко
`GetVisualValue()`	Этот метод используется для возвращения отображаемого значения свойства. Он используется вместе с методами `ListDrawValue()` и `PropDrawValue()` для рисования строкового представления `AnsiString` значения свойства

На рис. 10.3 показаны области в окне Object Inspector, в которых используются эти методы. Наиболее важными среди этих переопределяемых методов являются `ListMeasureWidth()`, `ListDrawValue()` и `PropDrawValue()`.

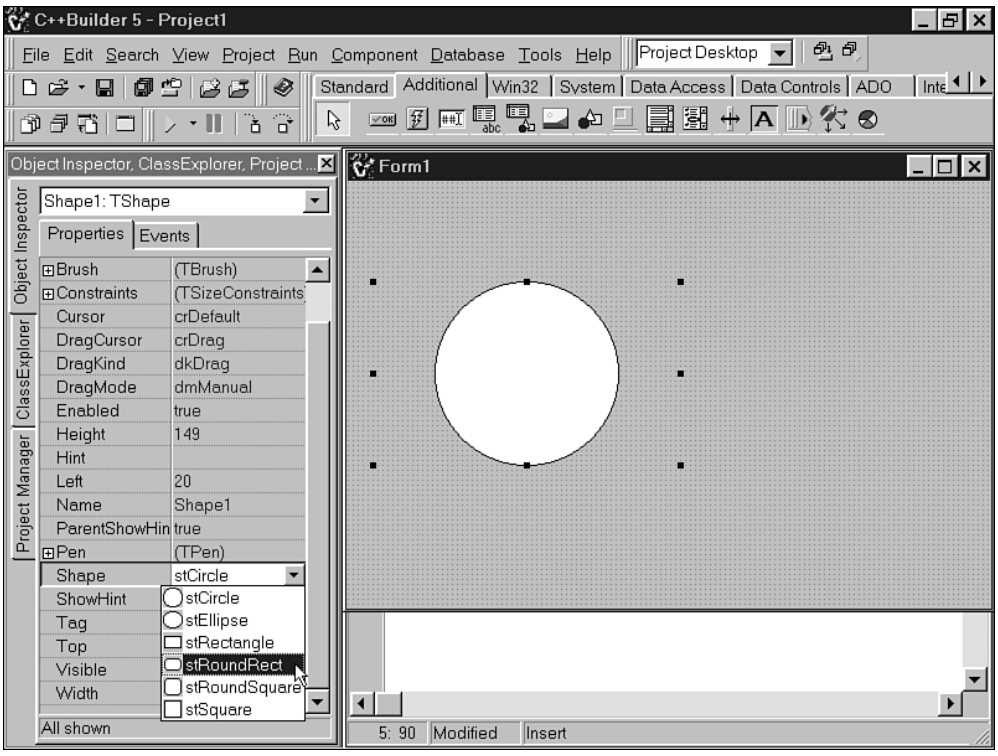


Рис. 10.3. Области окна Object Inspector*, которые связаны с новыми переопределяемыми методами класса* `TPropertyEditor`

Для создания собственных пользовательских изображений в окне Object Inspector необходимо создать новый класс редактора свойства, производный от класса `TPropertyEditor` или от производного от него класса. Выбор базового класса зависит от назначения редактируемого свойства. Например, редактор свойства типа `int` следует сделать наследником класса `TIntegerProperty`. Более подробная информация на эту тему приводится в разделе о создании пользовательских редакторов свойств выше в этой главе. Новый редактор свойств затем может быть определен в соответствии с форматом, указанным в листинге 10.10. В этом примере показан редактор свойства, производный от класса `TEnumProperty`.

Листинг 10.10. Код редактора свойства, который способен рисовать пользовательские изображения

```
#include "DsgnIntf.hpp"
class TCustomImagePropertyEditor : public TEnumProperty
{
   typedef TEnumProperty inherited;

public:
   DYNAMIC void __fastcall ListMeasureWidth(
                              const AnsiString Value,
```

```
                              Graphics::TCanvas* ACanvas,
                              int& AWidth);

   DYNAMIC void __fastcall ListMeasureHeight(
                              const AnsiString Value,
                              Graphics::TCanvas* ACanvas,
                              int& AHeight);

   DYNAMIC void __fastcall ListDrawValue(
                              const AnsiString Value,
                              Graphics::TCanvas* ACanvas,
                              const Windows::TRect& ARect,
                              bool ASelected);

   DYNAMIC void __fastcall PropDrawValue(
                              Graphics::TCanvas* ACanvas,
                              const Windows::TRect& ARect,
                              bool ASelected);

   DYNAMIC void __fastcall PropDrawName(
                              Graphics::TCanvas* ACanvas,
                              const Windows::TRect& ARect,
                              bool ASelected);

protected:
   #pragma option push -w-inl
   inline __fastcall virtual
      TCustomImagePropertyEditor(
         const _di_IFormDesigner ADesigner,
         int APropCount) : TEnumProperty(ADesigner,
                                         APropCount)
   {}
   #pragma option pop

public:
   #pragma option push -w-inl
   inline __fastcall virtual ~TCustomImagePropertyEditor(void)
   {}
   #pragma option pop

};
```

При этом предполагается, что будет изменена только та часть кода, которая отвечает в редакторе этого свойства за рисование. Остальная часть кода этого класса останется неизменной.

Код реализации пяти динамических (DYNAMIC) функций рассматривается в следующих разделах этой главы. В листинге каждой из них закомментирован рекомендуемый код, вслед за ним приводится фактический код, используемый для создания изображений, показанных на рис. 10.4, где представлен способ применения уже готового редактора свойств.

Рассмотрим редактор свойств для перечислимого типа `TShapeType`, созданного на основе компонента `TShape`. Класс для такого редактора свойства определяется так же, как в листинге 10.10. Однако в данном пример класс называется `TShapeTypePropertyEditor`. Для представления общей картины использования параметров пяти методов рисования они перечислены в табл. 10.9.

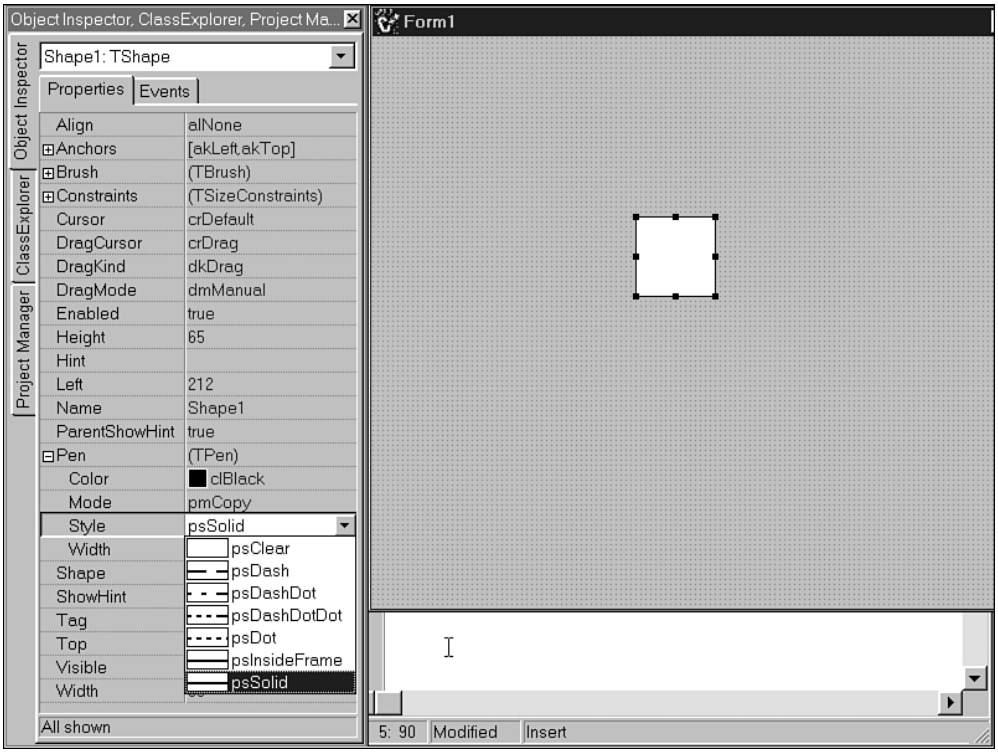


Рис. 10.4. Пример использования редактора свойства `TShapeTypePropertyEditor`

Таблица 10,9. Параметры пользовательских методов рисования изображений

Параметр	Назначение
`AWidth`	Текущая ширина в пикселях для строкового представления `AnsiString` этого значения в окне Object Inspector, включая передние и задние пробелы
`AHeight`	Принимаемая по умолчанию высота области рисования текущего элемента списка. Обычно она на 2 пикселя больше высоты `ACanvas->TextHeight("Ag")`. Например, для слова `Ag` (рис. 10.5) возвращается фактическая высота шрифта, т.е. высота надстрочной (у символа `A`) и подстрочной (у символа `g`) части. Добавление 2-х пикселей позволяет создать 1-пиксельную рамку. Напомним, что надстрочная высота также включает внутреннюю ведущую высоту (для указания акцентов, умлаутов и тильд в символах, используемых в некоторых языках) величиной 2 или 3 пикселя (рис. 10.5)
`ACanvas`	Инкапсулирует контекст устройства текущего элемента в окне Object Inspector
`ARect`	Представляет клиентскую область рисования
`Aselected`	Равен `true` для текущего выбранного элемента в окне Object Inspector

На рис. 10.5 показана схема, иллюстрирующая способ вычисления высоты текста.

На рис. 10.6 показана связь между параметрами в табл. 10.9 и фактическим способом рисования изображения и текста в окне Object Inspector. Этот рисунок будет постоянно использоваться в последующих обсуждениях, так как он содержит много различной информации.

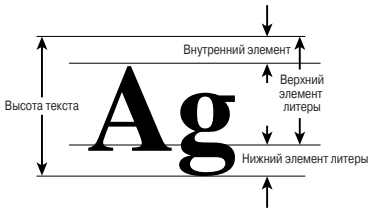


Рис. 10.5. Вычисление высоты текста

Метод ListMeasureWidth()

Первоначально значение свойства AWidth равно возвращаемому значению ACanvas->TextWidth(Value). Однако при создании изображения к величине AWidth следует прибавить ширину этого изображения. Метод ListMeasureWidth(), вызываемый во время вычисления ширины раскрывающегося списка, позволяет сделать это. Если область рисования имеет форму квадрата, значение AWidth легко обновить, прибавив к его текущему значению величину ACanvas->TextHeight("Ag")+2, которая является текущим значением высоты AHeight (см. табл. 10.9). (Как показано на рис. 10.6, значение ACanvas->TextHeight("Ag")+2 равно 18, т.е. 16 + 2 пикселей.) Напомним, что вместо Ag могут использоваться любые другие символы. При использовании рисунка большего размера к данной ширине можно прибавить необходимое постоянное значение. Если ширина рисунка известна заранее, то ее можно просто сложить с текущим значением AWidth. Соответствующий код показан в листинге 10.11.

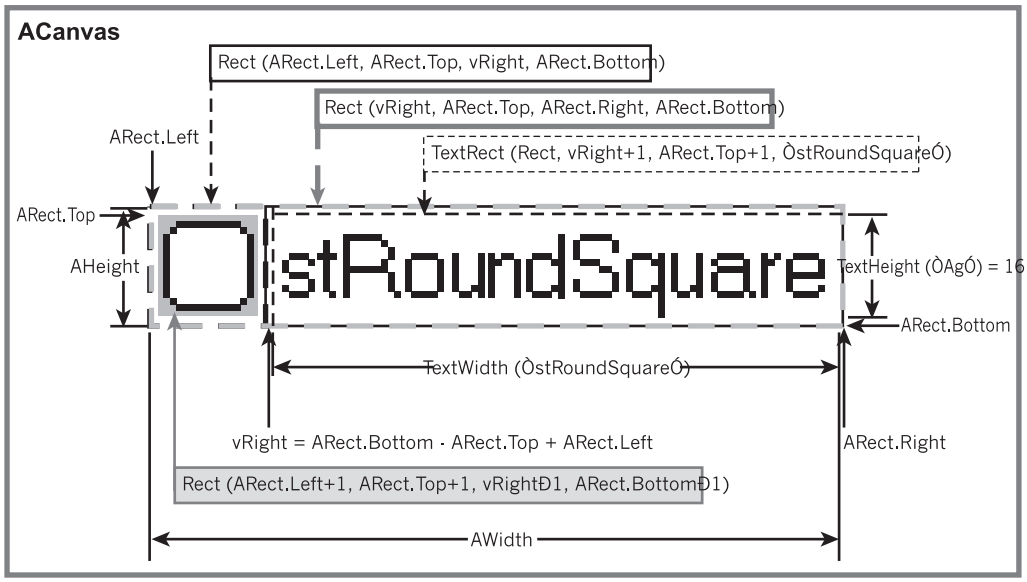


Рис. 10.6. Связь между параметрами рисования и фактическим видом изображения

Листинг 10.11. Переопределение метода ListMeasureWidth()

```
void __fastcall
       TShapeTypePropertyEditor::ListMeasureWidth(
          const AnsiString Value,
          Graphics::TCanvas* ACanvas,
```

```
          int& AWidth)
{
   AWidth +=
      (ACanvas->TextHeight("Ag")+2) + 0; // 0 может быть
                                         // заменен константой
}
```

Метод ListMeasureHeight()

Если нужный рисунок меньше принимаемой по умолчанию высоты, тот этот метод не придется переопределять. Однако при переопределении этого метода для высоты AHeight нельзя задавать значение, меньше ACanvas->TextHeight("Ag")+2, потому что отображаемый текст будет обрезан. Следовательно, в этом случае возможны два варианта. К текущему значению AHeight можно добавить некую константу для поддержания постоянного соотношения с шириной изображения либо непосредственно изменить значение AHeight. В последнем случае новое значение должно быть больше, чем ACanvas->TextHeight("Ag")+2. Пример переопределения этого метода показан в листинге 10.12.

Листинг 10.12. Переопределение метода ListMeasureHeight()

```
void __fastcall
   TShapeTypePropertyEditor::ListMeasureHeight(
      const AnsiString Value,
      Graphics::TCanvas* ACanvas,
      int& AHeight)
{
   AHeight += 0; // 0 можно заменить константой
}

// ИЛИ :

void __fastcall
   TShapeTypePropertyEditor::ListMeasureHeight(
      const AnsiString Value,
      Graphics::TCanvas* ACanvas,
      int& AHeight)
{
   if( (ACanvas->TextHeight("Ag")+2) < ImageHeight )
   {
      AHeight = ImageHeight;
   }
}
```

Метод ListDrawValue()

Именно этот метод отображает каждый элемент раскрывающегося списка в его канве. Для создания надежного и корректного кода этот метод должен иметь структуру, показанную в листинге 10.13. Для того чтобы получить представление о процессе отображения, обратитесь к рис. 10,6 и рис. 10.3.

Листинг 10.13. Шаблон переопределения метода `ListDrawValue()`

```
void __fastcall
   TCustomImagePropertyEditor::ListDrawValue(
      const AnsiString Value,
      Graphics::TCanvas* ACanvas,
      const Windows::TRect& ARect,
      bool ASelected)
{
   // Объявление переменной int vRight для указания правого
   // края изображения.
   // Префикс v указывает, что эта
   // переменная соответствует соглашению об именах,
   // которое используется в файле DsgnIntf.pas.

   try
   {
      // Этап 1 - Сохранить изменяемые свойства ACanvas.

      // Этап 2 - Окружить рамкой изменяемую область. Это нужно
      //          для изменения предыдущей перерисовки канвы.
      //          Например, при создании эффекта выделения в
      //          IDE-среде (значение свойства окружено
      //          желто-черным пунктиром и строка
      //          значения имеет цвет clNavy) изменяемые
      //          части канвы очищаются и готовятся к
      //          применению заданного пользователем способа
      //          перерисовки. Если при этом полностью
      //          вся канва ACanvas изменяется, то эта операция
      //          не потребуется.

      // Этап 3 - Выполнить подготовительные операции. Например,
      //          закрасить фон и разместить подсвеченную рамку
      //          вокруг изображения значения списка, если
      //          значение ASelected равно true.
      //
      //          Для выбора цвета, совпадающего с текущим
      //          цветом окна, выберите clWindowText. Его удобно
      //          использовать для границы изображения, поэтому
      //          он часто используется в качестве цвета пера
      //          ACanvas->Pen color.
      //
      //          Для создания чистого фона выберите для цветов
      //          ACanvas->Brush и/или ACanvas->Pen значение
      //          clWindow.
      //
      //          Для выбора такого же цвета, который используется
      //          в окне Object Inspector,
      //          выберите значение clBtnFace.

      // Этап 4 - Определение значения текущего элемента списка.
```

```
      // Этап 5 - Рисование нужного изображения в канве ACanvas.

      // Этап 6 - Восстановление измененных исходных значений
      //          свойств канвы ACanvas.
   }
   __finally
   {
      // Выполнить следующие действия для представления
      // в виде строки текущего элемента (т.е. Value) в канве
      // ACanvas.

      // 1. Вызвать родительский метод ListDrawValue с
      //    передачей значения vRight в качестве левого края
      //    переменной типа Rect, т.е.
      //
      //    TEnumProperty::ListDrawValue(Value,
      //                                 ACanvas,
      //                                 Rect(vRight,
      //                                      ARect.Top,
      //                                      ARect.Right,
      //                                      ARect.Bottom),
      //                                      ASelected);
      //
      //    который принимает следующий вид:
      //
      //    inherited::ListDrawValue(Value,
      //                             ACanvas,
      //                             Rect(vRight,
      //                                  ARect.Top,
      //                                  ARect.Right,
      //                                  ARect.Bottom),
      //                                  ASelected);
      //
      //    при использовании переопределения типа.

      // 2. Либо выполнить эту операцию непосредственно,
      //    вызвав член-функцию TextRect(), что позволяет
      //    избежать необходимости вызова родительской версии
      //    этой виртуальной функции (ListDrawValue()), т.е.
      //
      //    ACanvas->TextRect( Rect(vRight,
      //                            ARect.Top,
      //                            ARect.Right,
      //                            ARect.Bottom),
      //                       vRight+1,
      //                       ARect.Top+1,
      //                       Value );
   }
}
```

Фактический код на основе шаблона из листинга 10.13 показан в листинге 10.14. Этот код рисует каждый элемент раскрывающегося списка, который состоит из изображения с текстом, представляющего значение перечислимого типа, к которому относится данный элемент. На рис. 10.4 показано изображение в перерисованном раскрывающемся списке.

Листинг 10.14. Реализация метода `ListDrawValue()`

```
void __fastcall
   TShapeTypePropertyEditor::ListDrawValue(
      const AnsiString Value,
      Graphics::TCanvas* ACanvas,
      const Windows::TRect& ARect,
      bool ASelected)
{
   // Объявление vRight ('v' означает переменную (variable))
   int vRight = ARect.Bottom - ARect.Top + ARect.Left;

   try
   {
      // Этап 1 - Сохранение изменяемых свойств ACanvas

      TColor vOldPenColor = ACanvas->Pen->Color;
      TColor vOldBrushColor = ACanvas->Brush->Color;

      // Этап 2 - Создание рамки вокруг изменяемой области.

      ACanvas->Pen->Color = ACanvas->Brush->Color;
      ACanvas->Rectangle(ARect.Left, ARect.Top, vRight,
                         ARect.Bottom);

      // Этап 3 - Необходимая подготовка.

      if(ASelected)                      // Выбор цвета пера Pen
      {                                  // в зависимости от
         ACanvas->Pen->Color = clYellow; // выбранного значения
      }                                  // списка.
      else
      {
         ACanvas->Pen->Color = clBtnFace;
      }

      ACanvas->Brush->Color = clBtnFace; // Выбор цвета фона,
                                         // соответствующего
                                         // Object Inspector.

      ACanvas->Rectangle(ARect.Left + 1,    // Рисование фона
                         ARect.Top + 1,     // канвы Canvas на
                         vRight - 1,        // основе текущего
                         ARect.Bottom - 1); // пера Pen и
                                            // кисти Brush :-)

      // Этап 4 - Определение значения текущего элемента списка.
```

```
TShapeType ShapeType =
   TShapeType(GetEnumValue(GetPropType(), Value));

// Этап 5 - Рисование нужного изображения в канве ACanvas.
ACanvas->Pen->Color = clBlack;
ACanvas->Brush->Color = clWhite;

switch(ShapeType)
{
   case stRectangle : ACanvas->Rectangle(ARect.Left+2,
                                         ARect.Top+4,
                                         vRight-2,
                                         ARect.Bottom-4);
                      break;
   case stSquare    : ACanvas->Rectangle(ARect.Left+2,
                                         ARect.Top+2,
                                         vRight-2,
                                         ARect.Bottom-2);
                      break;
   case stRoundRect : ACanvas->RoundRect(ARect.Left+2,
                                         ARect.Top+4,
                                         vRight-2,
                                         ARect.Bottom-4,
                                         (ARect.Bottom-
                                         ARect.Top-6)/2,
                                         (ARect.Bottom-
                                         ARect.Top-6)/2);
                      break;
   case stRoundSquare : ACanvas->RoundRect(ARect.Left+2,
                                           ARect.Top+2,
                                           vRight-2,
                                           ARect.Bottom-2,
                                           (ARect.Bottom-
                                           ARect.Top)/3,
                                           (ARect.Bottom-
                                           ARect.Top)/3);
                        break;
   case stEllipse     : ACanvas->Ellipse(ARect.Left+1,
                                         ARect.Top+2,
                                         vRight-1,
                                         ARect.Bottom-2);
                        break;

   case stCircle      : ACanvas->Ellipse(ARect.Left+1,
                                         ARect.Top+1,
                                         vRight-1,
                                         ARect.Bottom-1);
                        break;
   default : break;
}
```

```
      // Этап 6 - Восстановление исходных значений свойств
      //          канвы ACanvas.
      ACanvas->Pen->Color = vOldPenColor;
      ACanvas->Brush->Color = vOldBrushColor;

   }
   __finally
   {
      // Рисование строки в канве ACanvas.
      // Используем метод 1, вызывая родительский метод

      inherited::ListDrawValue(Value,
                               ACanvas,
                               Rect(vRight,
                                    ARect.Top,
                                    ARect.Right,
                                    ARect.Bottom),
                                    ASelected);
   }
}
```

Этап 4 в листинге 10.14 имеет наиболее важное значение для метода `ListDrawValue()`. Именно здесь определяется значение текущего элемента раскрывающегося списка. Оно позволяет принять решение на этапе 5 о рисовании нужного изображения. Для перечислимого типа `TShapeType` строковое представление (`AnsiString`) значения должно быть преобразовано в фактическое значение с помощью следующего кода.

```
TShapeType ShapeType =
   TShapeType(GetEnumValue(GetPropType(), Value));
```

Метод `GetEnumValue()` объявлен в файле `$(BCB)\Include\Vcl\TypInfo.hpp` и возвращает значение `int`, которое используется для конструирования новой переменной `ShapeType` типа `TShapeType`. Функция `GetPropType()` возвращает указатель на структуру `TTypeInfo`, содержащую информацию о типе (`TypeInfo`) свойства (в этом примере `TShapeType`). Ее можно также получить с помощью следующего кода.

```
*GetPropInfo()->PropType
```

Этот подход аналогичен способу получения информации о типе при регистрации редакторов свойств (см. раздел о получении информации о типе с помощью существующего свойства и класса для небиблиотечных типов выше в этой главе) и может использоваться в более общем смысле. `Value` содержит строковое (`AnsiString`) представление для текущего перечислимого значения. Методы `GetPropType()` и `GetPropInfo()` являются членами-функциями класса `TPropertyEditor` и объявлены в файле `$(BCB)\Include\Vcl\DsgnIntf.hpp`. Описанные здесь способы создания редакторов свойств имеют очень большое значение, а потому их рекомендуется изучить особенно тщательно.

Все изображения перерисовываются в соответствии с параметрами `ARect`. Это значит, что вам не потребуется изменять код для увеличения или уменьшения перерисовываемых изображений. Для этого нужно просто изменить значения `AWidth` и `AHeight`. Замена константы `0` в методах `ListMeasureWidth()` и `ListMeasureHeight()` на значение `10` приведет к увеличению размера рисуемого изображения в раскрывающемся списке на 10 пикселей в каждом направлении. Обратите внимание, что изображение в области значения свойства при этом не будет изменено.

Метод PropDrawValue()

Этот метод отвечает за перерисовку текущего значения свойства в окне Object Inspector. Высота перерисовываемой области фиксирована (ARect.Bottom - ARect.Top), что означает меньшую гибкость при работе с перерисовываемыми изображениями по сравнению с изображениями, отображаемыми в раскрывающемся списке. Код этой операции аналогичен коду перерисовки этого же значения в раскрывающемся списке. Единственное отличие заключается в способе употребления параметра ARect. Таким образом, рисунок может быть отображен с помощью метода ListDrawValue(), аргументами которого являются параметры метода PropDrawValue(). Код этой функции-члена показан в листинге 10.15.

Листинг 10.15. Реализация метода PropDrawValue()

```
void __fastcall
   TShapeTypePropertyEditor::PropDrawValue(
      Graphics::TCanvas* ACanvas,
      const Windows::TRect& ARect,
      bool ASelected)
{
   if( GetVisualValue() != "" )
   {
      ListDrawValue(GetVisualValue(), ACanvas, ARect,
                    ASelected);
   }
   else
   {
      // Как и в случае метода ListDrawValue,
      // далее должен быть вызван либо родительский метод,
      // либо нужный код непосредственной перерисовки текста
      //
      // inherited::PropDrawValue(ACanvas, ARect, ASelected);
      //
      // или
      //
      // ACanvas->TextRect(ARect,
      //                   ARect.Left+1,
      //                   ARect.Top+1,
      //                   GetVisualValue());
      //
      // Здесь используем способ непосредственной
      // перерисовки текста, т.е.

      ACanvas->TextRect(ARect,
                        ARect.Left+1,
                        ARect.Top+1,
                        GetVisualValue());
   }
}
```

Метод PropDrawName()

Это последний и наиболее редко используемый переопределяемый метод из перечисленных выше. Он управляет отображением свойства Name (см. рис. 10.3). Как и в случае метода PropDrawValue(), высота рисуемой области остается фиксированной. Этот метод имеет ограниченное применение, хотя его можно использовать для добавления символов в свойстве с определенным поведением, например в используемом только для чтения свойстве About. Широкое использование этого метода не рекомендуется, так как пользователи могут запутаться.

Другой возможный способ его использования заключается в добавлении изображения для указания нужного компонента в свойствах, производных от класса TComponent. Этот способ употребления не применяется в данном случае для редактора свойств TShapeTypePropertyEditor, но необходимый для этого код показан в листинге 10.16.

Листинг 10.16. Код метода PropDrawName()

```
void __fastcall
   TCustomImagePropertyEditor::PropDrawValue(
      Graphics::TCanvas* ACanvas,
      const Windows::TRect& ARect,
      bool ASelected)
{
   if( GetName() != "" )
   {
      // Функция перерисовки нужного изображения, аналогичная
      // методу ListDrawValue(), т.е.
      //
      // PropDrawNameValue(GetName(),ACanvas,ARect,ASelected);
      // Она должна быть определена.

   }
   else
   {
      // Далее должен быть вызван либо родительский метод,
      // либо нужный код прямой перерисовки текста, т.е.
      //
      // inherited::PropDrawName(ACanvas, ARect, ASelected);
      //
      // или
      //
      // ACanvas->TextRect(ARect,
      //                   ARect.Left+1,
      //                   ARect.Top+1,
      //                   GetName());
      //
      // Здесь используем прямую перерисовку текста, т.е.

      ACanvas->TextRect(ARect,
                        ARect.Left+1,
                        ARect.Top+1,
                        GetVisualName());
   }
}
```

Редактор свойств TImageListPropertyEditor из пакета EnhancedEditors (см. табл. 10.1) использует этот метод для отображения пиктограммы компонента TImageList для свойств типа TCustomImageList*. В листинге 10.17 показан код этого метода. Обратите внимание, что ресурс ImageListPropertyImage загружается в конструкторе этого редактора свойств.

Листинг 10.17. Альтернативный вариант кода для метода PropDrawName()

```
void __fastcall
   TImageListPropertyEditor::PropDrawName(
      Graphics::TCanvas* ACanvas,
      const Windows::TRect& ARect,
      bool ASelected)
{
   TRect ValueRect = ARect;

   try
   {
      // Очистить канву, используя текущее перо и кисть
      ACanvas->FillRect(ARect);

      if( GetName() != "")
      {
         if(Screen->PixelsPerInch > 96) // Для крупного шрифта
         {
            ACanvas->Draw(ARect.Left + 1,
                          ARect.Top + 2,
                          ImageListPropertyImage );
         }
         else // Для мелкого шрифта
         {
            ACanvas->Draw(ARect.Left + 1,
            ARect.Top,
            ImageListPropertyImage );
         }

         ValueRect = Rect(ARect.Left + 16 + 2,
                          ARect.Top,
                          ARect.Right,
                          ARect.Bottom );
      }
   }
   __finally
   {
      // Независимо от исхода перерисовки изображения,
      // нужно отобразить текст
      inherited::PropDrawName(ACanvas, ValueRect, ASelected);
   }
}
```

Код в листинге 10.17 достаточно прост и не требует особых пояснений за исключением использования блока try/_finally, который нужен для отображения текста. Код внутри бло-

ка try аналогичен коду из листинга 10.16, за исключением того, что ресурс ImageListPropertyImage размещается по-разному, в зависимости от используемого размера шрифта. После отображения ресурса ImageListPropertyImage соответствующим образом смещается прямоугольная область Rect текста для предоставления места для этого ресурса.

Инсталляция пакетов только для редактирования

В предыдущем разделе был создан редактор свойств типа TShapeType, но при этом ничего не было сказано о его регистрации. Как показано выше в разделе о регистрации пользовательских редакторов свойств, для регистрации редактора в IDE-среде необходимо использовать функцию RegisterPropertyEditor(). Рассмотрим этот процесс еще раз. Заметим, что наиболее важны в данном случае *обязательный* вызов функции Register() и *обязательное* заключение функции Register() в пространство имен с тем же именем, что и имя файла, в котором она содержится. Причем первая буква названия пространства имен должна быть прописной, а остальные — строчными. Компонент не может регистрироваться автоматически с помощью программы-мастера ***Component Wizard***. Поэтому разработчику придется самостоятельно создать базовый код регистрации. Помня об этом, можно сэкономить время, потому что далеко не все ошибки кодирования могут быть обнаружены при компиляции. Базовый код регистрации пакетов только для редактирования показан в листинге 10.18.

Листинг 10.18. Код регистрации пакетов только для редактирования

```
#include <vcl.h>
#pragma hdrstop

#include "Имя_файла.h"    // Пустой файл, содержит только
                          // директивы include для исключения
                          // повторного указания заголовочных
                          // файлов.
#include "PropertyEditors.h" // Для редакторов свойств.
#include <TypInfo.hpp>       // Для TPropInfo* и GetPropInfo().
//---------------------------------------------------------//
#pragma package(smart_init)
//---------------------------------------------------------//
namespace Имя_файла // Первый символ - ПРОПИСНОЙ,
                    // остальные - строчные.
{
   void __fastcall PACKAGE Register()
   {
      // Использование следующего кода для регистрации
      // редактора TShapePropertyEditor для ВСЕХ
      // свойств типа TShapeType во ВСЕХ компонентах.

      TPropInfo* TShapeTypePropInfo = Typinfo::GetPropInfo(
         __typeinfo(TShape),
         "Shape");
      RegisterPropertyEditor(*TShapeTypePropInfo->PropType,
```

```
                        0,
                        "",
                        __classid(TShapeTypePropertyEditor));
    }
}
```

В листинге 10.18 упомянут пустой заголовочный файл *Имя_файла*.h, но включать его необязательно. Дело в том, что в исходном коде пакета файл с кодом регистрации (с расширением .cpp) включается с помощью макроса USEUNIT(), а потому заголовочный файл не является обязательным здесь.

Так как этот пакет предназначен только для регистрации редакторов (свойств или компонентов), то следует выбрать только параметр Designtime Only (Только для редактирования) в разделе Usage Options (Использование параметров) во вкладке Description (Описание) диалогового окна Options параметров пакета.

Наибольшим приоритетом обладают самые новые редакторы свойств и компонентов при условии, что их регистрация не менее специальна, чем в более старом редакторе. Это позволяет переопределять зарегистрированные в данный момент редакторы. Пример такого редактора можно найти в пакете EnhancedEditors (EnhancedEditors.bpk), который находится на прилагаемом к книге компакт-диске. Как видно в листинге 10.18, редактор TShapeTypePropertyEditor зарегистрирован для ВСЕХ свойств типа TShapeType ВСЕХ компонентов. Следовательно, свойство Shape класса TShape будет использовать именно этот редактор свойств. Благодаря этому возможна раздельная настройка и обновление редакторов свойств и компонентов. Как уже говорилось в разделе о работе с изображениями в раскрывающихся списках в окне Object Inspector в главе 2, редактор TFontNameProperty по умолчанию дезактивируется. Для его активации необходимо использовать специальные API-функции Open Tools, а это не так просто. Альтернативный подход заключается в переопределении редактора свойств TFontNameProperty для достижения желанного результата. Такой переопределенный редактор свойств TDisplayFontNameProperty также включен в пакет EnhancedEditors и позволяет читателю познакомиться с этим способом.

Использование связанных списков изображений в редакторах свойств

Чаще всего изображения в раскрывающихся списках редакторов свойств используются для отображения рисунков в свойстве-индексе изображений ImageIndex, связанном со свойством-списком TCustomImageList* данного компонента или его родительского компонента. В качестве общего примера рассмотрим связывание индекса изображений со свойством-списком TCustomImageList* в том же компоненте. Единственное отличие между связыванием со свойством-списком TCustomImageList* того же компонента и его родительского компонента заключается в способе извлечения копии указателя на список TCustomImageList. Более подробно эта тема рассматривается ниже в отдельном разделе о связывании со свойством-списком TCustomImageList родительского компонента и в разделе о свойствах-индексах ImageIndex.

Рассмотрим компонент TEnhancedImage, который содержит индекс изображений и список TCustomImageList*. Обратите внимание, что указатель списка TCustomImageList используется для создания классов, производных от данного класса, в противном случае использует-

ся указатель TImageList*. Назначение этого компонента — связывать компонент TEnhancedImage с компонентом TImageList, если последний располагается в той же форме. Изменяя индекс изображений компонента TEnhancedImage, можно добиться того, что отображаемое изображение будет соответствовать изображению, которое содержится в компоненте TImageList этого индекса. Индекс и список рекомендуется применять только в случае необходимости. В противном случае этот компонент будет вести себя, как обычный компонент TImage. Определение этого компонента приводится в листинге 10.19.

Листинг 10.19. Определение компонента TEnhancedImage

```
//---------------------------------------------------------//
#ifndef EnhancedImageH
#define EnhancedImageH
//---------------------------------------------------------//
#include <SysUtils.hpp>
#include <Controls.hpp>
#include <Classes.hpp>
#include <Forms.hpp>
//---------------------------------------------------------//

class PACKAGE TEnhancedImage : public TImage
{
   typedef TImage inherited;
private:
   Imglist::TCustomImageList* FImageList;
   int FImageIndex;
   bool FUseImageList;

protected:
   virtual void __fastcall SetUseImageList
                              (bool NewUseImageList);
   virtual void __fastcall SetImageIndex(int NewImageIndex);
   virtual void __fastcall SetImageList
      (Imglist::TCustomImageList* NewImageList);
   virtual void __fastcall UpdatePicture(void);

   // Переопределение функции оповещения компонента TComponent
      virtual void __fastcall Notification
                     (TComponent* AComponent,
                      TOperation Operation);
public:
      __fastcall TEnhancedImage(TComponent* Owner);

__published:
      __property bool UseImageList = {read=FUseImageList,
                                      write=SetUseImageList,
                                      default=false};
      __property Imglist::TCustomImageList* ImageList =
                                      {read=FImageList,
                                       write=SetImageList,
```

```
                                        default=0};
      __property int ImageIndex = {read=FImageIndex,
                                   write=SetImageIndex,
                                   default=-1};
};
//---------------------------------------------------------//
#endif
```

Чтобы расширить функциональные возможности компонента `TImage`, достаточно создать необходимые новые свойства в классе, производном от класса `TImage`, а затем создать необходимые функции получения и установления значений. Вот эти свойства: список изображений `ImageList` типа `TImageList`, индекс изображений `ImageIndex`, а также маркер `UseImageList` использования списка типа `TImageList`. Для каждого из этих свойств нужно создать метод установки его значения и организовать инициализацию в конструкторе класса. Кроме того, нужно переопределить метод `Notification()` класса `TComponent` и создать функцию обновления рисунка в случае необходимости при каждом изменении значения этих свойств. Код этих функций показан в листинге 10.20.

Листинг 10.20. Код функций компонента `TEnhancedImage`

```
//---------------------------------------------------------//
__fastcall TEnhancedImage::TEnhancedImage(TComponent* Owner)
         : TImage(Owner),
           FUseImageList(false),
           FImageIndex(-1),
           FImageList(0)
{
}
//---------------------------------------------------------//
void __fastcall TEnhancedImage::SetUseImageList
                                   (bool NewUseImageList)
{
   if(NewUseImageList != UseImageList)
   {
      FUseImageList = NewUseImageList;
      UpdatePicture();
   }
//---------------------------------------------------------//
void __fastcall TEnhancedImage::SetImageIndex(int NewImageIndex)
{
   if(NewImageIndex != FImageIndex)
   {
      FImageIndex = NewImageIndex;
      UpdatePicture();
   }
}
//---------------------------------------------------------//
void __fastcall TEnhancedImage::SetImageList
                  (Imglist::TCustomImageList* NewImageList)
{
```

```
   if(NewImageList != FImageList)
   {
      FImageList = NewImageList;
      if(ImageList == 0)ImageIndex = -1;
   }
}
//---------------------------------------------------------//
void __fastcall TEnhancedImage::UpdatePicture(void)
{
   if( UseImageList
       && ImageList != 0
       && ImageIndex >= 0
       && ImageIndex <ImageList->Count )
   {
      std::auto_ptr<Graphics::TBitmap>
         Image(new Graphics::TBitmap());

      ImageList->GetBitmap(ImageIndex, Image.get());
      Picture->Assign(Image.get());
   }
}
//---------------------------------------------------------//
void __fastcall TEnhancedImage::Notification
                                  (TComponent* AComponent,
                                   TOperation Operation)
{
   inherited::Notification(AComponent, Operation);

   if(Operation == opRemove)
   {
      if(AComponent == ImageList) ImageList = 0;
      // В случае необходимости можно организовать проверку
      // значений других свойств
   }
}
//---------------------------------------------------------//
```

Код конструктора имеет обычную структуру: члены-данные для свойств класса инициализируются в списке инициализации. Рассмотрим теперь более внимательно свойства `SetUseImageList()` и `SetImageIndex()`, которые имеют одинаковый принцип работы. Если вновь заданное значение не равно текущему, то текущее значение заменяется новым, после чего вызывается метод `UpdatePicture()`. С учетом текущих значений свойств он подстраивает значение свойства `Picture` (унаследованное от свойства `TImage`). Аналогично работает метод `SetImageList()`: если вновь заданное значение не равно текущему, то текущее значение заменяется новым. Но если новое значение равно `0` (т.е., `NULL`, что означает отсутствие списка `TImageList`), то для свойства `ImageIndex` задается значение `-1`, т.е. принимаемое по умолчанию значение этого свойства. Обратите внимание: после присвоения значения свойству `FImageList`, в следующем выражении `if` вместо него используется свойство `ImageList`, которое возвращает значение свойства `FImageList`, что в данном случае означает одно и то же. Но

если внутреннее представление свойства имеет более сложное устройство (т.е., для данного свойства нужно использовать функцию-член получения значения), то лучше считывать значение свойства, а не член-данные. Это улучшает восприятие кода.

Точно так же для гарантированного вызова функции установки значений ImageIndex обновляется индекс ImageIndex, а не FImageIndex. При создании пользовательских компонентов следует уделить особое внимание этим тонкостям.

Основную часть работы выполняет функция UpdatePicture(). Она проверяет существование связанного с компонентом списка TImageList, необходимость его использования (UseImageList = true), а также корректность значения индекса ImageIndex для списка TImageList. Иначе говоря, оно может иметь нулевое значение либо быть больше или меньше, чем количество изображений в списке. Если проверка завершена успешно, изображение извлекается из списка и передается в объект Graphics::TBitmap (переменная Image), где присваивается свойству Picture. Обратите внимание, что объект Graphics::TBitmap объявляется с помощью функции std::auto_ptr. Это гарантирует удаление объекта даже при возникновении исключительной ситуации. Наиболее оптимальным вариантом является использование блока try/_finally. В таком случае код внутри выражения if в функции UpdatePicture() будет выглядеть так, как показано ниже.

```
Graphics::TBitmap* Image = new Graphics::TBitmap();

try
{
   ImageList->GetBitmap(ImageIndex, Image);
   Picture->Assign(Image);
}
__finally
{
   delete Image;
}
```

Наконец, функция Notification() переопределяется таким образом, что если связанный список TImageList удаляется, то компонент может задать нулевое значение для указателя на список ImageList. Обратите внимание, что сначала вызывается унаследованный код для гарантированной обработки всех уведомлений. Здесь для автоматического вызова функции установки значений списка ImageList, вместо списка FImageList, используется список ImageList. Это очень важно для корректной работы компонента, так как в противном случае могут нарушаться права доступа.

Итак, мы получили компонент, для которого нужно создать редактор свойства с индексом изображений ImageIndex типа int. Обратите внимание: вместо этого типа следует использовать тип TImageIndex, который является простым переобозначением типа int в файле $(BCB)\Include\Vcl\Imglist.hpp. Этот редактор свойства также является производным от класса TIntegerProperty. В листинге 10.21 определяется редактор TImageIndexPropertyEditor для свойства ImageIndex.

Листинг 10.21. Определение редактора свойства TImageIndexPropertyEditor

```
#include "DsgnIntf.hpp"
class TImageIndexPropertyEditor : public TIntegerProperty
{
   typedef TIntegerProperty inherited;
```

```
private:
   static const int Border = 2;          // Граница вокруг
                                         // рисунка в пикселях
   static const int MaxImageWidth = 64;  // Максимальная ширина
   static const int MaxImageHeight = 64; // Максимальная высота

protected:protected:
   virtual Imglist::TCustomImageList*
      __fastcall GetComponentImageList(void);

public:
   virtual TPropertyAttributes __fastcall GetAttributes(void);
   virtual void __fastcall GetValues(Classes::TGetStrProc Proc);

   DYNAMIC void __fastcall
      ListMeasureWidth(const AnsiString Value,
                       Graphics::TCanvas* ACanvas,
                       int& AWidth);

   DYNAMIC void __fastcall
      ListMeasureHeight(const AnsiString Value,
                        Graphics::TCanvas* ACanvas,
                        int& AHeight);

   DYNAMIC void __fastcall
      ListDrawValue(const AnsiString Value,
                    Graphics::TCanvas* ACanvas,
                    const Windows::TRect& ARect,
                    bool ASelected);

   // Это используемое только для чтения свойство будет
   // применяться как указатель на список изображений компонента
   __property Imglist::TCustomImageList* ComponentImageList =
                                         {read=GetImageList};
protected:
   #pragma option push -w-inl
   inline __fastcall virtual TImageIndexPropertyEditor
      (const _di_IFormDesigner ADesigner,
      int APropCount) : TIntegerProperty(ADesigner,
                                         APropCount)
   {}
   #pragma option pop

public:
   #pragma option push -w-inl
   inline __fastcall virtual ~TImageIndexPropertyEditor(void)
   {}
   #pragma option pop

};
```

Код в листинге 10.21 аналогичен коду приведенных выше редакторов свойств. Однако в нем добавлено используемое только для чтения свойство Imglist::TCustomImageList* вместе с соответствующей функцией извлечения значения.

```
virtual Imglist::TCustomImageList* __fastcall
   GetComponentImageList(void);
```

Эта функция позволяет извлечь значение свойства TCustomImageList*, организовав доступ к отдельным изображениям списка. Как и в предыдущем разделе, для подробного обсуждения этого кода следует рассмотреть методы класса TImageIndexPropertyEditor.

Метод GetAttributes()

Он используется для определения атрибутов, отображаемых редактором свойств в окне Object Inspector. В этом случае желательно создать раскрывающийся список значений (paValueList), причем так, чтобы значение свойства нельзя было редактировать при выборе сразу нескольких компонентов (paMultiSelect). Вот как выглядит соответствующий код.

```
TPropertyAttributes __fastcall
   TImageIndexPropertyEditor::GetAttributes(void)
{
   return (inherited::GetAttributes() << paValueList >>
           paMultiSelect);
}
```

Метод GetComponentImageList()

Этот метод, организуя связь между списком изображений TCustomImageList* компонента и свойством с индексом изображений, имеет ключевое значение для успешной работы редактора свойств. Связь основана на извлечении указателя того компонента, к которому относится редактируемое свойство. Затем указатель используется для возвращения соответствующего списка изображений TCustomImageList*. Если TCustomImageList* и свойство с индексом изображений находятся в одном компоненте, то для этого придется использовать следующий код.

```
Imglist::TCustomImageList* __fastcall
   TImageIndexPropertyEditor::GetComponentImageList(void)
{
   TEnhancedImage* Component =
      dynamic_cast<TEnhancedImage*>(GetComponent(0));

   if(Component)
   {
      return Component->ImageList;
   }
   else return 0;
}
```

Здесь используется метод GetComponent() редактора свойства со следующим объявлением.

```
Classes::TPersistent* __fastcall GetComponent(int Index);
```

Он возвращает указатель типа TPersistent* на редактируемый компонент Index. Так как этот редактор не имеет атрибута paMultiSelect, то в каждый момент времени он может редактировать только один компонент. Таким образом, может быть возвращен только указатель на этот единственный компонент (Index == 0). Затем его тип динамически (dynamic_cast) приводится к типу редактируемого компонента. Получив корректный указатель на нужный компонент, достаточно просто вернуть указатель на свойство TCustomImageList. В случае сбоя динамического приведения типов (dynamic_cast) будет возвращено значение 0.

Код для свойства-индекса изображений, родительский компонент которого содержит список TCustomImageList, имеет более сложную структуру. Эта тема более подробно рассматривается в разделе об организации связи со списком TCustomImageList родительского компонента.

Метод GetValues()

Этот метод используется для наполнения соответствующими значениями раскрывающегося списка свойства. В данном случае — это индексы изображений списка. Вот как выглядит код этого метода.

```
void __fastcall TImageIndexPropertyEditor::GetValues
   (Classes::TGetStrProc Proc)
{
   TCustomImageList* ImageList = ComponentImageList;

   if(ImageList != 0)
   {
      for(int i = 0; i<ImageList->Count; ++i)
         Proc(IntToStr(i));
   }
}
```

Функция возвращает в виде строкового значения (AnsiString) текущее значение индекса изображения. Обратите внимание, что значение свойства ComponentImageList используется не непосредственно, а присваивается локальной переменной TCustomImageList* ImageList. Дело в том, что метод GetComponentImageList() вызывается каждый раз при чтении значения ImageList. А при прямом использовании свойства ComponentImageList метод GetComponentImageList() вызывался бы чрезмерно часто. В этом случае разумнее однократно считать значение свойства ComponentImageList и создать его копию. Такой способ применяется во всех методах, где используется указатель списка ComponentImageList.

Методы ListMeasureWidth() и ListMeasureHeight()

Эти методы управляют шириной и высотой изображений компонента TImageList. Так как размер изображений может быть очень большим, то следует указать верхний предел для высоты и ширины отображаемого рисунка. Чаще всего (и именно так происходит в случае библиотеки VCL) для этого используется квадрат 64×64 пикселя. Если изображение имеет бо́льшие размеры, то на экране оно будет отображено только с заданными максимальными значениями высоты и ширины. Метод StretchDraw() класса TCanvas может использоваться для оптимального представления большого рисунка.

Выбор значения 64 в качестве максимальной ширины и высоты изображения в раскрывающемся списке имеет несколько причин. Во-первых, разрешение экрана ограничено, а в списке может находиться сразу несколько изображений. При наличии пяти изображений для отображе-

ния всего списка потребуется 5×64 = 320 пикселей по вертикали. Даже при очень большом разрешении такой список будет очень большим. Размер 64 не очень велик и подходит для просмотра мелких деталей. Кроме того, это число является степенью числа 2, что очень нравится всем программистам. В следующем коде максимально допустимые значения ширины и высоты заменены статичными и постоянными (`static const`) членами-данными `MaxImageWidth` и `MaxImageHeight`, соответственно. Кроме того, статичный и постоянный (`static const`) член `Border` используется для представления заданной ширины границы вокруг изображения и имеет значение 2. Вот как выглядит код метода `ListMeasureWidth()`.

```
void __fastcall
   TImageIndexPropertyEditor::ListMeasureWidth
      (const AnsiString Value,
       Graphics::TCanvas* ACanvas,
       int& AWidth)
{
   TCustomImageList* ImageList = ComponentImageList;

   if(ImageList != 0)
   {
      if(ImageList->Width < MaxImageWidth)
      {
         AWidth += ImageList->Width + Border*2;
      }
      else AWidth += MaxImageWidth + Border*2;
   }
}
```

Обратите внимание, что отступ величиной 4 пикселя (`Border*2`) предназначен для создания некоторого промежутка между отображаемыми рисунками и текстом, представляющим значение индекса изображения. Код метода `ListMeasureHeight()` имеет аналогичную структуру.

```
void __fastcall
   TImageIndexPropertyEditor::ListMeasureHeight
      (const AnsiString Value,
       Graphics::TCanvas* ACanvas,
       int& AHeight)
{
   TCustomImageList* ImageList = ComponentImageList;
   if(ImageList != 0)
   {
      if(ImageList->Height < MaxImageHeight &&
         ImageList->Height > AHeight)
      {
         AHeight = ImageList->Height + Border*2;
      }
      else
         if(ImageList->Height > AHeight)
            AHeight = MaxImageHeight + Border*2;
   }
}
```

Код, управляющий высотой изображения, немного сложнее, поскольку необходимо учитывать возможность того, что данное изображение имеет меньшую высоту, чем высота текста.

Метод ListDrawValue()

Это наиболее сложный из всех рассматриваемых методов. Он используется для отображения рисунков из компонента TImageList, на который указывает указатель TCustomImageList*. Код этого метода показан в листинге 10.22.

Листинг 10.22. Код метода ListDrawValue()

```
void __fastcall
   TImageIndexPropertyEditor::ListDrawValue
      (const AnsiString Value,
       Graphics::TCanvas* ACanvas,
       const Windows::TRect& ARect,
       bool ASelected)
{
   TRect ValueRect = ARect;

   try
   {
      TCustomImageList* ImageList = ComponentImageList;

      // Очистка канвы с помощью текущего пера и кисти
      ACanvas->FillRect(ARect);

      if(ImageList != 0 && Value != "")
      {
         int ImageWidth = ImageList->Width;
         int ImageHeight = ImageList->Height;

         if(ImageWidth > MaxImageWidth ||
            ImageHeight > MaxImageHeight)
         {
            std::auto_ptr<Graphics::TBitmap>
               Image(new Graphics::TBitmap());

            // Указание ширины и высоты изображения
            Image->Width = ImageWidth;
            Image->Height = ImageHeight;

            // Отображение рисунка из списка ImageList в канве
            ImageList->Draw(Image->Canvas, 0, 0,
                            StrToInt(Value), true);

            if(ImageWidth >MaxImageWidth)
               ImageWidth = MaxImageWidth;
            if(ImageHeight >MaxImageHeight)
               ImageHeight = MaxImageHeight;

            // Определение области рисования
```

```
            TRect ImageRect = Rect(ARect.Left + Border,
                                   ARect.Top + Border,
                                   ARect.Left + Border +
                                                ImageWidth,
                                   ARect.Top + Border +
                                               ImageHeight);

            // Отображение рисунка в канве с помощью
            // функции StretchDraw
            ACanvas->StretchDraw(ImageRect, Image.get());
         }
         else
         {
            // Отображение рисунка в канве непосредственно из
            // списка ImageList с границей шириной 2 пикселя.
            ImageList->Draw(ACanvas,
                            ARect.Left + Border,
                            ARect.Top + Border,
                            StrToInt(Value),
                            true);
         }

         ValueRect = Rect(ARect.Left +
                          ImageWidth + Border*2,
                          ARect.Top,
                          ARect.Right,
                          ARect.Bottom);
      }
   }
   __finally
   {
      // Независимо от исхода рисования нужно
      // отобразить текст.
      inherited::ListDrawValue(Value, ACanvas, ValueRect,
                               ASelected);
   }
}
```

Код в листинге 10.22 предназначен для определения размера рисуемого изображения и принятия решения о способе рисования: с помощью метода `ACanvas->StretchDraw()` либо непосредственным отображением его из списка изображений. Если ширина или высота изображения больше максимальных значений `MaxImageWidth` или `MaxImageHeight`, то изображение сначала копируется в объект `Graphics::Tbitmap`, а затем подгоняется с помощью функции `StretchDraw`. В противном случае, рисунок отображается в канве списка как есть.

Обратите внимание на использование блока `try/__finally` для вызова унаследованного метода `ListDrawValue()` и гарантированного отображения текста, независимо от исхода операции отображения рисунка. Это верно всегда, за исключением случая, когда исключительная ситуация возникнет в первой строке кода метода за пределами блока `try`. Это сделано преднамеренно, поскольку возникновение исключительной ситуации в этой строке означало бы, что значение `ValueRect` некорректно, а потому код в блоке `__finally` выполнять не следует.

Помимо упомянутых элементов эта функция в целом аналогична методу ListDrawValue(), описанному прежде в разделе об использовании рисунков в редакторах свойств, так как канва должна быть очищена для удаления результатов предыдущих операций рисования. Метод ListDrawValue() не содержит ошибок указания размеров, которые имеются в аналогичных методах ListDrawValue() в библиотеке VCL, и способен должным образом подогнать размеры изображения. Методы ListDrawValue() в библиотеке VCL не могут выполнить эту операцию (по крайней мере во время написания этой книги).

Метод PropDrawValue()

Этот метод переопределен для отображения небольшой пиктограммной версии изображения. Полезность этого действия для данного класса весьма сомнительна, но метод все же представлен здесь для полноты картины. При необходимости его можно удалить. Код этого метода идентичен коду метода ListDrawValue(), за исключением того, что значения MaxImageWidth и MaxImageHeight заменяются переменной CurrentMaxSide, которая вычисляется на основании значения высоты области рисования ARect и с учетом 1-пиксельной рамки вокруг нее. Кроме того, вместо метода ListDrawValue() вызывается унаследованный метод PropDrawValue(). Код этого метода показан в листинге 10.23.

Листинг 10.23. Код метода PropDrawValue()

```
void __fastcall
   TImageIndexPropertyEditor::PropDrawValue
      (Graphics::TCanvas* ACanvas,
       const Windows::TRect& ARect,
       bool ASelected)
{
   TRect ValueRect = ARect;

   try
   {
      TCustomImageList* ImageList = ComponentImageList;

      // Очистка канвы с помощью текущего пера и кисти
      ACanvas->FillRect(ARect);

      if(ImageList != 0 && GetVisualValue() != "")
      {
         int ImageWidth = ImageList->Width;
         int ImageHeight = ImageList->Height;

         // Вычисление максимальной ширины MaxSide
         // квадратной области рисования.
         int CurrentMaxSide = ARect.Bottom - ARect.Top - 2;
         if(ImageWidth >CurrentMaxSide ||
            ImageHeight > CurrentMaxSide)
         {
            std::auto_ptr<Graphics::TBitmap>
               Image(new Graphics::TBitmap());
```

```
            // Указание высоты и ширины рисунка
            Image->Width = ImageWidth;
            Image->Height = ImageHeight;

            // Отображение рисунка из списка ImageList в канве
            ImageList->Draw(Image->Canvas, 0, 0,
                            StrToInt(Value), true);

            if(ImageWidth > CurrentMaxSide)
               ImageWidth = CurrentMaxSide;
            if(ImageHeight > CurrentMaxSide)
               ImageHeight = CurrentMaxSide;

            // Определение области рисования
            TRect ImageRect = Rect(ARect.Left + 2,
                                   ARect.Top + 1,
                                   ARect.Left + 2 + ImageWidth,
                                   ARect.Top + 1 + ImageHeight);

            // Отображение рисунка в канве с помощью
            // функции StretchDraw
            ACanvas->StretchDraw(ImageRect, Image);
         }
         else
         {
            // Отображение рисунка в канве непосредственно из
            // списка ImageList с границей шириной 1 пиксель.
            ImageList->Draw(ACanvas,
                            ARect.Left + 2,
                            ARect.Top + 1,
                            StrToInt(Value),
                            true);
         }

         ValueRect = Rect(ARect.Left + ImageWidth + 2,
                          ARect.Top,
                          ARect.Right,
                          ARect.Bottom);
      }
   }
   __finally
   {
      // Независимо от исхода рисования нужно
      // отобразить текст.
      inherited::PropDrawValue(ACanvas, ValueRect, ASelected);
   }
}
```

Замечания по поводу отображения рисунков

Редактор свойств `TImageIndexPropertyEditor` выполняет все нужные нам действия за исключением одного: при подгонке большого изображения с помощью метода `StretchDraw()` по размеру канвы не сохраняется исходное соотношение ширины и высоты изображения. Код этой операции не приводится здесь, потому что он существенно усложняет код всех четырех методов отображения рисунков. В целом это может усложнить восприятие основных принципов создания редактора свойств. Однако эта операция, вероятно, представляет собой наиболее корректный способ использования этого редактора свойств.

Связь со списком `TCustomImageList` родительского компонента

Здесь представлен способ использования свойства `ImageIndex` класса `TMenuItem` в коде, необходимом для отображения рисунка из списка `TCustomImageList` родительского компонента в свойстве-индексе. Здесь показаны только определение класса и метод `GetParentImageList()` (вместо метода `GetComponentImageList()`), потому что код других методов имеет аналогичную структуру.

В листинге 10.24 приведен код редактора свойств `TMenuItemImageIndexProperty`.

Листинг 10.24. Код определения редактора свойств `TMenuItemImageIndexProperty`

```
#include "Dsgnintf.hpp"
   class PACKAGE TMenuItemImageIndexProperty :
      public TIntegerProperty
{
   typedef TIntegerProperty inherited;

private:
   static const int Border = 2;
   static const int MaxImageWidth = 64;
   static const int MaxImageHeight = 64;

protected:
   virtual Imglist::TCustomImageList*
      __fastcall GetParentImageList(void);

public:
   virtual TPropertyAttributes __fastcall GetAttributes(void);
   virtual void __fastcall GetValues(Classes::TGetStrProc Proc);

   DYNAMIC void __fastcall
      ListMeasureWidth(const AnsiString Value,
                       Graphics::TCanvas* ACanvas,
                       int& AWidth);

   DYNAMIC void __fastcall
      ListMeasureHeight(const AnsiString Value,
                        Graphics::TCanvas* ACanvas,
```

```
                        int& AHeight);

   DYNAMIC void __fastcall
      ListDrawValue(const AnsiString Value,
                    Graphics::TCanvas* ACanvas,
                    const Windows::TRect& ARect,
                    bool ASelected);

   DYNAMIC void __fastcall
      PropDrawValue(Graphics::TCanvas* ACanvas,
                    const Windows::TRect& ARect,
                    bool ASelected);

   // Это свойство доступно только для чтения и используется
   // как указатель на список рисунков компонента.
   __property Imglist::TCustomImageList* ParentImageList =
                {read=GetParentImageList};

protected:
   #pragma option push -w-inl
   inline __fastcall virtual
      TMenuItemImageIndexProperty(
         const _di_IFormDesigner ADesigner,
         int APropCount) : TIntegerProperty(ADesigner,
                                            APropCount)
   {}
   #pragma option pop

public:
   #pragma option push -w-inl
   inline __fastcall virtual ~TMenuItemImageIndexProperty(void)
   {}
   #pragma option pop
};
```

В листинге 10.25 показан код функции `GetParentImageList()`.

Листинг 10.25. Код метода `GetParentImageList()`

```
Imglist::TCustomImageList* __fastcall
   TMenuItemImageIndexProperty::GetParentImageList(void)
{
   TMenuItem* Component =
      dynamic_cast<TMenuItem*>(GetComponent(0));

   if(Component)
   {
      TMenuItem* ParentMenuItem = Component->Parent;

      while(ParentMenuItem != 0 &&
```

```
            ParentMenuItem->SubMenuImages == 0)
      {
         ParentMenuItem = ParentMenuItem->Parent;
      }

      if(ParentMenuItem != 0)
         return ParentMenuItem->SubMenuImages;
      else
      {
         TMenu* ParentMenu = Component->GetParentMenu();
         if(ParentMenu != 0) return ParentMenu->Images;
      }

   }

   return 0;
}
```

Как всегда, первый и наиболее важный этап заключается в получении указателя на компонент редактируемого свойства, в данном случае это указатель на компонент `TMenuItem`. После этого можно получить указатель на родительский компонент `TMenuItem`, т.е. другой компонент типа `TMenuItem` (иначе говоря, текущий компонент `TMenuItem` является подчиненным элементом меню) или компонент типа `TMenu` (или его наследник типа `TPopupMenu`). Если компонент `TMenuItem` является подчиненной командой меню, то нужно проверить наличие списков изображений для подчиненных команд меню (`SubMenuImages TCustomImageList*`) в родительском компоненте `TMenuItem`. Родительский компонент `TMenuItem` (если таковой существует) может, в свою очередь, иметь свой родительский компонент `TMenuItem` со списками изображений подчиненных команд `SubMenuImages` и т.д. Нужно последовательно выполнить поиск всех возможных родителей, пока не будет найден самый последний. При нахождении родительского компонента `TMenuItem` с ненулевым свойством `SubMenuImages` итерации прекращаются и возвращается указатель на список `TCustomImageList`. В противном случае свойство `Parent` будет иметь нулевой указатель. В таком случае родительский компонент `TMenu` может быть получен с помощью функции `GetParentMenu()`. Далее возвращается указатель `TCustomImageList*` свойства `Images` компонента `TMenu`. Структура большинства индексов изображений `ImageIndex` в редакторах свойств компонентов, которые связаны со списком `TCustomImageList` родительского компонента, более проста, так как они имеют только один родительский компонент.

Существует гораздо более простой и тонкий способ реализации этой функции, который заключается в использовании функции `GetImageList()` компонента `TMenuItem` для возвращения подходящего указателя `TCustomImageList`. Вот как выглядит код функции `GetParentImageList()`.

```
Imglist::TCustomImageList* __fastcall
   TMenuItemImageIndexProperty::GetParentImageList(void)
{
   TMenuItem* Component =
      dynamic_cast<TMenuItem*>(GetComponent(0));

   if(Component)
   {
```

```
      return Component->GetImageList();
   }

   return 0;
}
```

Более сложный метод представлен здесь для демонстрации принципов получения указателя родительского компонента.

Продолжая обсуждение этой темы, рассмотрим редактор свойства `ImageIndex` для класса `THeaderSection`, производного от класса `TCollectionItem` (и далее от класса `TPersistent`) и используемого в компоненте `THeaderControl`. Рассмотрим редактор свойств `THeaderSectionIndexProperty`, на примере которого показан другой полезный метод создания редакторов свойств и представлено более общее решение для создания и регистрации редакторов свойств `ImageIndex` для класса, родитель которого содержит указатель `TCustomImageList`.

Здесь представлен код только функции `GetParentImageList()` (листинг 10.26). Как и при работе с редактором свойств `TMenuItemImageIndexProperty`, основная задача здесь заключается в получении указателя на родительский компонент, который содержит свойство `ImageIndex`. Класс `THeaderSection` происходит от класса `TCollectionItem` и, следовательно, имеет соответствующий класс `THeaderSections` производный от класса-контейнера `TCollection`. Указатель на объект-контейнер `THeaderSections` класса `THeaderSection` можно получить с помощью его свойства `Collection`. Возвращаемый указатель `TCollection*` при этом должен быть динамически (`dynamic_cast`) приведен к типу указателя `THeaderSections`. После получения указателя на контейнер элементов набора необходимо определить родителя этого набора, который содержит нужный список изображений. Единственный доступный нам метод — защищенная функция `GetOwner()`, которая возвращает указатель `TPersistent*`. К сожалению, эта функция является защищенной, а потому не существует способа прямого доступа к ней. Однако существует другой простой способ вызова этой функции.

Для вызова защищенного или закрытого члена класса необходимо выполнить два действия. Во-первых, нужно создать класс-наследник для данного класса и расширить область видимости нужных членов класса (функций или свойств). Иначе говоря, нужные члены класса следует переопределить как открытые. Такой класс называется *классом доступа* (*access class*). После этого доступ к нужным членам класса можно получить с помощью статического приведения (`static_cast`) типа указателя данного класса к типу его класса доступа. Этот метод демонстрируется в листинге 10.26.

Листинг 10.26. Код функции `THeaderSectionIndexProperty::GetParentImageList()`

```
//-------------------------------------------------------------//
//                                                             //
//         THeaderSectionsAccessImageIndexProperty             //
//                                                             //
//-------------------------------------------------------------//

class THeaderSectionsAccess : public THeaderSections
{
public:
   DYNAMIC Classes::TPersistent* __fastcall GetOwner(void);
};
```

```
//---------------------------------------------------------//
//                                                         //
//            THeaderSectionImageIndexProperty             //
//                                                         //
//---------------------------------------------------------//

//---------------------------------------------------------//
Imglist::TCustomImageList* __fastcall
   THeaderSectionImageIndexProperty::GetParentImageList(void)
{
   THeaderSection* Component =
      dynamic_cast<THeaderSection*>(GetComponent(0));

   if(Component)
   {
      THeaderSections* HeaderSections =
         dynamic_cast<THeaderSections*>(Component->Collection);

      if(HeaderSections)
      {
         TPersistent* Owner =
            static_cast<THeaderSectionsAccess*>
               (HeaderSections)->GetOwner();

         THeaderControl*HeaderControl =
            dynamic_cast<THeaderControl*>(Owner);

         if(HeaderControl)
         {
            return HeaderControl->Images;
         }
      }
   }

   return 0;
}
//---------------------------------------------------------//
```

Сразу после получения указателя на владельца полученного набора он может быть динамически приведен к заданному типу, т.е. в этом примере к типу указателя на класс `THeaderControl`. А затем можно возвратить указатель на список `TCustomImageList` компонента `THeaderControl`.

Обобщенное решение для свойств `ImageIndex`

Библиотека VCL содержит несколько классов, производных от классов `TPersistent` и `TComponent`, со свойствами `ImageIndex`. В предыдущих разделах рассмотрены два из них, а остальные перечислены в табл. 10..10.

Таблица 10.10. Классы библиотеки VCL со свойствами `ImageIndex`

Класс	Иерархия наследования
TCoolBand	→TCollectionItem→**TPersistent**
TCustomAction	→**TComponent**→TPersistent
THeaderSection	→TCollectionItem→**TPersistent**
TListColumn	→TCollectionItem→**TPersistent**
TMenuItem	→**TComponent**→TPersistent
TTabSheet	→TWinControl→TControl→**TComponent**→Tpersistent
TToolButton	→TGraphicControl→TControl→**TComponent**→TPersistent

Обратите внимание, что в табл. 10.10 указано различие между классами, происходящими от класса `TComponent` и теми, которые происходят только от класса `TPersistent`. Кроме того, три класса — `TListColumn`, `TMenuItem` и `TToolButton` — обладают некоторыми особенностями определения родительского списка `TCustomImageList*`. Эти особенности прояснятся позже, при знакомстве со способом создания обобщенных редакторов свойств `ImageIndex`.

Для каждого свойства `ImageIndex` этих классов можно и нужно определить свой редактор. Необходимый для этого код представлен в модуле `PropertyEditors` пакета `EnhancedEditors` на прилагаемом к книге компакт-диске.

Можно создать два более обобщенных редактора свойств: один для классов, производных от класса `TPersistent`, а другой — для классов, производных от класса `TComponent`. Независимо от этого рекомендуется определить базовый класс для редактора свойств, чтобы на основе этого базового класса можно было создавать новые редакторы свойств. Причем в производном классе придется переопределить только метод `GetImageList()`. Пример такого определения класса приводится в листинге 10.27.

Листинг 10.27. Определение базового класса для редактора свойств `TImageIndexProperty`

```
class PACKAGE TImageIndexProperty : public TIntegerProperty
{
   typedef TIntegerProperty inherited;

private:
   static const int Border = 2;
   static const int MaxImageWidth = 64;
   static const int MaxImageHeight = 64;

protected:
   virtual Imglist::TCustomImageList*
      __fastcall GetImageList(void) = 0;

public:
   virtual TPropertyAttributes __fastcall GetAttributes(void);
   virtual void __fastcall GetValues(Classes::TGetStrProc Proc);
```

```
   DYNAMIC void __fastcall
      ListMeasureWidth(const AnsiString Value,
                       Graphics::TCanvas* ACanvas,
                       int& AWidth);

   DYNAMIC void __fastcall
      ListMeasureHeight(const AnsiString Value,
                        Graphics::TCanvas* ACanvas,
                        int& AHeight);

   DYNAMIC void __fastcall
      ListDrawValue(const AnsiString Value,
                    Graphics::TCanvas* ACanvas,
                    const Windows::TRect& ARect,
                    bool ASelected);

   DYNAMIC void __fastcall
      PropDrawValue(Graphics::TCanvas* ACanvas,
                    const Windows::TRect& ARect,
                    bool ASelected);

   // Это свойство доступно только для чтения и используется
   // как указатель на список рисунков компонента.
   __property Imglist::TCustomImageList*
      RemoteImageList = {read=GetImageList};

protected:
   #pragma option push -w-inl
   inline __fastcall virtual
      TImageIndexProperty(const _di_IFormDesigner ADesigner,
                          int APropCount)
   : TIntegerProperty(ADesigner, APropCount)
   {}
   #pragma option pop

public:
   #pragma option push -w-inl
   inline __fastcall virtual ~TImageIndexProperty(void)
   {}
   #pragma option pop
};
```

Класс `TImageIndexProperty` является абстрактным базовым классом, потому что содержит виртуальную функцию `GetImageList()`, которая реализуется в производных классах. Код класса `TImageIndexProperty` аналогичен показанному выше.

Создавая на основе базового класса редакторы свойств `ImageIndex`, нужно позаботиться о создании метода `GetImageList()`. Пример кода такого метода для редактора свойств `TPersistentDerivedImageIndexProperty` показан в листинге 10.28. Обратите внимание, что переменная `FParentImageListName` является закрытым членом типа `AnsiString`, который инициализируется в конструкторе значением `"Images"`.

Листинг 10.28. Код редактора свойств `TPersistentDerivedImageIndexProperty`

```
//-----------------------------------------------------------//
//                                                           //
//                     TComponentAccess                      //
//                                                           //
//-----------------------------------------------------------//
class TComponentAccess : public TComponent
{
public:
   DYNAMIC Classes::TPersistent* __fastcall GetOwner(void);
};
//-----------------------------------------------------------//
//                                                           //
//           TPersistentDerivedImageIndexProperty            //
//                                                           //
//-----------------------------------------------------------//
__fastcall
   TPersistentDerivedImageIndexProperty::
      TPersistentDerivedImageIndexProperty
         (const _di_IFormDesigner ADesigner,
          int APropCount)
   : TImageIndexProperty(ADesigner,
                         APropCount)
{
   FParentImageListName = "Images";
}
//-----------------------------------------------------------//

Imglist::TCustomImageList* __fastcall
   TPersistentDerivedImageIndexProperty::GetImageList(void)
{
   TPersistent* Parent =
    static_cast<TComponentAccess*>(GetComponent(0))->GetOwner();

   while(Parent != 0 && !dynamic_cast<TComponent*>(Parent))
   {
      Parent =
         static_cast<TComponentAccess*>(Parent)->GetOwner();
   }

   if(Parent == 0) return 0;

   PPropInfo PropInfo =
      Typinfo::GetPropInfo
         (static_cast<PTypeInfo>(Parent->ClassInfo()),
          FParentImageListName);

   if(PropInfo == 0) return 0;
```

```
   return
   (
      reinterpret_cast<TCustomImageList*>(Typinfo::GetOrdProp
                                          (Parent, PropInfo))
   );
}
//---------------------------------------------------------//
```

Этот код не так сложен, как может показаться на первый взгляд. Класс доступа здесь используется для расширения области видимости защищенного метода `GetOwner()` класса `TComponent`. Сначала требуется найти непосредственного родителя компонента, свойство которого редактируется, рассматривая его как объект типа `TComponent` и вызывая метод `GetOwner()`. Если у компонента существует родитель, но его тип, отличен от `TComponent`, то следует проверить наличие родителя, производного от `TComponent`. Затем у него также следует проверить наличие родителя, производного от `TComponent` и т.д. После завершения цикла `while` будет либо найден самый последний родитель, производный от `TComponent`, либо свойство `Parent` будет иметь значение `NULL`. При наличии производного от `TComponent` родителя можно попытаться получить информацию `PPropInfo` для свойства с тем же именем, содержащегося в члене `FParentImageListName` типа `AnsiString` и относящегося к `Parent`. Для этого можно использовать следующую строку кода.

```
static_cast<PTypeInfo>(Parent->ClassInfo())
```

Если существует совпадающее свойство, функция `GetOrdProp()` будет использована для получения указателя `TCustomImageList`. Так как функция `GetOrdProp()` возвращает значение типа `int`, оно должно быть приведено к правильному типу указателя с помощью `reinterpret_cast`.

Следовательно, код метода `GetImageList()` будет использован для любого производного от `TPersistent` класса со свойством `ImageIndex`, при условии, что родительское имя свойства списка изображений называется `"Images"` (например, для `TCoolBand` и `THeaderSection`). В противном случае класс должен быть производным от данного, а имя `FParentImageListName` типа `AnsiString` — изменено. Это делается с помощью класса `TListColumn`, родительское свойство списка изображений которого называется `"SmallImages"`.

Аналогично выглядит код класса редактора свойств `TComponentDerivedImageIndexProperty` для производных от `TComponent` классов. Код метода `GetImageList()` показан в листинге 10.29.

Листинг 10.29. Код метода `GetImageList()`

```
Imglist::TCustomImageList*__fastcall
   TComponentDerivedImageIndexProperty::GetImageList(void)
{
   TComponent* Parent =
      static_cast<TComponentAccess*>
         (GetComponent(0))->GetParentComponent();
```

```
   if(Parent == 0) return 0;

   PPropInfo PropInfo =
      Typinfo::GetPropInfo(static_cast<PTypeInfo>
         (Parent->ClassInfo()),
          FParentImageListName);

   if(PropInfo == 0) return 0;

   return
   (
   reinterpret_cast<TCustomImageList*>
      (Typinfo::GetOrdProp(Parent, PropInfo))
   );
}
```

Код метода `GetImageList()` редактора свойств `TComponentDerivedImageIndexProperty` практически идентичен коду этого метода для редактора свойств `TPersistentDerivedImageIndexProperty`. Единственное отличие заключается в том, что свойство `Parent`, полученное как производный от `TComponent` класс, существенно упрощает код. Этот код может быть использован для производных от `TComponent` классов со свойством `ImageIndex` при условии, что родительское свойство `TCustomImageList` называется `"Images"` (например, `TTabSheet`). Он также может применяться для `TMenuItem` и `TToolButton`, но в случае `TMenuItem` также необходимо рассмотреть свойство `SubMenuImages`. Предыдущая реализация компонента `TMenuItem` все же лучше. Для компонента `TToolButton` оптимальнее всего было бы использовать свойство `ImageIndex` для родительского свойства `Images` только в тех случаях, когда кнопка доступна. В противном случае следует использовать родительское свойство `DisabledImages`. В листинге 10.30 показан код метода `GetImageList()` редактора свойств `TToolButtonImageIndexProperty`.

Листинг 10.30. Код метода `TToolButtonImageIndexProperty::GetImageList()`

```
Imglist::TCustomImageList* __fastcall
   TToolButtonImageIndexProperty::GetImageList(void)
{
   TToolButton* Component =
      dynamic_cast<TToolButton*>(GetComponent(0));
   if(Component)
   {
      TToolBar* ParentToolBar =
         dynamic_cast<TToolBar*>(Component->Parent);

      if(ParentToolBar)
      {
         if(Component->Enabled) return ParentToolBar->Images;
         else return ParentToolBar->DisabledImages;
      }
```

```
    }

    return 0;
}
```

Код в листинге 10.29 имеет достаточно простую структуру и понятен без лишних комментариев. Если редактируемая кнопка `TToolButton` неактивна (`Enabled = false`), то будет использован список изображений `DisabledImages` в родительском свойстве `TToolBar`; в противном случае — обычный список изображений `Images`.

Иерархия классов для этого обобщенного решения показана на рис. 10.7.

Из табл. 10.10 известно, что редактор свойств `TPersistentDerivedImageIndexProperty` используется для компонентов `TCoolBand` и `THeaderSection`, а редактор свойств `TComponentDerivedImageIndexProperty` — для компонентов `TCustomAction` и `TTabSheet`.

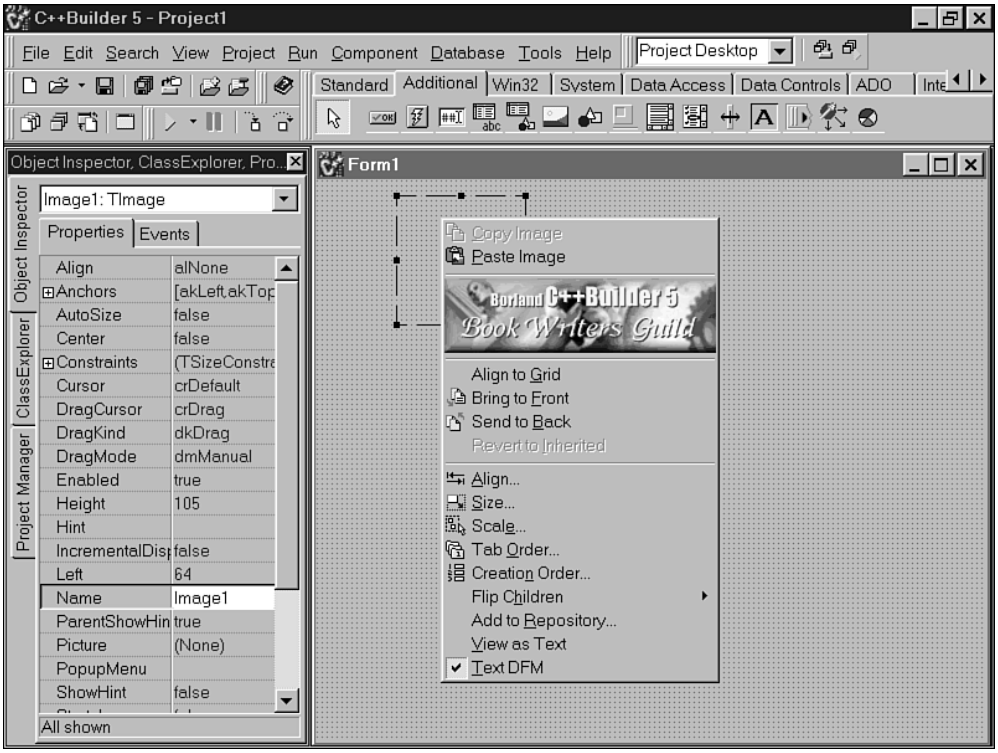


Рис. 10.7. Иерархия наследования редакторов свойств `TImageIndexProperty`

Создание пользовательских редакторов компонентов

В предыдущих разделах показано, что редакторы свойств используются в компонентах для создания более интуитивно понятного и надежного интерфейса времени создания

приложения. Редакторы компонентов выполняют ту же функцию для всего компонента в целом. Пользовательские редакторы компонентов также позволяют настраивать контекстное меню каждого компонента (которое отображается после щелчка правой кнопкой мыши), а также определить поведение компонента после двойного щелчка на форме. В этом разделе представлены основные принципы создания пользовательских редакторов компонентов. Редакторы компонентов позволяют настроить предлагаемый по умолчанию процесс редактирования компонента, а также расширить эту операцию другими функциями. Для создания редакторов компонентов используются классы `TComponentEditor` и `TDefaultEditor`, взаимосвязь которых схематически показана на рис. 10.8.

На рис. 10.8 также приведена дополнительная информация о виртуальных функциях, которые следует переопределить при настройке пользовательского редактора компонентов. Все это рассматривается в этом и последующих разделах с подробным описанием используемых методов.

Как уже говорилось, создание пользовательского редактора компонентов позволяет задавать действия, выполняемые по умолчанию в ответ на щелчок правой кнопкой мыши или двойной щелчок левой кнопкой мыши в IDE-среде. В табл. 10.16 перечислены события мыши и связанные с ними виртуальные функции вместе с описанием принимаемых по умолчанию действия компонентов.

На рис. 10.8 показано, что классы `TComponentEditor` и `TDefaultEditor` обладают одинаковой функциональностью. Как показано в табл. 10.11, они отличаются лишь реализацией метода `Edit()`. Выбор одного из этих двух классов в качестве базового для пользовательских редакторов компонентов определяется следующими критериями.

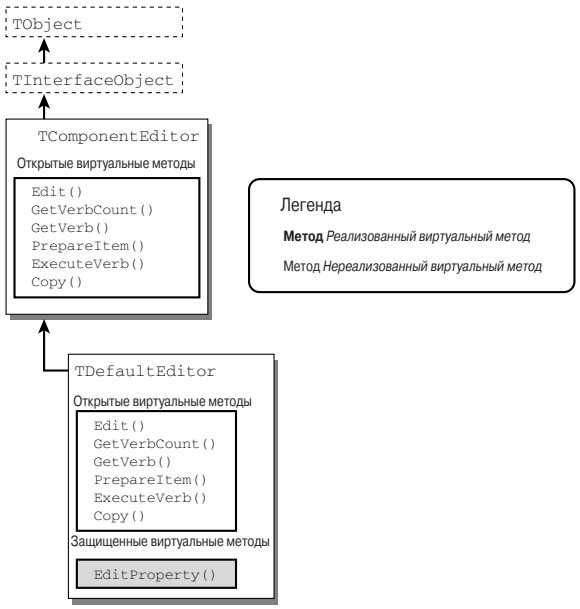


Рис. 10.8. Иерархия наследования класса `TComponentEditor`

Если нужно, чтобы редактор компонента при двойном щелчке на нем генерировал пустой обработчик для трех принимаемых по умолчанию событий или для некоторого заданного события, то в качестве базового класса следует выбрать `TDefaultEditor`. В противном случае следует использовать класс `TComponentEditor`. Если для компонента не создан пользовательский редактор свойств, то C++Builder автоматически использует класс `TDefaultEditor`.

Таблица 10.11. События мыши для редакторов `TComponentEditor` и `TDefaultEditor`

Действие пользователя	Реакция по умолчанию	Виртуальная функция
Щелчок правой кнопкой мыши	Отображение контекстного меню компонента	Сначала вызывается функция `GetVerbCount()`. Она используется для возвращения количества элементов, которые следует разместить в верхней части контекстного меню. Затем вызывается функция `GetVerb()`, которая позволяет представить команды меню в виде строк `AnsiString` Функция `PrepareItem()` вызывается перед отображением команды меню, что позволяет настроить ее. Функция `ExecuteVerb()` вызывается только в том случае, если щелчок выполнен на одной из вновь добавленных команд меню. Именно здесь располагается новый код действий компонента
Двойной щелчок кнопкой мыши	Принимаемые по умолчанию действия зависит от класса — родителя данного класса. `TComponentEditor`: если в контекстном меню созданы команды, то выполняется первая команда меню. `TDefaultEditor`: Пустой обработчик создается для событий `OnChange`, `OnCreate` или `OnClick`, какое бы из них не появилось первым в списке свойств событий компонента. Если ни одного из них нет для данного компонента, обработчик создается для первого события в списке. Если событий в компоненте нет вообще, то в таком случае ничего не происходит	Вызывается функция `Edit()`. Здесь располагается код нужных действий

Если нужно, чтобы редактор компонента при двойном щелчке на нем генерировал пустой обработчик для трех принимаемых по умолчанию событий или для некоторого заданного события, то в качестве базового класса следует выбрать `TDefaultEditor`. В противном случае следует использовать класс `TComponentEditor`. Если для компонента не создан пользовательский редактор свойств, то C++Builder автоматически использует класс `TDefaultEditor`.

После выбора базового класса для редактора компонента нужно переопределить соответствующие методы. В табл. 10.12 перечислены методы обоих классов и их назначение.

Таблица 10.12. Виртуальные функции классов TComponentEditor и TDefaultEditor

Виртуальная функция	Назначение
`int GetVerbCount(void)`	Возвращает значение типа `int`, представляющее количество команд меню (глаголов), которые будут созданы
`AnsiString GetVerb(int Index)`	Возвращает значение типа `AnsiString`, представляющее название команды меню, которое будет представлено в контекстном меню. При этом следует использовать такие обозначения: символ `&` следует применять для обозначения клавиши быстрого доступа; многоточие `...` следует помещать после команды, которая приводит к открытию диалогового окна; символ — следует применять для создания разделительной полосы
`void PrepareItem(int Index, const Menus::TMenuItem* AItem)`	Функция `PrepareItem()` вызывается для каждой команды меню и передает параметр `TMenuItem`, который используется для представления этой команды в контекстном меню. Это позволяет нужным образом настроить команду (ее, например, можно скрыть, задав значение `false` для свойства `Visible`)
`void ExecuteVerb(int Index)`	Функция `ExecuteVerb()` вызывается при выборе одной из пользовательских команд меню. Индекс `Index` обозначает выбранную команду
`void Edit(void)`	Функция `Edit()` вызывается двойным щелчком на компоненте. Выполняемые при этом действия определяются пользователем. Поведение по умолчанию описано в табл. 10.16
`void EditProperty(TpropertyEditor* PropertyEditor, bool& Continue, bool& FreeEditor)` (только `TDefaultEditor`)	Используется для определения события, генерируемого после двойного щелчка на компоненте
`void Copy(void)`	Функцию `Copy()` следует вызывать при копировании компонента в буфер обмена. Ее переопределяют только для копирования объекта в особом формате, например для копирования изображения из графического компонента

Определения редакторов компонентов, производных от `TComponentEditor` и `TDefaultComponent`, приведены в листингах 10.31 и 10.32.

Листинг 10.31. Код определения редактора компонента, производного от TComponentEditor

```
#include "dsgnintf.hpp"

class TCustomComponentEditor : public TComponentEditor
{
   typedef TComponentEditor inherited;
```

```
public:
   // Для двойного щелчка
   virtual void __fastcall Edit(void);

   // Для щелчка правой кнопкой мыши
   // КОНТЕКСТНОЕ МЕНЮ - Этап 1
   virtual int __fastcall GetVerbCount(void);

   //                  - Этап 2
   virtual AnsiString __fastcall GetVerb(int Index);

   //                  - Этап 3 (ПО ЖЕЛАНИЮ)
   virtual void __fastcall PrepareItem(int Index,
                              const Menus::TMenuItem* AItem);

   //                  - Этап 4
   virtual void __fastcall ExecuteVerb(int Index);

   // Копирование в буфер обмена
   virtual void __fastcall Copy(void);

public:
   #pragma option push -w-inl
   inline __fastcall virtual
      TCustomComponentEditor(Classes::TComponent* AComponent,
                             _di_IFormDesigner ADesigner)
      : TComponentEditor(AComponent, ADesigner)
   {}
   #pragma option pop

public:
   #pragma option push -w-inl
   inline __fastcall virtual ~TCustomComponentEditor(void) {}
   #pragma option pop
};
```

Листинг 10.32. Код определения редактора компонента, производного от `TDefaultEditor`

```
#include "dsgnintf.hpp"

class TCustomDefaultEditor : public TDefaultEditor
{
   typedef TDefaultEditor inherited;

protected:
   // Для двойного щелчка
   // СОБЫТИЕ - ВЫБОР
   virtual void __fastcall EditProperty
```

```
      (TPropertyEditor* PropertyEditor,
       bool& Continue,
       bool& FreeEditor);

public:
   // Для щелчка правой кнопкой мыши
   // КОНТЕКСТНОЕ МЕНЮ - Этап 1
   virtual int __fastcall GetVerbCount(void);

   //                 - Этап 2
   virtual AnsiString __fastcall GetVerb(int Index);

   //                 - Этап 3 (ПО ЖЕЛАНИЮ)
   virtual void __fastcall PrepareItem
      (int Index,
       const Menus::TMenuItem* AItem);

   //                 - Этап 4
   virtual void __fastcall ExecuteVerb(int Index);

   // Копирование в буфер обмена
   virtual void __fastcall Copy(void);

public:
   #pragma option push -w-inl
   inline __fastcall virtual
      TCustomDefaultEditor(Classes::TComponent* AComponent,
                           _di_IFormDesigner ADesigner)
      : TDefaultEditor(AComponent,ADesigner)
   {}
   #pragma option pop

public:
   #pragma option push -w-inl
   inline __fastcall virtual ~TCustomDefaultEditor(void) {}
   #pragma option pop
};
```

Между определениями редакторов компонентов в листингах 10.31 и 10.32 не так много различий. Действительно, методы реализации команд контекстного меню идентичны. Различие между этими классами заключается в том, что для организации отклика на двойной щелчок мыши для класса, производного от `TComponentEditor`, нужно переопределить метод `Edit()`, а для класса `TDefaultEditor` — метод `EditProperty()`.

В следующих разделах эти виртуальные методы рассматриваются более подробно. Причем информация, представленная для метода `Edit()`, применима только для классов, производных от `TComponentEditor`, а информация представленная для метода `EditProperty()` — только для классов, производных от `TDefaultEditor`. Обратите внимание, что имена классов отражают этот факт. Класс `TCustomComponentEditor` является гипотетическим производным классом от `TComponentEditor`, класс `TCustomDefaultEditor` — от `TDefaultEditor`, а класс `TMyCustomEditor` может быть производным любого из них.

Метод Edit()

Основное назначение переопределенного метода Edit() заключается в отображении для пользователя формы, в которой он сможет легко редактировать значения компонента. Прекрасным примером является редактор компонента TChart во вкладке Additional диалогового окна Component Palette. Поэтому необходимый для него код аналогичен коду метода Edit() класса TPropertyEditor в разделе о создании пользовательских редакторов свойств выше в этой главе. Для реализации формы можно выбрать один из двух способов. С помощью первого форма может обновляться мгновенно, а с помощью второго — только после ее закрытия.

Кроме того, следует иметь в виду следующее: при каждом обновлении компонента *должен* вызываться метод Modified() свойства Designer класса TComponentEditor. Именно таким образом IDE-среда получает указание о необходимости модификации компонента. Следовательно, после каждого фрагмента кода с модификацией компонента нужно использовать следующую строку кода.

```
if(Designer) Designer->Modified();
```

Оператор if используется здесь для проверки значения, возвращаемого Designer, перед вызовом метода Modified(). При возвращении нулевого значения ничего нельзя сделать, так как это означает, что IDE-среда недоступна. Чтобы форма могла изменять значения свойств компонента, необходимо установить связь между формой и компонентом точно так же, как это делалось ранее при работе с редакторами свойств. Это достаточно простая задача, выполняемая в два этапа. Во-первых, в определении формы следует объявить открытое свойство, которое имеет тип указателя на редактируемый компонент. Во-вторых, нужно получить указатель на текущий экземпляр редактируемого компонента. Это делается с помощью свойства Component класса TComponentEditor следующим образом.

```
TMyComponent* MyComponent =
   dynamic_cast<TMyComponent*>(Component);
```

Полученный указатель можно сравнить со значением свойства, которое содержит указатель на компонент данной формы. Однако при этом нужно также сослаться на Designer, доступный в этой форме, чтобы IDE-среда могла получить информацию о внесенных в компонент изменениях. Ссылку можно передать в конструктор формы в виде параметра. В таком случае компонент может быть модифицирован непосредственно на основе значения свойства формы. Соответствующий код представлен в листинге 10.33. Помните о том, что после кода модификации компонента в форме нужно организовать вызов метода Designer->Modified().

Листинг 10.33. Код мгновенного обновления в методе Edit()

```
// Сначала приводится важный код класса TComponentEditorForm

// В ЗАГОЛОВОЧНОМ ФАЙЛЕ
//---------------------------------------------------------------//
#ifndef MyComponentEditorFormH
#define MyComponentEditorFormH
//---------------------------------------------------------------//
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include "HeaderDeclaringTComponentClass"
//---------------------------------------------------------------//
```

```
class TMyComponentEditorForm : public TForm
{
__published: // Компоненты, управляемые IDE-средой

private:
   TComponentClass* FComponentClass;
   _di_IformDesigner& Designer;
   // Далее идут другие объявления, например, для восстановления
   // значений в случае щелчка на кнопке 'Cancel'

protected:
   void __fastcall SetComponentClass(TComponentClass* Pointer);

public:
   __fastcall TMyComponentEditorForm
      (TComponent* Owner,
       _di_IformDesigner&EditorDesigner);

   __property TComponentClass*ComponentClass =
      {read=FComponentClass,
       write=SetComponentClass};

   // Другие объявления
};
//---------------------------------------------------------//
#endif

// В ФАЙЛЕ РЕАЛИЗАЦИИ
//---------------------------------------------------------//
#include <vcl.h>
#pragma hdrstop

#include "MyComponentEditorForm.h"
//---------------------------------------------------------//
#pragma package(smart_init)
#pragma resource "*.dfm"
//---------------------------------------------------------//
__fastcall
   TMyComponentEditorForm::
      TMyComponentEditorForm(TComponent* Owner,
                             _di_IformDesigner& EditorDesigner)
   : TForm(Owner), Designer(EditorDesigner)
{
}
//---------------------------------------------------------//
void __fastcall
   TMyPropertyForm::SetComponentClass(TComponentClass* Pointer)
{
   FComponentClass = Pointer;
   if(FComponentClass != 0)
```

```
   {
      // Сохранение текущих значений компонента и их отображение
   }
}
//---------------------------------------------------------//

// Код метода Edit()
#include "MyComponentEditorForm.h" // Запомним это

void __fastcall TCustomComponentEditor::Edit(void)
{
   // Создание формы
   std::auto_ptr<TMyComponentEditorForm*>
      MyComponentEditorForm(new TMyComponentEditorForm(0));

   // Связь со свойством компонента
   MyComponentEditorForm->ComponentClass
      = dynamic_cast<TComponentClass*>(Component);

   // Отображение формы. Всю остальную работу выполняет форма.
   MyPropertyForm->ShowModal();
}
```

Как и при работе с формами пользовательских редакторов свойств, текущие значения свойств компонента могут быть сохранены, если свойство `Component` формы связано с этим компонентом. Это позволяет отменить выполняемые действия и восстановить предыдущие значения. При этом следует учитывать возможность появления нулевого указателя после выполнения динамического приведения типа (`dynamic_cast`). Этого следует избегать, потому что в таком случае в форме нельзя будет изменить свойство компонента. В этом случае для уведомления пользователя следует организовать вызов и обработку исключительной ситуации.

Второй способ реализации метода `Edit()` достаточно прост. Форма отображается в виде диалогового окна, а после ее возвращения все введенные значения присваиваются компоненту. Указатель на текущий экземпляр редактируемого компонента при этом может быть получен из свойства `Component` класса `TComponentEditor`, как показано ниже.

```
TMyComponent* MyComponent =
   dynamic_cast<TMyComponent*>(Component);
```

В этом подходе код метода `Edit()` имеет больший размер, потому что значения свойств компонента должны быть присвоены форме после ее создания, но перед отображением. При закрытии формы эти значения должны быть присвоены нужным свойствам компонента. В листинге 10.34 показан этот код метода `Edit()`.

Листинг 10.34. Код метода `Edit()` для обновления значений после закрытия формы

```
#include "MyComponentEditorDialog.h" // Для диалогового окна
                                     // TMyComponentDialog !
void __fastcall TCustomComponentEditor::Edit(void)
{
   TMyComponent* MyComponent =
      dynamic_cast<TMyComponent*>(Component);
```

```
   if(MyComponent != 0)
   {
      // Создание формы
      std::auto_ptr<TMyComponentDialog*>
         MyComponentDialog(new TMyComponentDialog(0));

      // Задание текущих значений в диалоговом окне
      // MyComponentDialog->value1 = MyComponent->value1;
      // MyComponentDialog->value2 = MyComponent->value2;
      // и т.д. ...

      // Отображение формы и просмотр результатов.
      if(MyPropertyDialog->ShowModal() == IDOK)
      {
         // Задание новых значений свойств
         // MyComponent->value1 = MyComponentDialog->value1;
         // MyComponent->value2 = MyComponentDialog->value2;
         // и т.д. ...
         if(Designer)Designer->Modified();//DON 'T FORGET!
      }
   }
   else
   {
      throw EInvalidPointer
         ("Нельзя редактировать: " +
          "не доступен указатель компонента!");
   }
}
```

При описании метода `Edit()`для второго способа в листинге 10.34 не показан код диалогового окна. Дело в том, что он не имеет никаких особенностей, а потому его описание в данном случае можно опустить. Также следует иметь в виду, что, вместо непосредственного вызова диалогового окна, может быть использован класс-оболочка. В этом случае для отображения диалогового окна следует вызвать метод `Execute()`.

Метод `EditProperty()`

Переопределение метода `EditProperty()` необходимо для указания одного или нескольких возможных событий, для которых IDE-среда автоматически сгенерирует пустой обработчик после двойного щелчка на компоненте. Рассмотрим компонент, предназначенный для обмена данными через последовательный порт. Обычно для него используется событие `OnDataReceived`, которое сигнализирует о получении и наличии данных. Для создания обработчика этого события метод `EditProperty()` нужно переопределить следующим образом.

```
void __fastcall
   TCustomDefaultEditor::EditProperty
      (TPropertyEditor* PropertyEditor,
       bool& Continue,
       bool& FreeEditor)
{
```

```
   if( PropertyEditor->ClassNameIs("TMethodProperty") &&
       (CompareText(PropertyEditor->GetName(), "OnDataReceived")
       == 0) )
   {
      inherited::EditProperty(PropertyEditor,
                              Continue,
                              FreeEditor);
   }
}
```

Оператор if здесь нужен для проверки того, что имя класса редактора свойств — TMethodProperty, т.е. проверяется наличие требуемого типа редактора для обработки данного события. А затем с помощью функции CompareText() проверяется имя самого редактора свойства (OnDataReceived). Функция CompareText() возвращает 0, если ей переданы две одинаковые строки типа AnsiString. Напомним, что в функции CompareText() не учитывается регистр символов. После проверки всех этих условий вызывается родительский метод EditProperty(), т.е. метод EditProperty() класса TDefaultEditor, который генерирует пустой обработчик события. Он вызывается с помощью унаследованного модификатора типа (inherited), поэтому предыдущий код можно записать в следующем виде.

```
TDefaultEditor::EditProperty(PropertyEditor,
                             Continue, FreeEditor);
```

Необходимость использования модификатора типа вызвана тем, что при изменении класса TDefaultEditor это никак не отразится на уже имеющемся коде. В таком случае потребуется только отредактировать определение класса в заголовочном файле.

Если нужно предоставить возможность выбора события (например, в том случае, когда один и тот же редактор компонента зарегистрирован для разных компонентов), то оператор if следует заменить конструкциями if-else-if.

```
if( PropertyEditor->ClassNameIs("TMethodProperty") &&
    (CompareText(PropertyEditor->GetName(), "OnEvent1") == 0) )
{
   inherited::EditProperty(PropertyEditor,Continue,FreeEditor);
}
else if( PropertyEditor->ClassNameIs("TMethodProperty") &&
         (CompareText(PropertyEditor->GetName(), "OnEvent2")
         == 0) )
{
   inherited::EditProperty(PropertyEditor,Continue,FreeEditor);
}
else if( PropertyEditor->ClassNameIs("TMethodProperty")&&
         (CompareText(PropertyEditor->GetName(), "OnEvent3")
         == 0) )
{
   inherited::EditProperty(PropertyEditor,Continue,FreeEditor);
}
```

Конструкции if-else-if можно заменить одним оператором if с несколькими операторами ||, которые разделяют список возможных событий.

```
if( (PropertyEditor->ClassNameIs("TMethodProperty") &&
     (CompareText(PropertyEditor->GetName(), "OnEvent1") == 0 )
```

```
    ||
    (PropertyEditor->ClassNameIs("TMethodProperty") &&
     (CompareText(PropertyEditor->GetName(), "OnEvent1") == 0 )
    ||
    (PropertyEditor->ClassNameIs("TMethodProperty") &&
     (CompareText(PropertyEditor->GetName(), "OnEvent1") == 0) )
{
   inherited::EditProperty(PropertyEditor,Continue,FreeEditor);
}
```

В любом из этих вариантов будет выбрано первое подходящее событие.

Метод GetVerbCount()

Это один из немногих наиболее простых для переопределения методов. В нем достаточно указать целочисленное значение, представляющее количество дополнительных команд контекстного меню компонента. Не забудьте, что разделительная полоса также входит в это число. Для трех команд меню код будет иметь такой вид.

```
int __fastcall TMyCustomEditor::GetVerbCount(void)
{
   return 4;
}
```

Метод GetVerb()

Такую же простую структуру имеет код метода `GetVerb()`. В нем требуется организовать возврат текста типа `AnsiString` для каждой команды меню. Напомним, что символ `-` означает разделительную полосу.

```
AnsiString __fastcall TMyCustomEditor::GetVerb(int Index)
{
   switch(Index)
   {
      case 0 : return "&Edit Component...";
      case 1 : return "© 2000 Me";
      case 2 : return "-";
      case 3 : return "Do Something Else";
      default : return "";
   }
}
```

Если клавиша быстрого доступа не будет указана явно (с помощью символа `&`), то она будет определена автоматически IDE-средой. Действительно, все заданные клавиши быстрого доступа к командам контекстного меню определяются IDE-средой во время выполнения. Это позволяет избежать конфликта с клавишами быстрого доступа, определенными пользователем. При этом последние обладают большим приоритетом. При совпадении клавиш заданная клавиша переназначается для другого символа. Наконец, напомним, что разделительная полоса автоматически располагается между заданными и пользовательскими командами меню, поэтому ее необязательно указывать в качестве последнего элемента меню. Причем в данном случае ее указание никак не повлияет на вид меню, поэтому для свойства `AutoLineReduction` контекстного меню задано значение `maAutomatic` (более подробное описание этого свойства вы найдете в интерактивной справке C++Builder).

Метод PrepareItem()

Этот метод является новинкой в C++Builder 5, и его не нужно переопределять. Он позволяет выполнять более тонкую настройку команды меню: рисование пользовательской команды меню, возможность дезактивации команды меню (Enable = false), возможность скрытия команды меню (Visible = false), а также возможность добавления подчиненных команд меню. Эти операции выполняются с помощью двух параметров метода PrepareItem(). Первый параметр, Index, выполняет ту же роль, что и в предыдущих функциях контекстного меню, т.е. показывает, к какой команде меню относится вызов этой функции. Второй параметр содержит указатель на элемент меню (TMenuItem), который будет использован для его представления в контекстном меню. Он позволяет разработчику воспользоваться всеми возможностями, предлагаемыми в TMenuItem. Однако здесь следует учесть одну особенность: этот указатель имеет тип const TMenuItem, поэтому элемент меню нельзя изменить с помощью переданного указателя. Вместо него следует использовать непостоянный (non-const) указатель типа TMenuItem и только с помощью этого указателя изменять этот элемент меню. Например, в предыдущем примере для перерисовки второго элемента меню с символом авторских прав следовало бы использовать код, показанный в листинге 10.35.

Листинг 10.35. Основной код метода PrepareItem()

```
void __fastcall
   TMyCustomEditor::PrepareItem(int Index,
                                const Menus::TMenuItem* AItem)
{
   switch(Index)
   {
      case 0 : break;
      case 1 :
      {
         TMenuItem* MenuItem = const_cast<TMenuItem*>(AItem);
         // Теперь, имея этот указатель,
         // можно выполнять нужные действия.
         // Например:
         // 1.Дезактивировать команду меню
         // MenuItem->Enabled = false;
         // 2.Скрыть команду меню
         // MenuItem->Visible = false;
         // 3.Добавить изображение в команду меню
         // MenuItem->Bitmap->LoadFromResourceName
         //    (reinterpret_cast<int>(HInstance),
         //     "BITMAPNAME");
         // и т.д., например присвоить обработчик события
         // или даже создать подчиненные команды меню...
      }
               break;
      case 2  : break;
      case 3  : break;
      default : break;
   }
}
```

Обратите особое внимание на следующую строку, в которой показан способ получения указателя для редактирования элемента меню TMenuItem.

```
TMenuItem* MenuItem = const_cast<TMenuItem*>(AItem);
```

Также обратите внимание на третий пример создания изображения в элементе меню.

```
MenuItem->Bitmap->LoadFromResourceName
   (reinterpret_cast<int>(HInstance), "BITMAPNAME");
```

При этом предполагается, что файл ресурсов импортирован в пакет, который содержит рисунок по имени BITMAPNAME. В противном случае рисунок не может быть загружен в элемент меню. Это может иметь очень печальные последствия, так как IDE-среда выйдет из строя, поэтому имена рисунков следует указывать особо тщательно. Более подробную информацию по этому поводу можно найти в разделе об использовании заданных рисунков в пользовательских редакторах свойств и компонентов ниже в этой главе. Обратите внимание, что оператор reinterpret_cast используется здесь для приведения типа void* объекта HInstance к типу int, который используется параметром члена-функции LoadFromResourceName.

Создание пользовательских обработчиков событий для элементов контекстного меню

Этот процесс состоит из двух этапов.

Во-первых, нужный обработчик события должен быть создан как член-функция класса редактора компонента. Его надпись должна точно соответствовать надписи обрабатываемого события. Во-вторых, при вызове функции PrepareItem() и получении непостоянного указателя на элемент меню MenuItem этот член-функция может быть приравнен соответствующему событию элемента меню MenuItem. Например, для создания пользовательского обработчика для события OnAdvancedDrawItem элемента меню объявите функцию AdvancedDrawMenuItem1() (так как она относится к элементу меню с номером 1) с теми же параметрами, что и у OnAdvancedDrawItem в определении класса редактора компонента. Вероятно, потребуется сделать эту функцию защищенной и виртуальной, чтобы на основании этого класса можно было создать производный. Ниже показан соответствующий код в определении класса.

```
protected:
   virtual void __fastcall
      AdvancedDrawMenuItem1(System::TObject* Sender,
                            Graphics::TCanvas* ACanvas,
                            const Windows::TRect& ARect,
                            TOwnerDrawState State);
```

Пустой код реализации этой функции имеет следующий вид.

```
virtual void __fastcall TMyCustomEditor::AdvancedDrawMenuItem1
                           (System::TObject* Sender,
                            Graphics::TCanvas* ACanvas,
                            const Windows::TRect& ARect,
                            TOwnerDrawState State)
{
   // Пользовательский код рисования
}
```

На втором этапе следует эту функцию приравнять событию OnAdvancedDrawItem элемента меню MenuItem с помощью приведенной ниже строки кода, которую нужно вставить в код листинга 10.35 (после получения элемента меню MenuItem).

```
MenuItem->OnAdvancedDrawItem = AdvancedDrawMenuItem1;
```

Теперь в каждом случае возникновения события OnAdvancedDrawItem() будет выполнен пользовательский код рисования. Затем следует переопределить другие события элемента меню TMenuItem: OnMeasureItem, OnDrawItem и OnClick, что позволяет отказаться от использования кода метода ExecuteVerb() для этого элемента меню. Однако это не рекомендуется, потому что в методе ExecuteVerb() удачно сконцентрирован код, связанный с обработкой щелчков на командах контекстного меню. Обработчик события OnDrawItem — более простая (и старая) версия обработчика события OnAdvancedDrawItem. Он реже вызывается и содержит меньше информации, поэтому вместо него рекомендуется использовать обработчик события OnAdvancedDrawItem. Обработчик события OnMeasureItem может изменять размеры команды в контекстном меню. Вот как выглядит определение этого обработчика в определении класса.

```
protected:
   virtual void
      __fastcall MeasureMenuItem1(System::TObject* Sender,
                                  Graphics::TCanvas* ACanvas,
                                  int& Width,
                                  int& Height);
```

А код его реализации будет выглядеть следующим образом.

```
virtual void __fastcall
   TMyCustomEditor::MeasureMenuItem1(System::TObject* Sender,
                                     Graphics::TCanvas* ACanvas,
                                     int& Width,
                                     int& Height)
{
   Width = x - Height; // Здесь x – нужная ширина
                       // минус высота Height, которая
                       // позволяет разместить стрелку для
                       // указания подчиненных команд меню.
   Height = y;         // y – нужная высота.
}
```

Для организации вызова этого события нужно вставить следующую строку в соответствующий раздел кода метода PrepareItem() в листинге 10.35.

```
MenuItem->OnMeasureItem = MeasureMenuItem1;
```

Следует напомнить, что изменение ширины Width повлияет на размер команды меню только в том случае, если элемент меню шире всех остальных команд контекстного меню. Иначе говоря, текущая ширина контекстного меню будет равна ширине самого широкого элемента. Учтите, что указываемое значение для Width (обычно при рисовании изображения внутри элемента меню) будет задано после *вычитания* принятой по умолчанию высоты. Как показано на рис. 10.9, это нужно для размещения стрелки, указующей на подчиненные команды меню для элемента меню Flip Children.

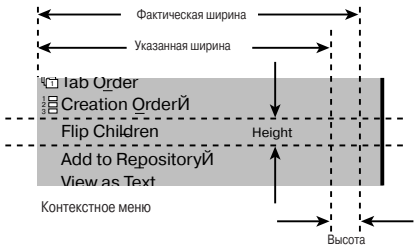


Рис. 10.9. Фрагмент контекстного меню `TImageComponentEditor` с указанием высоты и ширины

Ширина стрелки, указующей на подчиненные команды меню, равна принятой по умолчанию высоте `Height` элемента меню Flip Children. Это значение прибавляется к любому заданному значению ширины `Width`, так что это нужно учесть для предотвращения некорректного отображения правой части вашего элемента контекстного меню. Модификация параметра `Height` всегда влияет на высоту элемента меню, а указание для него значения `0` приведет к его исчезновению.

Пользовательские обработчики событий элемента меню нужны для применения пользовательского способа отображения этого элемента меню. Новые возможности для этого открываются благодаря применению события `OnAdvancedDrawItem`. Переменная `TOwnerDrawState` предоставляет достаточно информации о текущем состоянии элемента меню. Например, если команда меню выбрана, переменная `State` будет содержать значение `odSelected`, что позволяет разместить следующий код в обработчике события.

```
if(State.Contains(odSelected))
{
   // Рисование элемента управления с фоном clRed.
}
else
{
   // Рисование элемента управления с фоном clBtnFace.
}
```

Напомним, что, присваивая обработчик событию `OnAdvancedDrawItem` или `OnDrawItem`, разработчик полностью несет ответственность за весь процесс рисования, включая отображение текста в канве. Для этого следует использовать метод `TextRect` класса `TCanvas`. Более подробную информацию по этому поводу можно получить в разделе об использовании изображений в редакторах свойств в этой главе или интерактивной справке C++Builder. Пользовательский редактор компонента (`TImageComponentEditor`), который обрабатывает события `OnAdvancedDrawItem` и `OnMeasureItem`, для класса `TImage` показан на рис. 10.10. Редактор компонента также реализует методы копирования в буфер обмена и вставки.

Элементы меню можно настраивать как угодно. Например, в команде меню можно в виде изображения разместить логотип компании или увеличить пользовательские элементы меню с использованием более привлекательного фона, чем фон в стандартных элементах меню IDE-среды. Команды меню, которые выполняют редко используемые действия, следует размещать *после* часто используемых команд. В противном случае пользователю вряд ли понравится долго искать нужную команду меню, особенно ту, которая выполняет наиболее нужные ему действия. Именно поэтому наиболее часто используемые элементы меню рекомендуется располагать первыми.

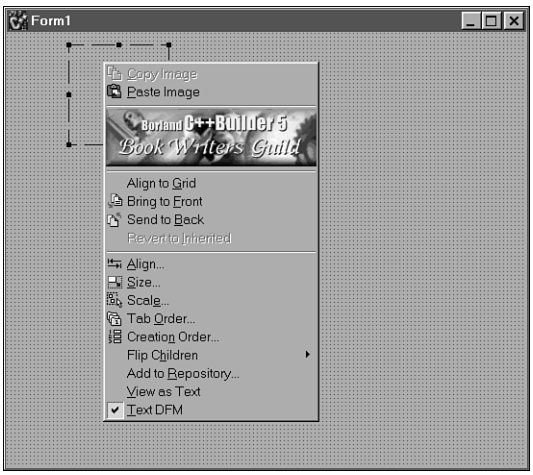


Рис. 10.10. Контекстное меню редактора компонента `TImageComponentEditor`

Создание подчиненных команд меню

При выполнении приложения для команды контекстного меню можно добавить подчиненные команды (которые также являются элементами меню типа `TMenuItem`) с помощью метода `Add()`. Обычно добавляется сразу несколько команд, а метод `Add()` перегружается для обработки массива элементов `TMenuItem` так же, как и для одного элемента `TMenuItem`. Поскольку вновь добавленные подчиненные элементы меню имеют тип `TMenuItem`, то они функционируют, как класс `MenuItem`, и могут быть настроены аналогичным образом. Рассмотрим код добавления подчиненных команд для второй команды меню (напомним, что нумерация начинается с нуля). Число добавленных команд является произвольным. Это может быть переменная типа `static const` класса редактора компонента. Для большей наглядности в последующих фрагментах кода она называется `NoOfSubMenusForItem1`.

Сначала подчиненные элементы меню должны быть объявлены. Если нужно добавить сразу несколько подчиненных команд меню (как в приведенном примере), то проще всего объявить их в виде массива указателей `TMenuItem`. Доступ к подчиненным командам меню необходимо предоставить только для класса редактора компонента, поэтому объявим массив указателей как закрытую переменную.

```
TMenuItem* SubMenuItemsFor1[NoOfSubMenusForItem1];
```

Затем нужно создать подчиненные элементы меню. Лучше всего это сделать в конструкторе редактора компонента. В рассматриваемом примере этот конструктор пуст и встроен в код. Его необходимо изменить в определении и реализации класса с помощью следующего кода.

```
// В файле "MyCustomEditor.h" измените объявление конструктора
// на следующее объявление и удалите директивы
// #pragma option push и #pragma option pop

__fastcall virtual
  TCustomComponentEditor(Classes::TComponent* AComponent,
                         _di_IFormDesigner ADesigner);
```

```
// Код конструктора
__fastcall TCustomComponentEditor::
   TCustomComponentEditor(Classes::TComponent* AComponent,
                          _di_IFormDesigner ADesigner)
   : TComponentEditor(AComponent, ADesigner)
{
   for(int i=0; i<NoOfSubMenusForItem1; ++i)
   {
      SubMenuItemsFor1[i] == new TMenuItem(Application);
      SubMenuItemsFor1[i]->Caption.sprintf("Sub-Menu %d", i);
      // Другие операции инициализации
   }
   // Другие операции инициализации
}
```

Если подчиненные команды меню создаются в конструкторе редактора компонента, то удаляться они должны в деструкторе редактора компонента. В данный момент деструктор тоже пуст и встроен в код, а потому его следует изменить аналогично конструктору.

```
// В файле "MyCustomEditor.h" измените объявление деструктора
// на следующее объявление и удалите директивы
// #pragma option push и #pragma option pop

__fastcall virtual ~TCustomComponentEditor(void);

// Код деструктора
__fastcall TCustomComponentEditor::~TCustomComponentEditor(void)
{
   for(int i=0; i<NoOfSubMenusForItem1; ++i)
   {
      delete SubMenuItemsFor1[i];
   }
}
```

Теперь приступим к созданию кода добавления подчиненных команд меню для команды с индексом 1. Для этого в листинге 10.35 с кодом метода `PrepareItem()` нужно после получения непостоянного указателя `MenuItem` добавить следующую строку.

```
MenuItem->Add(SubMenuItemsFor1, NoOfSubMenuItemsFor1-1);
```

Далее подчиненные команды меню могут использоваться так же, как и все другие команды контекстного меню.

С особой осторожностью используйте метод `ExecuteVerb()` для подчиненных команд меню, потому что при неправильном употреблении этого метода могут возникнуть непредсказуемые последствия.

Метод `ExecuteVerb()`

Метод `ExecuteVerb()` используется для размещения кода, который должен быть выполнен после щелчка мышью на пользовательской команде меню. Основная структура остается такой же, как и у метода `GetVerb()`; т.е. код находится внутри оператора `switch`.

```
void __fastcall TMyCustomEditor::ExecuteVerb(int Index)
{
```

```
switch(Index)
{
   case 0 : EditComponet();
            break;
   case 1 : break; // Никаких действий - Информация о правах
   case 2 : break; // Никаких действий - Разделительная линия
   case 3 : // Какие-то другие действия ...
            break;
   default : break;
   }
}
```

Так выглядит основная структура метода `ExecuteVerb()`. Обычно после щелчка мышью на элементе меню отображается диалоговое окно, если только этот элемент не является разделительной линией или представлением некоторой информации в текстовом или графическом виде. Для выполнения этого действия нужно разместить соответствующий код. В этом примере после щелчка на первой команде меню на экране должна отображаться форма для редактирования компонента. Это очень удобно и практично для пользователей. Необходимый для этого код идентичен показанному ранее для метода `Edit()`. Если редактор компонента происходит от класса `TComponentEditor`, а метод `Edit()` уже содержит код отображения формы редактора компонента, то не имеет смысла повторять этот код снова. Лучше разместить необходимый код в отдельной функции, в данном случае — в функции `EditComponent()`, а затем вызвать ее в методах `ExecuteVerb()` и `Edit()`. Действительно, если для функции используется первый элемент меню, необходимо только гарантировать вызов кода из метода `ExecuteVerb()`. Дело в том, что `TComponentEditor` уже содержит код метода `Edit()` для выполнения кода, связанного с первым элементом меню. Поэтому метод `Edit()` не нужно переопределять. Сложный код лучше разместить в отдельной функции, независимо от того, дублируется он или нет.

Метод Copy()

Метод `Copy()` используется для копирования объектов с *нестандартным* форматом в буфер обмена, что позволяет предоставить пользователям дополнительные функциональные возможности. Например, этот метод используется для копирования изображения из компонента `TImage` в буфер обмена для последующей вставки его в графический пакет. Необходимый для этого код целиком зависит от типа копируемых данных, что позволяет варьировать реализацию этого метода. Поэтому мы не будем подробно останавливаться на нем. Эти принципы демонстрируются в следующем примере кода, который позволяет копировать изображение из компонента `TImage` в буфер обмена.

```
#include "Clipbrd.hpp"

void __fastcall TImageComponentEditor::Copy(void)
{
   // Этап 1 : Получение указателя на компонент
   TImage* Image = dynamic_cast<TImage*>(Component);

   // Этап 2 : Если это изображение, продолжить работу
   if(Image)
   {
      // Этап 3 : Получение данных в формате, "понятном"
      // буферу обмена.
```

```
      WORD     AFormat;
      unsigned AData;
      HPALETTE APalette;

      Image->Picture->SaveToClipboardFormat(AFormat,
                                            AData, APalette);

      // Этап 4 : Получение указателя на глобальный экземпляр
      // буфера обмена.
      TClipboard* TheClipboard = Clipboard();

      // Этап 5 : Копирование данных в буфер обмена
      TheClipboard->SetAsHandle(AFormat, AData);
   }
}
```

Первый этап достаточно прост: нужный указатель может быть получен в результате динамического приведения (`dynamic_cast`) типа для указателя `TComponent`, возвращенного свойством `Component` класса `TComponentEditor`. В противном случае копирование не выполняется. На следующих этапах нужно представить данные в формате, "понятном" буферу обмена. Все поддерживаемые форматы перечислены в интерактивной справке (можно зарегистрировать и другие пользовательские форматы, однако эта тема выходит за рамки обсуждения). Сразу после этого следует получить глобальный указатель на буфер обмена с помощью функции `Clipboard()`. При этом *не* следует создавать новый экземпляр класса `TClipboard`. Наконец, данные копируются в буфер обмена. Более простой вариант кода этой функции показан ниже.

```
void __fastcall TImageComponentEditor::Copy(void)
{
   TImage* Image = dynamic_cast<TImage*>(Component);

   if(Image)
   {
      Clipboard()->Assign(Image->Picture);
   }
}
```

Более сложный вариант кода был показан для того, чтобы показать применимость этого способа для выполнения других операций копирования.

Тут важно отметить, что функцию `Copy()` не следует путать с операциями копирования и вставки компонентов в форму в IDE-среде с помощью стандартных команд меню и клавиш быстрого доступа. Эта функция обладает дополнительными возможностями копирования и должна вызываться вручную. Поэтому ее следует располагать в виде элемента контекстного меню компонента. Кроме того, очевидно, потребуется определить и реализовать метод `Paste()`. Код определения этого метода приведен ниже.

```
virtual void __fastcall Paste(void);
```

Соответствующий код реализации этого метода имеет следующий вид.

```
void __fastcall TImageComponentEditor::Paste(void)
{
   TImage* Image = dynamic_cast<TImage*>(Component);
```

```
   if(Image)
   {
      Image->Picture->Assign(Clipboard());
   }
}
```

Регистрация редакторов компонентов

Для регистрации редакторов компонентов используется функция `RegisterComponentEditor()`, которая имеет следующее объявление.

```
extern PACKAGE void __fastcall
   RegisterComponentEditor(TMetaClass* ComponentClass,
                           TMetaClass* ComponentEditor);
```

Она должна вызываться внутри функции `Register()` пакета. Оба ее параметра являются наследниками класса `TObject`, поэтому оператор `__classid` может использоваться для получения указателя `TMetaClass` для каждого из них. Первый параметр содержит класс компонента, для которого регистрируется редактор компонента. Второй параметр содержит класс самого редактора компонента. Например, для регистрации пользовательского компонента `TImage` с его редактором `TImageComponentEditor` нужно применить следующую строку кода.

```
RegisterComponentEditor(__classid(TImage),
                        __classid(TImageComponentEditor));
```

Используются редакторы компонентов аналогично редакторам свойств, т.е. от новых к старым. Вследствие этого становится возможным переопределять существующие редакторы компонентов в пользу пользовательских редакторов компонентов с более широкой функциональностью.

Как и в случае с редакторами свойств, пакеты редакторов компонентов можно регистрировать без самих компонентов. Именно так был зарегистрирован редактор компонентов `TImageComponentEditor`. Он включен в пакет, содержащий усовершенствованные редакторы свойств, которые рассматривались выше в этой главе в разделе об использовании изображений в редакторах свойств.

Использование заданных изображений в пользовательских редакторах свойств и компонентов

Большая часть предыдущего материала имела отношение к настройке графического представления. При этом отмечалось, что внешний вид и практичность редакторов можно в значительной степени повысить с помощью изображений. В этом разделе приводятся подробные рекомендации, которые могут быть использованы не только при создании пользовательских редакторов (свойств или компонентов), но и для проекта в целом.

Прежде чем использовать заданные изображения нужно создать их. Это можно сделать с помощью любого графического пакета, например Image Editor, который входит в состав C++Builder. Главное, чтобы они находились в формате `bmp`. Для указания заданного количества цветов для определенной платформы (например, 8-битовой палитры цветов) можно создать два набора изображений с разным количеством цветов. В современных системах это ус-

ловие не имеет столь большого значения, каким оно было в прежних системах. Созданные изображения их необходимо включить в пакет.

Это можно сделать разными способами. Один из них заключается в использовании инструмента Image Editor среды C++Builder для создания *откомпилированного файла ресурсов* (с расширением `res`). При этом ваши изображения будут содержаться в растровом формате в откомпилированном файле ресурсов. Конечно, Image Editor имеет очень ограниченные возможности, однако с его помощью довольно просто создать откомпилированный файл ресурсов. Следует внимательно указывать *имена ресурсов* для изображений, потому что впоследствии они будут использованы для загрузки ресурса в ваш код. Инструмент Image Editor также позволяет создавать пиктограммы и курсоры и добавлять их в откомпилированный файл ресурсов.

Другой подход заключается в ручном способе создания файла *сценария ресурсов* (с расширением `rc`). Для этого нужно создать новый текстовый файл, выбрав команду меню File⇨New и пиктограмму Text во вкладке New. Вот как выглядит синтаксис указания изображения, используемого в качестве ресурса.

ресурсID `BITMAP` *имя_файла* `.bmp`

Такую строку нужно создать для каждого изображения-ресурса. Идентификатор ресурса *ресурсID* может быть целым числом или уникальной строкой, которая вводится вместо идентификатора *ресурсID*. Если для него необходимо использовать целое число, то для файла ресурсов создается заголовочный файл с директивами `#define`, которые связывают строковые идентификаторы с их целочисленными значениями. Связь между ними устанавливается во время компиляции. В представленных здесь примерах, идентификатор ресурса *ресурсID* будет иметь строковый формат. Ключевое слово `BITMAP` сообщает компилятору о том, что ресурс является растровым изображением, а параметр *имя_файла* указывает имя файла с этим изображением, которое может быть указано с кавычками или без них. Например, файл ресурсов в пакете `EnhancedEditors` на прилагаемом к книге компакт-диске содержит следующие определения ресурсов.

```
RESOURCE_CopyImage BITMAP "CopyImage.bmp"
RESOURCE_PasteImage BITMAP "PasteImage.bmp"
RESOURCE_GreyedPasteImage BITMAP "GrayedPasteImage.bmp"
RESOURCE_GreyedCopyImage BITMAP "GrayedCopyImage.bmp"
RESOURCE_ActiveWritersGuildLogo BITMAP
↳"ActiveWritersGuildLogo.bmp"
RESOURCE_InActiveWritersGuildLogo BITMAP
↳"InActiveWritersGuildLogo.bmp"
```

Префикс `RESOURCE_` используется как метка для более легкого обнаружения ресурсов. Прописные символы также очень часто применяются для облегчения поиска имен ресурсов. Однако слова, полностью состоящие из прописных символов, очень плохо воспринимаются, поэтому для идентификаторов ресурсов принят показанный выше формат. Кроме того, префикс `RES_` можно использовать для ресурсов в откомпилированных файлах ресурсов (например созданных с помощью Image Editor), а префикс `RC_` для ресурсов, упомянутых в файле сценария ресурсов. Это позволяет легко различать оба эти типа идентификаторов ресурсов. В завершение работы со сценарием ресурсов нужно сохранить этот файл с расширением `.rc`. Таким образом, мы получили файл ресурсов, и теперь остается только включить указанные изображения в наш пакет (или проект).

Ручной метод создания файла сценария ресурсов обладает некоторыми преимуществами по сравнению со способом на основе использования инструмента Image Editor. Во-первых, он позволяет редактировать и обновлять отдельные изображения с помощью практически любого графического редактора. Кроме того, таким образом в файл ресурсов можно включить не только растровые изображения, пиктограммы и курсоры. Это позволяет централизованно управлять ресурсами на основе их использования, а не только на основе их типа. Как уже было сказано, воз-

можности инструмента Image Editor очень ограничены. Он не может адекватно обрабатывать цветовые палитры, и просто обидно, что такая прекрасная RAD-среда содержит столь несовершенный редактор и менеджер ресурсов-изображений. Хорошие средства редактирования не так важны по сравнению с поддержкой интеграции вставленных изображений и управления ресурсами. Возможно, в следующем выпуске C++Builder эта часть RAD-среды будет улучшена.

Включение файлов ресурсов в пакеты

После сохранение файла с расширением `.res`, содержащим нужные изображения, или файла с расширением `.rc`, который содержит ссылки на них, можно приступать к включению файлов ресурсов в пакеты. Для этого в окне редактора пакетов, которое отображается на экране выбором команды File⇨New и пиктограммы Package (при создании пакета) или File⇨Open и нужного пакета (при открытии пакета), щелкните на кнопке Add to Package. Затем во вкладке Add Unit найдите нужные файлы с расширениями `.res` или `.rc` и щелкните на кнопке OK. После этого показанная ниже строка кода будет включена между директивами `#pragma hrdstop` и `#pragma package(smart_init)` в исходном файле пакета.

```
USERES("PathnameOfCompiledResource.res"); // Для файлов .res
USERC("PathnameOfResource.rc"); // Для файлов .rc
```

Обычно файл ресурсов располагается в том же каталоге, что и пакет, и необходимо знать только имя файла ресурса. Различие в использовании откомпилированного ресурса и сценария ресурса заключается в том, что откомпилированный ресурс компонуется с файлом пакета с расширением `.bpl` (или выполняемым файлом проекта с расширением `.exe`); а файл сценария ресурсов нужно компилировать вместе с пакетом. Кроме того, откомпилированный файл ресурсов можно включить в пакет с помощью следующей директивы `#pragma resource`.

```
#pragma resource "PathnameOfCompiledResource.res"
```

После включения ресурсов в пакет можно приступать к их использованию.

Применение ресурсов в редакторах свойств и компонентов

В предыдущем разделе предполагалось, что заданные изображения будут использоваться, например, в раскрывающемся списке значений свойства, в командах меню или даже в качестве статических логотипов в редакторах компонентов. Действительно, возможности их использования просто безграничны. В этом разделе мы рассмотрим использование таких изображений.

Включенные в пакет ресурсы сравнительно просто использовать, загружая их в оперативную память в нужном формате. Для отдельных изображений используется класс `TBitmap` (либо класс `TPicture`, который обладает свойством `TBitmap`, либо класс `TImage`, который обладает свойством `TPicture`). Для нескольких изображений одинакового размера следует использовать класс `TImageList`. В классе `TBitmap` предусмотрены следующие методы, предназначенные для загрузки ресурсов-изображений.

```
void __fastcall LoadFromResourceName(int Instance,
                                     const AnsiString ResName);
void __fastcall LoadFromResourceID(int Instance,
                                   int ResID);
```

Первая функция `LoadFromResourceName()` используется для ресурсов, заданных строковыми идентификаторами, а вторая — для ресурсов, заданных целочисленными идентификаторами. В следующих ниже примерах будет использоваться функция

LoadFromResourceName(). Первый параметр, Instance, содержит дескриптор экземпляра для пакета, который содержит данный ресурс. Он содержится в глобальной переменной HInstance, определенной в файле $(BCB)\Include\Vcl\Sysinit.hpp.

```
extern PACKAGE HINSTANCE HInstance;
```

Тип HINSTANCE определен в файле $(BCB)\Include\wtypes.h как void*. Это значит, что перед использованием тип дескриптора HInstance должен быть приведен к типу int с помощью оператора reinterpret_cast.

```
reinterpret_cast<int>(HInstance)
```

Класс TImageList содержит следующие методы загрузки изображений-ресурсов (производные от класса TCustomImageList).

```
bool __fastcall ResourceLoad(TResType ResType,
                             AnsiString Name,
                             Graphics::TColor MaskColor);

bool __fastcall ResInstLoad(int Instance,
                             TResType ResType,
                             System::AnsiString Name,
                             Graphics::TColor MaskColor);
```

В пакете используется только метод ResInstLoad(), поэтому ниже будет обсуждаться именно он, хотя обе эти функции идентичны, за исключением переменной Instance. Первый параметр функции содержит дескриптор HInstance. Второй параметр содержит информацию о типе ресурса: rtBitmap означает растровый рисунок, rtIcon — пиктограмму, а rtCursor — курсор. Цвет маски может быть задан в последнем параметре, который используется только в том случае, если свойство Mask класса TImageList имеет значение true.

Загрузка ресурсов выполняется без особых проблем. Основная задача заключается в определении места загрузки. В листинге 10.36 показан пример реализации обработчика события OnAdvancedDrawItem для третьего элемента пользовательского контекстного меню редактора компонента TImageComponentEditor.

Листинг 10.36. Загрузка и отображение ресурсов-изображений

```
void __fastcall TImageComponentEditor::AdvancedDrawMenuItem3
   (System::TObject* Sender,
    Graphics::TCanvas* ACanvas,
    const Windows::TRect& ARect,
    TOwnerDrawState State)
{
   std::auto_ptr<Graphics::TBitmap>
      Logo(new Graphics::TBitmap());

   if(State.Contains(odSelected))
   {
      Logo->LoadFromResourceName
         (reinterpret_cast<int>(HInstance),
          "RESOURCE_ActiveWritersGuildLogo");
   }
   else
   {
      Logo->LoadFromResourceName
```

```
        (reinterpret_cast<int>(HInstance),
         "RESOURCE_InActiveWritersGuildLogo");
   }
   // Рисование логотипа в канве
   ACanvas->Draw(ARect.Left, ARect.Top, Logo);
}
```

Код в листинге 10.36 проверяет состояние элемента меню, и если он выбран, отображается один рисунок, в противном случае — другой. Этот код вполне удовлетворительно работает, но он недостаточно эффективен. Дело в том, что логотип `Logo` создается каждый раз при перемещении мыши над элементом меню `MenuItem3`, затем в него загружается ресурс `RESOURCE_ActiveWritersGuildLogo`, а после отображения этого рисунка-ресурса логотип `Logo` удаляется. После вывода указателя мыши за пределы элемента меню `MenuItem3` этот процесс повторяется, за исключением того, что загружается рисунок-ресурс `RESOURCE_InActiveWritersGuildLogo`.

Однако существует более эффективный подход: ресурсы, используемые редактором компонентов, загружаются в объекты `TBitmap` или объекты `TImageList` при конструировании редактора компонента. В этом примере функции потребуется в конструкторе `TImageComponentEditor` создать объекты `TBitmap` (и удалить их в его деструкторе) и загрузить в них ресурсы. Указатели на эти рисунки являются закрытыми переменными. Соответствующий код показан в листинге 10.37.

Листинг 10.37. Улучшенный код управления ресурсами-изображениями

```
// СНАЧАЛА ОПРЕДЕЛЕНИЕ КЛАССА

#include "dsgnintf.hpp"

class TImageComponentEditor : public TDefaultEditor
{
   typedef TComponentEditor inherited;

private:
   virtual void __fastcall
      AdvancedDrawMenuItem3(System::TObject* Sender,
                            Graphics::TCanvas* ACanvas,
                            const Windows::TRect& ARect,
                            TOwnerDrawState State);

   virtual void __fastcall
      MeasureMenuItem3(System::TObject* Sender,
                       Graphics::TCanvas* ACanvas,
                       int& Width,
                       int& Height);

   // Закрытые данные
   Graphics::TBitmap* ActiveWritersGuildLogo;
   Graphics::TBitmap* InActiveWritersGuildLogo;

public:
   // По щелчку правой кнопкой мыши
   // КОНТЕКСТНОЕ МЕНЮ - Этап 1
```

```
   virtual int __fastcall GetVerbCount(void);
   //                  - Этап 2
   virtual AnsiString __fastcall GetVerb(int Index);
   //                  - Этап 3
   virtual void __fastcall
      PrepareItem(int Index,
                  const Menus::TMenuItem* AItem);
   //                  - Этап 4
   virtual void __fastcall ExecuteVerb(int Index);

   // Копирование рисунка в буфер обмена
   virtual void __fastcall Copy(void);

   // Вставка рисунка из буфера обмена
   virtual void __fastcall Paste(void);

public:
   __fastcall virtual
      TImageComponentEditor(Classes::TComponent* AComponent,
                            _di_IFormDesigner ADesigner);

public:
   __fastcall virtual ~TImageComponentEditor(void);
};

// ТЕПЕРЬ КОД РЕАЛИЗАЦИИ
//-------------------------------------------------------------//
//                        КОНСТРУКТОР                          //
//-------------------------------------------------------------//
__fastcall TImageComponentEditor::TImageComponentEditor
   (Classes::TComponent* AComponent,
   _di_IFormDesigner ADesigner)
   : TDefaultEditor(AComponent, ADesigner)
{
   ActiveWritersGuildLogo = new Graphics::TBitmap();
   InActiveWritersGuildLogo = new Graphics::TBitmap();

   ActiveWritersGuildLogo->LoadFromResourceName
                           (reinterpret_cast<int>(HInstance),
                           "RESOURCE_ActiveWritersGuildLogo");

   InActiveWritersGuildLogo->LoadFromResourceName
                           (reinterpret_cast<int>(HInstance),
                           "RESOURCE_InActiveWritersGuildLogo");
}
//-------------------------------------------------------------//
//                          ДЕСТРУКТОР                         //
//-------------------------------------------------------------//
__fastcall TImageComponentEditor::~TImageComponentEditor(void)
{
   delete ActiveWritersGuildLogo;
   delete InActiveWritersGuildLogo;
```

```
}
//---------------------------------------------------------------//
void __fastcall TImageComponentEditor::AdvancedDrawMenuItem3
   (System::TObject* Sender,
    Graphics::TCanvas* ACanvas,
    const Windows::TRect& ARect,
    TOwnerDrawState State)
{
   if(State.Contains(odSelected))
   {
      ACanvas->Draw(ARect.Left, ARect.Top,
                    ActiveWritersGuildLogo);
   }
   else
   {
      ACanvas->Draw(ARect.Left, ARect.Top,
                    InActiveWritersGuildLogo);
   }
}
//---------------------------------------------------------------//
```

Нетрудно заметить, что новая реализация `OnAdvancedDrawItem` выглядит гораздо проще и содержит только код, необходимый для рисования изображения.

Заключительное замечание: ресурсы можно загружать в объект `TImageList` с помощью цикла. Для этого в конце имен связанных изображений нужно указать их порядковый номер. Допустим, что у нас есть 17 изображений, которые нужно разместить в списке. Тогда их имена могут иметь следующий вид: `RESOURCE_Image`*`XX`*, где *`XX`* — числа от `01` до `17`. А код в целом будет иметь следующий вид.

```
// В определении класса -------------------------------------//
TImageList* ImageList;
//---------------------------------------------------------------//

// В коде конструктора --------------------------------------//
ImageList = new TImageList(this);
ImageList->Masked = false;

for(int i=0; i<17; ++i)
{
   ImageList->ResInstLoad(reinterpret_cast<int>(HInstance),
      rtBitmap,
      AnsiString("RESOURCE_Image").cat_sprintf("%.2d",i+1),
      clWhite);
}
//---------------------------------------------------------------//

// В деструкторе --------------------------------------------//
delete ImageList;
//---------------------------------------------------------------//
```

Как видите, работа с ресурсами не так уж сложна. Зато вы можете сделать внешний вид своих редакторов и проектов более привлекательным и профессиональным.

Регистрация категорий свойств в пользовательских компонентах

Категорией называется класс, производный от класса `TPropertyCategory`. В этом разделе рассматриваются способы использования категорий в пользовательских компонентах. При этом особое внимание следует уделить следующим вопросам. Во-первых, нужно определить категории, используемые компонентом. Если они отличаются от 13 заданных категорий (табл. 10.13), то следует создать классы пользовательских категорий. Во-вторых, нужно зарегистрировать свойства (и события) в соответствующих категориях.

Категории и их создание

Имена 13 категорий, установленных в C++Builder, и соответствующие им классы показаны в табл. 10.13.

Таблица10.13. Классы категорий в C++Builder

Название категории	Класс категории
Action (Действие)	`TActionCategory`
Data (Данные)	`TDataCategory`
Database (Баз данных)	`TDatabaseCategory`
Drag, Drop and Docking (Перетаскивание, опускание и закрепление)	`TDragNDropCategory`
Help and Hints (Помощь и подсказки)	`THelpCategory`
Input (Ввод)	`TInputCategory`
Layout (Макет)	`TLayoutCategory`
Legacy (Устаревшие операции)	`TLegacyCategory`
Linkage (Связывание)	`TLinkageCategory`
Locale (Локализация)	`TLocaleCategory`
Localizable (Локализуемость)	`TLocalizableCategory`
Miscellaneous (Дополнительные)	`TMiscellaneousCategory`
Visual (Визуальные)	`TVisualCategory`

Краткое описание каждой категории приводится в разделе о категориях свойств в окне Object Inspector в главе 2. Объявления перечисленных в табл. 10.13 категорий приводятся в файле `$(BCB)\Include\Vcl\DsgnIntf.hpp`.

Для создания категории нужно определить новый класс, который является наследником как класса `TPropertyCategory`, так и одного из его наследников (например, одной из 13 установленных категорий). Дополнительное условие заключается в переопределении методов `Name` и `Description`, которые соответствовали бы новой категории. Строго говоря, рекомендуется переопределить метод `Description`, хотя это и не обязательно. Для определения новой категории нужно настроить соответствующим образом код в листингах 10.38 и 10.39. Параметр *`имя_категории`* следует заменить фактическим именем категории, а также указать строковые значения для имени (`Name`) и описания (`Description`) категории.

Листинг 10.38. Код определения новой категории

```
#include <Dsgnintf.hpp>

class PACKAGE Tимя_категории : public TPropertyCategory
{
   typedef TPropertyCategory inherited;

public:
   #pragma option push -w-inl
   virtual AnsiString __fastcall Name()
   {
      return Name(__classid(Tимя_категории));
   }
   #pragma option pop

   static AnsiString __fastcall Name(System::TMetaClass* vmt);

   #pragma option push -w-inl
   virtual AnsiString __fastcall Description()
   {
      return Description(__classid(Tимя_категории));
   }
   #pragma option pop

   static AnsiString __fastcall Description(System::TMetaClass*vmt);

   #pragma option push -w-inl

   // Конструктор
   inline __fastcall T NameOfCategory (void)
      : TPropertyCategory() {}
   #pragma option pop

   #pragma option push -w-inl
   // Деструктор
   inline __fastcall virtual ~T NameOfCategory (void) {}
   #pragma option pop
};
```

Далее нужно переопределить статические члены-функции `Name` и `Description` так, как, например, показано в листинге 10.39. Каждая из них возвращает соответствующую строку. Строка, возвращаемая методом `Name`, используется в окне Object Inspector. Альтернативный вариант реализации методов `Name` и `Description` заключается в загрузке строки из ресурса для того, чтобы разные строки могли использоваться для разных локализованных версий программного продукта.

Листинг 10.39. Код новой категории

```
AnsiString __fastcall
   Tимя_категории::Name(System::TMetaClass* vmt)
{
   return "Имя категории";
```

```
}

AnsiString __fastcall
   Tимя_категории::Description(System::TMetaClass* vmt)
{
   return "Свойства и/или события категории Имя категории";
}
```

Этот код определения методов следует разместить в заголовочном файле, который содержит другие определения, связанные с регистрацией, а код их реализации — в соответствующем файле с расширением `.cpp`. После этого новая категория станет доступной для регистрирующих свойств (и событий). Если категория не является непосредственно производной от класса `TPropertyCategory` (например, унаследована от класса `TInputCategory`), то тип следует переобозначить с указанием унаследованного класса для отражения этого факта, хотя от этого нет никакой реальной выгоды. Гораздо удобнее сделать класс данной категории непосредственным наследником класса `TPropertyCategory`.

Регистрация свойств в категории

В C++Builder предусмотрены две перегружаемые функции, которые предназначены для регистрации одного или нескольких свойств в данной категории: `RegisterPropertyInCategory()` или `RegisterPropertiesInCategory()` (они объявлены в файле `$(BCB)\Include\Vcl\DsgnIntf.hpp`). Эти функции регистрации вызываются в функции `Register()` пакета. На самом деле при этом регистрируется не одно свойство этой категории, а фильтр свойства, который может применяться для нескольких свойств. В этом смысле имена, выбранные для регистрирующих функций, могут внести дополнительную путаницу. Для регистрации одного фильтра свойства в некоторой категории можно использовать одну из следующих четырех перегружаемых функций.

```
extern PACKAGE TPropertyFilter* __fastcall
   RegisterPropertyInCategory(TMetaClass* ACategoryClass,
                              const AnsiString APropertyName);

   extern PACKAGE TPropertyFilter* __fastcall
   RegisterPropertyInCategory(TMetaClass* ACategoryClass,
                              TMetaClass* AComponentClass,
                              const AnsiString APropertyName);

   extern PACKAGE TPropertyFilter* __fastcall
   RegisterPropertyInCategory(TMetaClass* ACategoryClass,
                              Typinfo::PTypeInfo APropertyType,
                              const AnsiString APropertyName);

   extern PACKAGE TPropertyFilter* __fastcall
   RegisterPropertyInCategory(TMetaClass* ACategoryClass,
                              Typinfo::PTypeInfo APropertyType);
```

В каждой из перечисленных выше функций используется разный способ указания фильтра свойства, который требуется зарегистрировать в данной категории. Для регистрации в категории сразу нескольких фильтров свойств можно использовать одну из трех перегруженных версий метода `RegisterPropertiesInCategory`.

```
extern PACKAGE TPropertyCategory* __fastcall
   RegisterPropertiesInCategory(TMetaClass* ACategoryClass,
                                const System::TVarRec* AFilters,
                                const int AFilters_Size);

extern PACKAGE TPropertyCategory* __fastcall
   RegisterPropertiesInCategory(TMetaClass* ACategoryClass,
                                TMetaClass* AComponentClass,
                                const AnsiString* AFilters,
                                const int AFilters_Size);

extern PACKAGE TPropertyCategory* __fastcall
   RegisterPropertiesInCategory(TMetaClass* ACategoryClass,
                               Typinfo::PTypeInfo APropertyType,
                               const AnsiString* AFilters,
                               const int AFilters_Size);
```

Как видите, существует семь функций для регистрации фильтров свойств для категории: четыре функции предназначены для регистрации отдельных фильтров свойств и три — для регистрации сразу нескольких фильтров свойств. На основании параметров ввода функция `RegisterPropertyInCategory()` генерирует один фильтр `TPropertyFilter`, а функция `RegisterPropertiesInCategory()` — несколько фильтров `TPropertyFilter`. Впоследствии IDE-среда может использовать список фильтров `TPropertyFilter` для определения тех свойств, которые будут относиться к каждой из заданных категорий. Напомним, что одно свойство может относиться к нескольким категориям. Определение класса `TPropertyFilter` приводится в файле `$(BCB)\Include\Vcl\DsgnIntf.hpp` и показано в листинге 10.40.

Листинг 10.40. Определение класса `TPropertyFilter`

```
class DELPHICLASS TPropertyFilter;

class PASCALIMPLEMENTATION TPropertyFilter :
   public System::TObject
{
   typedef System::TObject inherited;

private:
   Masks::TMask* FMask;
   TMetaClass* FComponentClass;
   Typinfo::TTypeInfo* FPropertyType;

   int FGroup;

public:
   __fastcall TPropertyFilter(const AnsiString APropertyName,
                              TMetaClass* AComponentClass,
                              Typinfo::PTypeInfo APropertyType);

   __fastcall virtual ~TPropertyFilter(void);

   bool __fastcall Match(const AnsiString APropertyName,
                         TMetaClass* AComponentClass,
```

```
                        Typinfo::PTypeInfo APropertyType);

    __property TMetaClass* ComponentClass =
       {read=FComponentClass};
    __property Typinfo::PTypeInfo PropertyType =
       {read=FPropertyType};
};
```

Ниже перечислены три основных члена-данных класса `TPropertyFilter` .

- **Маска имени свойства (`FMask`)**. Если этот член типа `AnsiString` содержит не пустую строку, то на ее основе создается объект `TMask`. Он содержит маску имени свойства, которой должно соответствовать имя, которое предполагается использовать в поле этого свойства. Обратите внимание на специальные символы, которые могут использоваться для создания этой маски (табл. 10.19). В противном случае свойство может иметь любое имя.
- **Класс компонента (`FComponentClass`)**. Этот член содержит указатель `TMetaClass*`, указывающий класс того компонента, в состав которого входит свойство. Если задано значение `0`, то может использоваться класс любого компонента.
- **Тип свойства (`FPropertyType`).** Этот член содержит информацию о типе `PTypeInfo` (`TTypeInfo*`) свойства. Если задано значение `0`, то тип не имеет значения.

В табл. 10.14 перечислены специальные символы, которые могут содержаться в имени свойства для генерации нужной маски свойства.

Таблица 10.14. Специальные символы, используемые для создания имени свойства в классе `TPropertyFilter`

Символ	Назначение
`*`	Используется как символ подстановки для любого количества символов
`?`	Используется как символ подстановки для любого *одного* символа
`[Набор_Диапазон]`	Используется для задания набора или диапазона (или и того, и другого) символов, которым должен соответствовать данный символ. Например, `[AbcDE0-9]` включает символы `AbcDE0123456789`
`[!Набор_Диапазон]`	Используется для задания набора или диапазона (или и того, и другого) символов, которым *не* должен соответствовать данный символ. Например, `[!AbcDEf-j]` исключает символы `AbcDEfghij`

Под *символом* здесь подразумевается любая буква или цифра без учета регистра.

Множеством называется группа символов, заключенных в квадратные скобки (`[]`). Символы во множестве не разделяются запятыми или пробелами, например `[AbcDE]`.

Диапазоном называется ряд смежных символов, заданных по первому и последнему символу с дефисом (`-`) между ними. Диапазоны также заключаются в квадратные скобки (`[]`). Например, `[0-9]` означает символы `0123456789`.

Примеры использования этих функций регистрации показаны ранее в этой главе в листинге 10.40. В табл. 10.15 перечислены параметры этих функций и рассматривается назначение каждой из них. За исключением `ACategoryClass`, все эти параметры используются для указания одного или нескольких фильтров `TPropertyFilter`, которые относятся к данной категории свойств и определены в `ACategoryClass`.

Таблица 10.15. Параметры функций `RegisterPropertyInCategory()` и `RegisterPropertiesInCategory()`

Параметр	Назначение
`TMetaClass* ACategoryClass`	Используется для указания категории, в которой следует зарегистрировать свойство. Используйте оператор `__classid` для получения указателя `TMetaClass` на класс категории. Пример: `__classid(TMyCategory)`
`TMetaClass* AComponentClass`	Используется для генерации фильтра свойства. Он предназначен для указания компонента, к которому должно относиться свойство, регистрируемое в данной категории. Оператор `__classid` применяется для получения указателя `TMetaClass*` на заданный класс компонента. Пример: `__classid(TMyComponent)`
`const AnsiString PropertyName`	Используется для генерации фильтра свойства. Он предназначен для указания маски, которой должно соответствовать имя свойства, регистрируемого в данной категории. Например, задавая маску `"Shape"`, мы тем самым ограничиваемся свойствами под названием `Shape`, а задавая маску `"OnMouse*"`, — свойствами (возможно событиями), которые начинаются с символов `OnMouse`

`Typinfo::PTypeInfo APropertyType`	Используется для генерации фильтра свойства. Он предназначен для указания информации о типе `PTypeInfo`, которому должно соответствовать данное свойство, регистрируемое в данной категории. Этот параметр позволяет регистрировать свойства только определенного типа. Например, с помощью аргумента `&IntTypeInfo` можно разрешить регистрацию только свойств типа `int`, где `IntTypeInfo` определяется следующим образом:
	`static TTypeInfo IntTypeInfo;`
	`IntTypeInfo.Name = "int";`
	`IntTypeInfo.Kind = tkInteger;`
	Более подробные сведения можно найти в разделе о регистрации пользовательских редакторов свойств

`const AnsiString*AFilters, const int AFilters_Size`	Используется для генерации фильтра свойства. Он предназначен для указания массива масок для имен свойств (по значению), которым должно соответствовать имя данного свойства, регистрируемое в данной категории. Для этого следует использовать макрос `OPENARRAY`
`const System::TVarRec* Afilters, const int Afilters_Size`	Используется для генерации фильтра свойства и указания массива значений TvarRec (по значению), которым должно соответствовать имя данного свойства для его регистрации в данной категории. Для этого следует использовать макрос ARRAYOFCONST. Значения должны иметь тип AnsiString (для указания маски для имени свойства), TMetaClass* (для указания класса компонента, к которому должно относиться данное свойство), или тип PTypeInfo (для указания типа данного свойства)

Как показано в табл. 10.15, функция RegisterPropertiesInCategory() должна принимать в качестве параметра по значению либо постоянный массив значений TVarRec ((BCB)\Include\Vcl\Systvar.h), либо постоянный массив значений AnsiString.

Для постоянного массива значений TVarRec используется макрос ARRAYOFCONST(значения), который содержит следующее определение в файле (BCB)\Include\Vcl\Sysopen.h).

```
OpenArray<TVarRec>значения, OpenArrayCount<TVarRec>значения.GetHigh()
```

Это позволяет передавать массив значений TVarRec по значению в функции регистрации. Поэтому параметры с указателем и размером массива рассматриваются вместе. В этой связи существует еще одна проблема, которая заключается в том, что OpenArray<TVarRec> и OpenArrayCount<TVarRec> ограничены 19 аргументами. Следовательно, для регистрации 20 и более свойств или событий в одной категории нужно несколько раз вызывать функцию RegisterPropertiesInCategory().

Для постоянного массива значений AnsiString используется макрос OPENARRAY(тип, значения) со следующим определением.

```
OpenArray<тип>значения, OpenArrayCount<тип>значения.GetHigh()
```

Например, для значений типа AnsiString макрос OPENARRAY будет выглядеть следующим образом.

```
OPENARRAY(AnsiString, "Значение1", "Значение2", "Значение3");
```

В этом примере используется три аргумента типа AnsiString, а всего в этом макросе допускается не более 19 аргументов.

В табл. 10.16 показаны фильтры, создаваемые каждой из четырех перегруженных версий функции RegisterPropertyInCategory().

Таблица 10.16. Фильтры свойств TPropertyFilter, сгенерированные функцией RegisterPropertyInCategory()

Список параметров перегруженной функции	Сгенерированный фильтр свойства (маска, компонент, тип)
(TMetaClass* ACategoryClass, const AnsiString APropertyName)	(APropertyName, 0, 0)
(TMetaClass* ACategoryClass, TMetaClass* AComponentClass, const AnsiString APropertyName)	(APropertyName, AcomponentClass, 0)
(TMetaClass* ACategoryClass, Typinfo::PTypeInfo APropertyType, const AnsiString APropertyName)	(APropertyName, 0, ApropertyType)
(TMetaClass* ACategoryClass, Typinfo::PTypeInfo ApropertyType)	("", 0, ApropertyType)

В табл. 10.17 показаны фильтры, созданные каждой из трех перегруженных версий функции RegisterPropertiesInCategory().

Таблицы 10.16 и 10.17 следует использовать как руководство при выборе перегруженной версии этих функций для регистрации фильтров свойств. Фильтры свойств описываются на

основе маски имени свойства (Mask), класса того компонента, к которому свойство относится (Component), а также типа свойства (Type).

Таблица 10.17. Фильтры свойств TPropertyFilter, сгенерированные функцией RegisterPropertiesInCategory()

Список параметров перегруженной функции	Сгенерированный фильтр свойства (маска, компонент, тип)
(TMetaClass* ACategoryClass, const System::TVarRec* AFilters, const int AFilters_Size)	Если Afilters[i] имеет тип AnsiString, то (AFilters[i],0,0) Если AFilters[i] имеет тип TMetaClass*, то ("",AFilters[i],0) Если AFilters[i] имеет тип PTypeInfo, то ("",0,AFilters[i])
(TmetaClass* AcategoryClass, TMetaClass* AcomponentClass, const AnsiString* AFilters, const int Afilters_Size)	(AFilters[i], AcomponentClass,0)
(TmetaClass* ACategoryClass, Typinfo::PTypeInfo APropertyType, const AnsiString* AFilters, const int AFilters_Size)	(AFilters[i],0, APropertyType)

Особое внимание в табл. 10.17 следует уделить первой версии перегруженной функции RegisterPropertiesInCategory(). Ее фильтры свойств зависят от типа аргументов, используемых в массиве TVarRec. Обратите внимание также на то, что в одном массиве могут использоваться разные типы, что позволяет очень гибко использовать эту функцию. В листинге 10.41 показан пример кода с использованием этих функций регистрации. В нем предполагается, что используемые категории уже были определены ранее.

Листинг 10.41. Регистрация свойств в категориях в C++Builder

```
namespace Nameoffilecontainingthisregistration
{
   void __fastcall PACKAGE Register()
   {
      // 1 - Регистрация одного фильтра свойств
      //     для категории TMouseCategory.
      //     Фильтр имеет вид ("OnMouse*",0,0), т.е. включает
      //     все свойства (возможно, события), имена которых
      //     начинаются с символов "OnMouse".
      // Применение:
      //     RegisterPropertyInCategory
      //        (TMetaClass* ACategoryClass,
      //         const AnsiString APropertyName);

      RegisterPropertyInCategory(__classid(TMouseCategory),
```

```
                                "OnMouse*");

      // 2 - Регистрация двух фильтров свойств
      //     для категории TMouseCategory.
      //     Первый фильтр имеет вид ("",0,CursoTypeInfo),
      //     т.е. включает свойства типа TCursor.
      //     Второй фильтр имеет вид ("OnMouse*",0,0).
      // Применение:
      //     RegisterPropertiesInCategory
      //        (TMetaClass* ACategoryClass,
      //         const System::TVarRec* AFilters,
      //         const int AFilters_Size);

      PTypeInfo CursorTypeInfo
         = *Typinfo::GetPropInfo(__typeinfo(TForm),
                                 "Cursor")->PropType;

      RegisterPropertiesInCategory
         (__classid(TMouseCategory),
          ARRAYOFCONST( ( CursorTypeInfo,
                          AnsiString("OnMouse*"),
                          AnsiString("EventName2") ) ) );

      // 3 - Регистрация двух фильтров свойств
      //     для категории TMouseCategory.
      //     Первый фильтр имеет вид ("OnClick",0,0),
      //     т.е. включает все свойства (возможно, события),
      //     имена которых начинаются с символов "OnMouse".
      //     Второй фильтр имеет вид ("OnDblClick",0,0).
      //     т.е. включает все свойства (возможно, события)
      //     с именами "OnDblClick".
      // Применение:
      //     RegisterPropertiesInCategory
      //        (TMetaClass* ACategoryClass,
      //         TMetaClass* AComponentClass,
      //         const AnsiString* AFilters,
      //         const int AFilters_Size)

      TMetaClass*AnyComponent = 0;

      RegisterPropertiesInCategory(__classid(TMouseCategory),
                                   AnyComponent,
                                   OPENARRAY(AnsiString,
                                             ("OnClick",
                                              "OnDblClick")));
   }
}
```

В листинге 10.41 показаны не все возможные способы использования и перегружаемые версии двух категорий функций регистрации. Однако в нем проиллюстрированы возможные способы использования списков параметров. Это позволяет выбрать корректный метод вызова каждой функции регистрации. Среди всех параметров класс `AComponentClass` является наименее полезным, потому что подчиненные свойства класса вряд ли могут принадлежать одной категории, если только сам класс не является свойством, так как в этом случае все подчиненные свойства по умолчанию попадают в одну категорию с их классом.

Обратите внимание, что в третьем варианте регистрации передается нулевой (`NULL`) указатель `TMetaClass*` в качестве аргумента для параметра `AComponentClass`. Это значит, что только маски для имен свойств являются частями фильтра свойств. Это нулевое значение нельзя передавать непосредственно, так как при этом может возникнуть двузначная ситуация в отношении используемой версии функции `RegisterPropertiesInCategory()`. Например, литерал `0` в равной степени может быть значением параметра `PTypeInfo`. Аналогичный (показанный ниже) способ может использоваться для передачи нулевого (`NULL`) значения типа `PTypeInfo`.

```
PTypeInfo AnyPropertyType = 0;
```

Резюме

В этой главе представлены следующие основные концепции и технологии, связанные с интерфейсом времени создания компонента.

- Процесс создания пользовательских редакторов свойств подробно описывается на протяжении всей этой главы. В ней показано, что многие классы редакторов свойств уже определены в библиотеке VCL, а оптимальный выбор базового класса для пользовательских редакторов свойств может существенно сэкономить ресурсы, которые обычно требуется затратить на их кодирование. Большая часть приведенного здесь материала посвящена описанию способов переопределения виртуальных (и динамических) методов класса `TPropertyEditor`. Особое внимание уделялось новым методам, которые были введены в C++Builder 5. При этом особое внимание уделялось обработке исключительных ситуаций, связанных с вводом пользователем неверного значения свойства.
- Б*о*льшая часть этой главы посвящена описанию процесса создания пользовательских редакторов компонентов. В отличие от пользовательских редакторов свойств, для создания пользовательских редакторов компонентов используются только два базовых класса: `TComponentEditor` и `TDefaultEditor`. Выбор одного из них зависит от требуемого поведения после двойного щелчка кнопкой мыши. Для генерации обработчика события в качестве базового класса следует использовать класс `TDefaultEditor`, а в других случаях — класс `TComponentEditor`. Также подробно в этой главе описываются способы переопределения виртуальных методов класса `TComponentEditor`. При этом особое внимание уделяется методу `PrepareItem()`, который впервые появился в C++Builder 5 и позволяет разработчикам полностью настраивать контекстное меню пользовательского редактора компонента.
- Как для редакторов свойств, так и для редакторов компонентов показано, что виртуальный метод `Edit()` может использоваться для отображения формы, предназначенной для редактирования этого свойства или компонента. При этом допускаются два

режима работы с такой формой: с мгновенным обновлением изменений, вносимых в свойство или компонент, или только после закрытия этой формы.

- Обсуждаются вопросы использования изображений для улучшения интерфейса редактора и предлагаются подробные рекомендации по их оптимальному применению, чем довольно часто пренебрегают авторы других книг. Материал этого раздела имеет большое значение и для проекта C++Builder в целом.
- Подробно описываются методы корректной регистрации редакторов свойств и компонентов. В частности, детально рассматривается получение информации о типе для небиблиотечных типов.
- Также подробно рассматриваются категории свойств, которые впервые появились в C++Builder версии 5. Показаны способы создания пользовательских категорий свойств и методы регистрации фильтров свойств, которые позволяют IDE-среде определить нужную категорию для каждого свойства.

После прочтения этой главы читатель сможет самостоятельно решать подобные задачи. В частности, разработчики, имеющие большой опыт работы с особо трудными задачами, по достоинству оценят предусмотренные в библиотеке VCL возможности получения информации о типе.

В целом создание редакторов свойств и компонентов представляет собой очень сложный и запутанный процесс, а многие используемые для этого методы могут оказаться сложными и малопонятными даже опытным разработчикам. Автор надеется, что ему удалось прояснить хотя бы некоторые сложные вопросы этой большой и непростой темы.

Глава

11

Другие методы настройки компонентов

Джэйми Оллсоп
Дэймон Чандлер
Малькольм Смит

Эта глава посвящена созданию компонентов. Она не такая общая, как глава 9, посвященная созданию пользовательских компонентов, и не такая специальная, как глава 10, в которой обсуждается создание редакторов свойств и компонентов. В ней рассматриваются различные темы, связанные с созданием компонентов. Начинается она с описания наиболее часто встречающихся при создании компонентов проблем. В ней обсуждаются функции обратного вызова (callback functions), которые применяются в компонентах. Затем рассматриваются фрэймы — важная новинка C++Builder, пополнившая арсенал инструментов разработчика. Приведенные в этой главе сведения помогут читателю понять основные принципы работы с фрэймами.

Заключительный раздел этой главы связан с распространением компонентов. Эта чрезвычайно сложная и запутанная тема обсуждается здесь очень подробно в нескольких частях, посвященных отдельным вопросам. Этот раздел, несомненно, будет очень полезен для всех разработчиков, которым предстоит заниматься распространением компонентов.

Дополнительные вопросы настройки пользовательских компонентов

В этом разделе рассматриваются проблемы, которые встречаются при создании пользовательских компонентов. Вполне возможно, что большинство нынешних создателей компонентов сталкивались с некоторыми из них в своей работе, а будущие разработчики, вероятно, не раз встретятся с ними в будущем. Изучив их сейчас, вам не придется впоследствии тратить время на поиски их решения.

Отображение опубликованных свойств свойства-класса в окне Object Inspector

Если свойство компонента является классом и список его опубликованных свойств требуется расширить, то в качестве базового класса следует выбрать `TPersistent`. В этом случае класс редактора свойства `TClassProperty` будет использован для редактирования свойства класса, что позволяет расширять и редактировать список опубликованных свойств свойства-класса. Для использования в определении компонента свойства, производного от класса `TPersistent`, компилятор должен получить указание о том, что этот класс используется в стиле класса библиотеки VCL. Это значит, что в предварительном объявлении этого класса нужно применить макрос `DELPHICLASS` (т.е. `__declspec(delphiclass,package)`), чтобы сообщить компилятору о том, что это класс библиотеки VCL. В противном случае будет получено сообщение об ошибке.

```
class DELPHICLASS TPersitentDerivedClass; // В предварительном
                                          // объявлении
```

Если компилятор обнаружит определение класса до его использования в качестве свойства, то такая предосторожность не потребуется. Более подробно этот вопрос на конкретном примере рассматривается в обзоре библиотеки VCL в главе 8. Вообще, любое предварительное объявление класса в стиле библиотеки VCL должно содержать макрос `DELPHICLASS`, как например в объявлении класса `TFont`, который является примером класса свойства, производного от класса `TPersistent`.

Использование пространств имен в списках параметров событий

Следует отметить, что указатель пространства имен нельзя использовать в списках параметров событий непосредственно. Это может привести к созданию практически бесполезных событий. Рассмотрим следующее объявление события.

```
typedef void __fastcall
   (__closure *TNotifyFrameAvailable)(System::TObject* Sender,
                                      Graphics::TBitmap* Frame);
```

Объявленное таким образом событие должно возникать в момент доступности нового фрэйма изображения для некоторого компонента, например для компонента захвата изображений. Для использования этого события необходимо создать переменную для хранения значения события, свойство для типа события (обычно опубликованное `__published`), а также функцию запуска события (обычно защищенную).

```
// Переменная для хранения значения события (в разделе private)
private:
TNotifyFrameAvailable FOnFrameAvailable;

// Свойство для типа события (в разделе __published)
public:
__published:
   __property TNotifyFrameAvailable OnFrameAvailable
      = {read=FOnFrameAvailable,
         write=FOnFrameAvailable};

// Функция инициации события в коде реализации
void __fastcall
   ComponentName::FrameAvailable(Graphics::TBitmap* Frame)
{
   if(FOnFrameAvailable != 0) FOnFrameAvaliable(this, Frame);
}
```

На первый взгляд все выглядит прекрасно, но на самом деле этот код нельзя использовать в таком виде. Если компонент с таким кодом инсталлировать в IDE-среде и дважды щелкнуть, инициируя событие `OnFrameAvailable`, то IDE-среда генерирует следующий обработчик событий (для формы `Form1` и компонента `Component1`).

```
void __fastcall
   TForm1::Component1OnFrameAvailable(TObject* Sender,
                                      TBitmap* Frame)
{
   // Код обработчика события
}
```

Возникшая проблема теперь очевидна; она связана с тем, что указатели пространства имен отсутствуют в списке параметров события. Для параметра `TObject* Sender` это совсем не проблема и код будет откомпилирован без ошибок. Однако проблема возникнет с классом `TBitmap`, так как в этом случае возникает двусмысленная ситуация, и компилятор выдаст сообщение об ошибке. Класс `TBitmap` может быть упомянут как `Graphics::TBitmap` (что нам и нужно) или `Windows::TBitmap` (что не нужно). Модификация списка параметров в данном случае невозможна, так как это приведет к получению несовместимого списка параметров. Решение в том, чтобы

избежать использования пространств имен в списке параметров события. Тем не менее в данном случае можно воспользоваться следующей уловкой. Вместо Graphics::TBitmap в качестве типа для указателя мы применим такую модификацию этого типа.

```
typedef Graphics::TBitmap GraphicsTBitmap;
```

В таком случае объявление события будет выглядеть следующим образом.

```
typedef void __fastcall
   (__closure *TNotifyFrameAvailable)(System::TObject* Sender,
                                      GraphicsTBitmap* Frame);
```

Если предположить, что аналогичным образом изменена инициация события, то IDE-среда будет генерировать следующий обработчик события.

```
void __fastcall
   TForm1::Component1OnFrameAvailable(TObject* Sender,
                                      GraphicsTBitmap* Frame)
{
   // Код обработчика события
}
```

Этот код будет компилироваться без ошибок. Но модифицированный тип в данном случае должен быть доступен пользователю компонента, поэтому он должен быть глобальным. Следовательно, здесь важно правильно выбрать имя этого типа. Удалив символ ::, можно получить недвусмысленное имя с ясным смысловым значением и без совпадения с другими именами.

Что нужно учесть при создании списка параметров события

События по сути являются указателями на члены-функции. Поэтому, для события следует указать соответствующий список параметров. Обычно это делается с помощью ключевого слова typedef, как показано ниже.

```
typedef void __fastcall (__closure *TNotifyIsDataTransmitted)
   (System::TObject* Sender,
    bool DataTransmitted);
```

Список параметров этого события имеет следующий вид.

```
(System::TObject* Sender, bool DataTransmitted)
```

Напомним, что компилятор фактически воспринимает список параметров в следующем виде.

```
(System::TObject* , bool )
```

При этом имена параметров Sender и DataTransmitted игнорируются. Действительно, имена здесь не имеют никакого значения. Но их использование позволяет сделать список параметров более информативным, поэтому они практически всегда используются в объявлениях событий и функций. В результате этого список параметров (System::TObject* Sender, bool DataReceived) рассматривается компилятором как аналогичный предыдущему. Следовательно, с точки зрения компилятора приведенное ниже объявление события обладает тем же списком параметров, что и предыдущее объявление.

```
typedef void __fastcall
   (__closure *TNotifyIsDataReceived)(System::TObject* Sender,
                                      bool DataReceived);
```

Проблема возникает, когда IDE-среда используется для генерации обработчика события для компонента, уже содержащего такие события. При этом для первого открытого свойства событий используются имена параметров, которые уже имеются в определении компонента. В следующем примере при объявлении двух событий для их параметров IDE-среда будет использовать имена Sender и DataReceived.

```
// Глобальное объявление
typedef void __fastcall (__closure *TNotifyIsDataTransmitted)
   (System::TObject* Sender,
    bool DataTransmitted);

typedef void __fastcall (__closure *TNotifyIsDataReceived)
   (System::TObject* Sender,
    bool DataReceived);

// В определении класса
private:
   TNotifyIsDataTransmitted FOnIsDataTransmitted;
   TNotifyIsDataReceived FOnIsDataReceived;

public:
__published:
   __property TNotifyIsDataReceived OnIsDataReceived
      = {read=FOnIsDataReceived,
         write=FOnIsDataReceived};

   __property TNotifyIsDataTransmitted OnIsDataTransmitted
      = {read=FOnIsDataTransmitted,
         write=FOnIsDataTransmitted};
```

Следовательно, для компонента Component1 в форме Form1 будет генерироваться следующий обработчик события OnIsDataTransmitted.

```
void __fastcall TForm1::Component1OnIsDataTransmittedEvent
   (TObject* Sender, bool DataReceived)
{
   // Код обработчика события
}
```

Изменение порядка этих объявлений приведет к тому, что IDE-среда будет использовать имена параметров Sender и DataTransmitted.

Эту проблему можно решить по-разному, и каждое решение имеет преимущества и недостатки. Одно решение заключается в введении подстановочных параметров, которые обычно реализуются с помощью пустых определений классов.

```
class NotUsed1{};

class NotUsed2{};
```

Используя эти пустые классы в объявлениях событий, можно получить следующий результат.

```
typedef void __fastcall (__closure *TNotifyIsDataTransmitted)
   (System::TObject* Sender,
    bool DataTransmitted,
    NotUsed1 Dummy);

typedef void __fastcall (__closure *TNotifyIsDataReceived)
```

```
   (System::TObject* Sender,
    bool DataReceived,
    NotUsed2 Dummy);
```

Теперь при инициации события следует выполнить следующую проверку.

```
if(FOnIsDataTransmitted != NULL)
   FOnIsDataTransmitted(this, true, NotUsed1());

if(FOnIsDataReceived != NULL)
   FOnIsDataReceived(this, true, NotUsed2());
```

Вместо вызова обработчика события мы просто конструируем нужный пустой класс. Конечно, чтобы различить эти объявления, нужно использовать параметр пустого класса только в одном из объявлений событий. Дополнительный параметр используется для поддержания последовательности. Пустые определения классов могут быть использованы повторно в других объявлениях событий с совпадающими типами параметров, поэтому вряд ли стоит объявлять очень большое количество таких классов. Их можно также сделать глобальными, размещая в отдельном файле, для последующего включения в случае необходимости. Преимущество этого подхода заключается в том, что проблема решается в исходном коде, не оставляя ее для пользователя компонента. Достаточно только убедить пользователя в том, что эти параметры используются только для правильного отображения IDE-средой имен параметров.

Второе решение заключается в использовании единого объявления для обоих событий, но с добавлением параметра перечислимого типа `enum` в список параметров. Этот параметр затем может использоваться для указания события-получателя или события-передатчика данных. Например, в качестве такого типа можно использовать следующий перечислимый тип для указания процесса получения или передачи данных.

```
enum TDataCommMode {dcmDataReceived, dcmDataTransmitted};
```

В таком случае понадобится использовать только одно объявление события.

```
typedef void __fastcall (__closure *TDataCommEvent)
   (System::TObject* Sender, TDataCommMode CommMode);
```

В определении класса оно будет выглядеть так, как показано ниже.

```
private:
   TDataCommEvent FOnDataCommEvent;

public:
__published:
   __property TDataCommEvent OnDataCommEvent
      = {read=FOnDataCommEvent,
         write=FOnDataCommEvent};
```

Для запуска события используются два события: одно в случае получения данных, а другое — в случае передачи. Соответствующий код этих двух методов имеет следующий вид.

```
void __fastcall ComponentName::DataReceivied()
{
   if(FOnDataCommEvent != 0)
      FOnDataCommEvent(this, dcmDataReceived);
}

void __fastcall ComponentName::DataTransmitted()
{
```

```
   if(FOnDataCommEvent != 0)
      FOnDataCommEvent(this, dcmDataTransmitted);
}
```

Для определения используемого процесса (получения или передачи данных) пользователь затем может применить в обработчике события параметр `CommMode`. Еще одно решение заключается в создании более общего события, которое имело бы соответствующий смысл для каждого из событий. Альтернативным вариантом объявления по отношению к двум предыдущим является следующее объявление.

```
typedef void __fastcall (__closure *TNotifySendReceiveData)
   (System::TObject* Sender,
    bool Succeeded);
```

В определении класса эти события будут иметь следующий вид.

```
private:
   TNotifySendReceiveData FOnIsDataTransmitted;
   TNotifySendReceiveData FOnIsDataReceived;
public:
__published:
   __property TNotifySendReceiveData OnIsDataReceived
      = {read=FOnIsDataReceived,
         write=FOnIsDataReceived};

   __property TNotifySendReceiveData OnIsDataTransmitted
      = {read=FOnIsDataTransmitted,
         write=FOnIsDataTransmitted};
```

Преимуществом данного способа, по сравнению с двумя предыдущими, является минимальное количество объявлений событий. Однако в некоторых случаях не всегда удается использовать предыдущие два способа, поэтому важно помнить о существовании самого первого способа с применением параметров на основе пустых классов.

При использовании компонентов сторонних разработчиков, в которых возникает проблема использования разных событий с одинаковым списком параметров без возможности доступа к их исходному коду, можно просто вручную изменить имена параметров. Иногда это позволяет повысить читабельность кода. Изменение имени вручную хоть и допускается, но это не удобно при дальнейшем сопровождении кода и не очень практично. Если имя параметра присутствует, то это никак не повлияет на компиляцию кода. Рассмотрим, например, процесс создания следующих компонентов формы: `TEdit`, `TUpDown` и `TLabel` (которые по умолчанию получат имена `Edit1`, `UpDown1` и `Label1`, соответственно). Свяжем компонент `UpDown1` с компонентом `Edit1`. Щелкнем дважды на событии `OnChangeEx` компонента `UpDown1`. IDE-среда генерирует следующее объявление в определении класса формы и пустой обработчик события в коде реализации формы.

```
// В определении класса
void __fastcall UpDown1ChangingEx(TObject *Sender,
                                  bool &AllowChange,
                                  short NewValue,
                                  TUpDownDirection Direction);

// В файле реализации
void __fastcall TForm1::UpDown1ChangingEx(TObject *Sender,
                                          bool &AllowChange,
                                          short NewValue,
```

```
                                        TUpDownDirection
                                           Direction)
{

}
```

Разместите в обработчике события следующий код.

```
void __fastcall TForm1::UpDown1ChangingEx(TObject *Sender,
                                        bool &AllowChange,
                                        short NewValue,
                                        TUpDownDirection
                                           Direction)
{
   Label1->Caption = NewValue; // Автоматическое преобразование
                               // к типу AnsiString.
}
```

Скомпилируем и запустим код. После щелчка на компоненте `UpDown1` надпись `Label1` будет содержать новое значение. Изменим теперь в объявлении имя параметра `NewValue` на новое имя **`Pandas`** и **`Elephants`**, как показано ниже.

```
// В определении класса
void __fastcall UpDown1ChangingEx(TObject *Sender,
                                  bool &AllowChange,
                                  short Pandas,
                                  TUpDownDirection Direction);

// В файле реализации
void __fastcall TForm1::UpDown1ChangingEx(TObject *Sender,
                                        bool &AllowChange,
                                        short Elephants,
                                        TUpDownDirection
                                           Direction)
{
   Label1->Caption = Elephants; // Автоматическое преобразование
                               // к типу AnsiString.
}
```

Убедитесь в том, что в коде реализации обработчика события значение **`NewValue`** также заменено значением **`Elephants`**. Скомпилируйте и запустите код. Он будет работать так же, как и прежде. Компилятору безразлично какие имена используются для параметров, но какие-то конкретные имена обязательно должны быть указаны в объявлении обработчика события. Если их удалить, то будет получено сообщение об ошибке `Incomplete method declaration in class` *`class`* (`Неполное объявление методов класса` *`класс`*). В коде реализации, конечно, можно закомментировать или удалить все ненужные имена.

Переопределение динамических методов

Макрос `DYNAMIC` означает `__declspec(dynamic)` и используется для обозначения функций, которые фактически ведут себя как виртуальные. Боле подробно о различиях между ними можно прочесть в обзоре библиотеки VCL в главе 8. При работе с ними следует помнить главное: динамическая (`DYNAMIC`) функция не может быть переопределена виртуальной функцией, и наоборот. Кроме того, при каждом повторном объявлении динамической функции

в дочернем классе переопределение должно содержать ключевое слово DYNAMIC. Пользователи прежних версий C++Builder могут попасться на эту удочку, так как в новой версии многие функции стали динамическими. При переопределении существующего компонента библиотеки VCL следует проверить заголовочный файл этого компонента, чтобы найти переопределяемые методы, которые являются динамическими. Например, функция обработки события KeyPress класса TEdit является динамической, а не виртуальной (как ошибочно утверждается в руководстве разработчика Developer's Guide, который поставляется вместе с C++Builder). Чтобы проиллюстрировать это, рассмотрим исходный код простого компонента TFilterEdit. Определение этого компонента показано в листинге 11.1, а код его реализации — в листинге 11.2.

Листинг 11.1. Определение компонента TFilterEdit

```
class PACKAGE TFilterEdit : public TEdit
{
private:
   AnsiString FFilter;
   bool FExcludeFilter;

protected:
   DYNAMIC void __fastcall KeyPress(char &Key);

public:
   __fastcall TFilterEdit(Tcomponent *Owner);

__published:
   __property AnsiString Filter = {read=FFilterString,
                                   write=FFilterString};
   __property bool ExcludeFilter = {read=FExcludeFilter,
                                    write=FExcludeFilter,
                                    default=false};
};
```

Листинг 11.2. Код реализации компонента TFilterEdit

```
//---------------------------------------------------------//
__fastcall TFilterEdit::TFilterEdit(TComponent* Owner)
   : TEdit(Owner), FExcludeFilter(false)
{
}
//---------------------------------------------------------//
void __fastcall TFilterEdit::KeyPress(char &Key)
{
   // Проверка того, что Key не является клавишей возврата
   if(Key != '\b')
   {
      // Проверка того, что Key находится в фильтре Filter
      for(int i=1; i<=FFilter.Length(); ++i)
      {
         if(Key == FFilter[i])
         {
```

```
            // Key находится в фильтре Filter
            if(ExcludeFilter)
            {
               Key = 0;
               return;
            }
            else return; // Никаких действий
         }
      } // Конец for

      // Key не находится в фильтре Filter
      if(!ExcludeFilter)Key =0;
   }
}
//---------------------------------------------------------//
```

Обратите внимание на использование макроса `DYNAMIC` при объявлении метода `KeyPress`. Показанный здесь компонент `TFilterEdit` позволяет ввести строку символов типа `AnsiString` в качестве фильтра. Этот фильтр может применяться для исключения символов, представленных в фильтре (`ExcludeFilter = true`), либо их включения (`ExcludeFilter = false`) — вариант принимаемый по умолчанию. Этот компонент находится в пакете `NewAdditionalComponents`, который располагается в каталоге главы 10 на прилагаемом к книге компакт-диске. В вводном разделе главы 10 подробно описан процесс инсталляции этого пакета.

Управление сообщениями в пользовательских компонентах

Вполне возможно, что на каком-то этапе создания пользовательского компонента понадобится организовать обработку сообщений. Примеры такой обработки уже приводились в главе 9. Во многих случаях код обработки сообщений выглядит очень просто. В этом разделе рассматриваются наиболее часто встречающиеся проблемы и подходящие способы их решения. Этот раздел можно рассматривать как справочное пособие по способам обработки сообщений в пользовательских компонентах. Проще всего обработка сообщений организована в визуальных компонентах, т.е. производных от класса `TControl` компонентах (и в первую очередь от классов `TGraphicControl` и `TWinControl`). Обработка сообщений в визуальных компонентах, т.е. производных от класса `TComponent`, организована немного сложнее и также рассматривается в этом разделе.

На заметку

Разработчик может создавать собственные сообщения, которые могут передаваться вашим компонентам. Для этого нужно использовать директиву `#define` и присвоить заданному сообщению значение `WM_USER` и некоторое постоянное значение. Использование значения `WM_USER` гарантирует, что созданное вами сообщение не будет конфликтовать с уже имеющимися системными сообщениями. Значение `WM_USER + постоянная` не должно быть больше, чем `0x7FFF`. Кроме того, в имени сообщения следует использовать специальный информативный префикс, например `UM` (User Message, т.е. сообщение пользователя). Вот как выглядит типичное определение пользовательского сообщения.

```
#define UM_MYMESSAGE (WM_USER +1);
```

Второе слагаемое выбирается произвольно. Достаточно только убедиться в том, что нет другого сообщения с таким же значением.

Управление сообщениями в визуальных компонентах можно организовать либо на основе карты сообщений, либо переопределяя метод WndProc() компонента. Сначала рассмотрим способ на основе карты сообщений.

Применение карты сообщений для управления сообщениями в визуальных компонентах

Для использования карты сообщений нужно выполнить два действия.

1. Создать карту сообщений в определении класса компонента (обычно это делается в разделе защищенных переменны), с указанием макроса VCL_MESSAGE_HANDLER для каждого сообщения.
2. Создать метод-обработчик для каждого сообщения. Он не возвращает значения (void) и принимает в качестве единственного параметра ссылку на структуру сообщения.

Создание карты сообщения начинается с начального макроса BEGIN_MESSAGE_MAP.

За этим макросом следует один или несколько макросов VCL_MESSAGE_MAP, по одному для каждого обрабатываемого сообщения. Этот макрос имеет вид VCL_MESSAGE_MAP(msg, type, meth).

Параметр msg содержит фактическое сообщение, type — структуру, используемую библиотекой VCL для представления этого сообщения, а параметр meth — имя метода-обработчика этого сообщения.

На заметку

Макрос VCL_MESSAGE_HANDLER используется вместо макроса MESSAGE_HANDLER. Дело в том, что макрос MESSAGE_HANDLER определен в библиотеке Microsoft Active Template Library (ATL). Если вы не используете библиотеку ATL, то макрос MESSAGE_HANDLER определяется как VCL_MESSAGE_HANDLER, поэтому в случае необходимости можно использовать макрос с более кратким названием.

Например, для обработки сообщения WM_CHAR в методе WMChar() соответствующая строка в карте сообщения будет иметь следующий вид.

```
VCL_MESSAGE_MAP(WM_CHAR, TWMChar, WMChar)
```

Имя структуры TWMChar получается при добавлении символа T к имени сообщения, удалении символа подчеркивания и записи в верхнем регистре только начальных букв каждого слова. Для проверки правильного выбора структуры для представления этого сообщения следует просмотреть содержимое файла $(BCB)\Include\Vcl\Messages.hpp, в котором объявлены структуры сообщений, используемых в библиотеке VCL. Если для сообщения структура не задана специально, то используется структура TMessage.

Структура TMessage определена в файле $(BCB)\Include\Vcl\Messages.hpp следующим образом.

```
struct TMessage
{
   unsigned Msg;
   union
   {
      struct
      {
         Word WParamLo;
         Word WParamHi;
         Word LParamLo;
```

```
         Word LParamHi;
         Word ResultLo;
         Word ResultHi;
      };
      struct
      {
         int WParam;
         int LParam;
         int Result;
      };
   };
};
```

Сравните ее с более специальной структурой `TWMKey` для сообщений, связанных с клавишами.

```
struct TWMKey
{
   unsigned Msg;
   Word CharCode;
   Word Unused;
   int KeyData;
   int Result;
};
```

Очевидно, что эта структура проще и понятнее, чем структура `TMessage`. Для обработки сообщения, которое не имеет специальной структуры, можно создать собственную структуру. Она должна быть длиной 16 байт и иметь формат, аналогичный показанным ниже.

```
struct TMyMessageStructure
{
   unsigned Msg;      // Параметр Msg
   Word MessageData1; // Параметр WParam
   Word Unused;       // Параметр WParam
   int MessageData2;  // Параметр LParam
   int Result;        // Параметр Result
};
// или
struct TMyMessageStructure
{
   unsigned Msg;      // Параметр Msg
   int MessageData1;  // Параметр WParam
   int MessageData2;  // Параметр LParam
   int Result;        // Параметр Result
};
```

Первые 4 байта предназначены для поля `Msg`, последние 4 байта — для поля `Result`. Распределение 8 байтов для `WParam` и `LParam` зависит от назначения сообщения. Более подробные сведения на эту тему можно получить, обратившись к файлу `$(BCB)\Include\Vcl\Messages.hpp`.

Наконец, заключительной строкой карты сообщений будет макрос END_MESSAGE_MAP(base), где base — базовый класс библиотеки VCL для компонента. Например, для компонента с базовым классом TWinControl эта строка будет иметь следующий вид.

```
END_MESSAGE_MAP(TWinControl)
```

Как работают эти макросы? Как показано в листинге 11.3, они переопределяют метод Dispatch(). Их фактические определения можно найти в файле $(BCB)\Include\Vcl\Sysmac.h.

Листинг 11.3. Метод Dispatch() в макросах карты сообщений

```
// Макрос BEGIN_MESSAGE_MAP
virtual void __fastcall Dispatch(void*Message)
{
   switch(((TMessage*)Message)->Msg)
   {

// Макрос VCL_MESSAGE_HANDLER(msg,type,meth):
      case msg : meth(*((type*)Message));
      break;

// Макрос END_MESSAGE_MAP(base):
      default : base::Dispatch(Message);
      break;
   }
}
```

Метод Dispatch() переопределяется здесь, потому что он вызывается по умолчанию в конце метода WndProc() компонента. Именно метод WndProc() фактически получает сообщения. Он подробно описывается в следующем разделе о переопределении метода WndProc() для обработки сообщений в визуальных компонентах.

Карта сообщений и подробный вид макросов для сообщения WM_CHAR в компоненте с базовым классом TWinControl и методом-обработчиком WMChar() показаны в листингах 11.4 и 11.5.

Листинг 11.4. Карта сообщения для сообщения WM_CHAR

```
BEGIN_MESSAGE_MAP
VCL_MESSAGE_HANDLER(WM_CHAR,TWMChar,WMChar)
END_MESSAGE_MAP(TWinControl)
```

Листинг 11.5. Код метода Dispatch() для сообщения WM_CHAR

```
virtual void __fastcall Dispatch(void* Message)
{
   switch(((TMessage*)Message)->Msg)
   {
      case msg : WMChar(*((TWMChar*)Message));
      break;
      default : TWinControl::Dispatch(Message);
      break;
   }
}
```

Макросы карты сообщений существенно упрощают процесс переопределения метода Dispatch() за счет скрытия подробностей обработки сообщений. Знание принципа работы этих макросов позволит разработчику сэкономить время и свести до минимума вероятность возникновения ошибок.

Теперь, переопределив метод Dispatch(), рассмотрим код метода-обработчика сообщения. Как уже говорилось, этот метод не возвращает значения (void) и принимает в качестве единственного параметра ссылку на структуру сообщения. В более общем случае, т.е. при использовании структуры TMessage, объявление метода-обработчика выглядит следующим образом.

```
void __fastcall имя_обработчика(TMessage& Message);
```

Объявление метода-обработчика будет иметь следующий вид.

```
void __fastcall WMChar(TWMChar& Message);
```

На заметку

В объявлении метода-обработчика сообщения можно использовать макрос MESSAGE. Хотя он не содержит компилируемого кода, но позволяет быстро найти все обработчики сообщения в определении класса. Вот как выглядит пример объявления такого метода-обработчика `MESSAGE void __fastcall имя_обработчика(TMessage& Message);`.

Код метода-обработчика зависит от реакции компонента на сообщение и от типа обрабатываемого сообщения.

- Если обрабатываемое сообщение служит указателем на некоторое условие (это условие определяется с помощью проверки параметров сообщения), то метод Dispatch() базового класса компонента следует вызывать в том случае, если сообщение не удовлетворяет требуемому условию.
- Если обработчик сообщения выполняет только дополнительную обработку, связанную с сообщением, то следует вызвать метод Dispatch() базового класса компонента.
- Если обработчик сообщения выполняет всю необходимую обработку, связанную с сообщением, включая обработку, выполняемую в базовом классе, то не следует вызывать метод Dispatch() базового класса компонента.
- Если при поступлении сообщения вы не хотите выполнять никакую обработку, за исключением заданной в методе-обработчике сообщения, то также не следует вызывать метод Dispatch() базового класса компонента.

Вызов метода Dispatch() базового класса компонента гарантирует выполнение всей заданной по умолчанию обработки сообщений. В большинстве случаев этот метод следует вызвать после выполнения специальной обработки данного сообщения. Если метод Dispatch() базового класса компонента не вызывается, то придется вручную указать значение для поля Result структуры сообщения (обычно структуры TMessage), которая передается методу-обработчику. Конкретное значение зависит от обрабатываемого сообщения и способа его обработки. Более подробные сведения по этому поводу можно найти в справочных файлах Win32 SDK. Типичный код метода-обработчика сообщения имеет следующий вид:

```
void __fastcall ТПользовательскийКомпонент::ОбработчикСообщения
   (TMessage& Message)
{
   if(Message.SomeField == SomeValue)
   {
      // Обработка сообщения
      // Message.Result =?
   }
```

```
   else
   {
      BaseClass::Dispatch(Message);
   }
}
```

Карты сообщений позволяют управлять сообщениями в визуальных компонентах. Примеры использования сообщений в компонентах приводятся в главе 9 в разделе, посвященном созданию визуальных компонентов.

Переопределение метода WndProc() для управления сообщениями в визуальных компонентах

Метод WndProc() получает сообщения, посланные компоненту. Метод Dispatch() вызывается в конце метода WndProc() для управления сообщениями, полученными обработчиком. Для управления сообщениями компонента можно просто переопределить этот метод и обнаружить все заинтересовавшие вас сообщения. Метод WndProc() базового класса следует вызывать всякий раз при получении сообщения, которое не следует обрабатывать, или при наличии принятого по умолчанию обработчика сообщения. Этот гарантирует вызов метода Dispatch(). Общий вид переопределенного метода WndProc() будет иметь следующий вид.

```
void __fastcall ТПользовательскийКомпонент::WndProc(TMessage& Message)
{
   if(Message.Msg == Сообщение)
   {
      // Обработка сообщения
      // Message.Result = ?
   }
   else
   {
      БазовыйКласс::WndProc(Message);
   }
}
```

Не забудьте проверить параметры обрабатываемого сообщения, чтобы установить нужное значение для поля Result сообщения Message. Вызов метода WndProc() базового класса имеет большое значение, поскольку он вызывает метод Dispatch(), что гарантирует корректную обработку других сообщений. Это также гарантирует корректную обработку сообщений при работе с компонентом при создании приложения. Переопределяя метод WndProc(), вы можете ответить на сообщения еще до их обработки. Это значит, что появляется возможность полного игнорирования сообщений.

Использование методов AllocateHWnd() и DeallocateHWnd() для обработки сообщений в невизуальных компонентах

Невизуальные компоненты (т.е. производные от класса TComponent) не имеют окна и не получают сообщений. Однако иногда требуется организовать обработку сообщений в невизуальных компонентах. Это бывает нужно при создании компонентов, которые являются оболочкой функций Windows API. Если сам компонент должен получать сообщения, то следует использовать функции AllocateHWnd() и DeallocateHWnd() для создания и удаления скрытого окна, используемого компонентом. Функции AllocateHWnd() и DeallocateHWnd() объявлены в файле $(BCB)\Include\Vcl\Forms.hpp в следующем виде.

```
HWND __fastcall AllocateHWnd(Controls::TWndMethod Method);
void __fastcall DeallocateHWnd(HWND Wnd);
```

При вызове функции `AllocateHWnd()` передается имя оконной процедуры, которая будет обрабатывать полученные сообщения.

```
void __fastcall ИмяОконнойПроцедуры(TMessage& Message);
```

Это объявление можно найти в файле `$(BCB)\Include\Vcl\Controls.hpp`, оно также повторяется в конце следующего раздела, посвященного использованию невизуальных компонентов для ответа на сообщения, посланные другим компонентам. Оно имеет такой же вид, как метод `WndProc()` в классах-наследниках класса `TControl`. Затем функция `AllocateHWnd()` возвращает дескриптор созданного окна. Для использования функции `AllocateHWnd()` с целью получения сообщений в невизуальных компонентах нужно выполнить следующие действия.

1. Создать оконную функцию-процедуру для получения сообщений окном, созданным функцией `AllocateHWnd()`. Она должна иметь следующее объявление: `void __fastcall WndProc(TMessage& Message);`.
2. Создать закрытый член `FWindowHandle` типа `HWND` для хранения дескриптора окна, возвращаемого функцией `ALlocateHWnd()`.
3. Вызвать функцию `AllocateHWnd()` в конструкторе компонента для выделения окна, передавая в качестве аргумента имя оконной процедуры `WndProc`.
4. Вызвать функцию `DeallocateHWnd()` в деструкторе компонента для, передавая в качестве аргумента дескриптор окна `FWindowHandle`.

Обработку сообщения следует выполнить внутри метода `WndProc()`. Итак, в определение класса компонента необходимо добавить следующие строки.

```
// В разделе закрытых членов
HWND FWindowHandle;
// В разделе закрытых членов
void __fastcall WndProc(TMessage& Message);
```

В код конструктора компонента следует добавить такую строку кода.

```
// В коде конструктора
FWindowHandle = AllocateHWnd(WndProc);
```

В код деструктора компонента следует добавить такую строку кода.

```
// В коде деструктора
DeallocateHWnd(FWindowHandle);
```

Наконец, в код метода `WndProc()` нужно добавить следующие строки.

```
void __fastcall TCustomComponent::WndProc(TMessage& Message)
{
   //*********************//
   // Обработка сообщений //
   //*********************//

   Dispatch(&Message); // Управление другими сообщениями
}
```

Для необрабатываемых сообщений можно вызвать метод `Dispatch()`.

Использование невизуальных компонентов для ответа на сообщения, посланные другим компонентам

Иногда невизуальным компонентам нужно отслеживать сообщения, посланные другим оконным элементам управления (наследникам), например кнопке или форме. В таких случаях необходимо заменить оконную процедуру элемента управления оконной процедурой вашего компонента. Это значит, что сообщения, посланные этому элементу управления, будут получены вашим компонентом. Хитрость в данном случае заключается в том, что обычно перехватываются только одно или два сообщения, а остальные — передаются для обработки элементу управления.

Для этого используется *технология подчиненных классов* (*subclassing*), которая заключается в вызове собственной оконной процедуры для выполнения пользовательской обработки сообщений, вместо стандартной обработки сообщений окном. Затем вызывается исходная оконная процедура для выполнения принятой по умолчанию обработки сообщений. Эта технология может быть реализована с помощью функций Windows API или на основе библиотеки VCL.

Подход на основе функций Windows API сложнее, чем на основе библиотеки VCL, но он позволяет лучше понять весь этот процесс. Для этого предусмотрены две функции Windows API. Одна функция, `SetWindowLong()`, позволяет указать новую оконную процедуру, которая возвращает адрес предыдущей оконной процедуры этого окна. Другая функция, `CallWindowProc()`, позволяет вызвать заданную оконную процедуру по указанному адресу. Более подробно обе эти функции описываются в конце этого раздела, а также в справочных файлах Win32 SDK.

Чтобы использовать эти функции для пересылки невизуальному компоненту сообщений элемента управления, необходимо выполнить следующие действия.

1. Объявить оконную процедуру для получения сообщений элемента управления: `void __fastcall WndProc(TMessage& Message);`.
2. Объявить переменную для хранения указателя на оконную процедуру элемента управления, чтобы ее можно было восстановить после удаления компонента: `void* PreviousControlWndProc;`.
3. Преобразовать адрес функции `WndProc()` в указатель, необходимый для функции `SetWindowLong()`, с помощью функции `MakeObjectInstance()`. Эта функция объявлена в файле `$(BCB)\Include\Vcl\Forms.hpp`. В качестве аргумента ей передается имя функции, а возвращается значение типа `void*`, которое может использоваться функцией `SetWindowLong()`. Следовательно, для хранения этого значения нужно определить другую переменную `void* NewComponentWndProc;`.

После этих действий у нас есть практически все для получения сообщений другого элемента управления. Заключительное действие состоит в том, чтобы определить элемент управления, сообщения которого будут перехватываться. Этот выбор зависит от типа компонента. Например, при создании компонента типа вам нужно отслеживать сообщения, посланные в форму, в которой он размещается. В этом случае элемент управления будет владельцем вашего компонента. Однако не исключено, что вашему компоненту может потребоваться также отслеживать сообщения, посланные другим элементам управления той же формы. В таком случае удобно включить свойство `TWinControl*` и оконную процедуру элемента управления с помощью set-метода заменить оконной процедурой компонента. Это немного сложнее, чем в первом случае, когда элемент управления является владельцем компонента, но в общем эти методы похожи. В следующем примере показан владелец компонента (полученный с помощью свойства `Owner`), сообщения которого требуется отслеживать. В лис-

тинге 11.6 показано определение класса для невизуального компонента, который может отслеживать сообщения владельца (Owner). Этот класс является наследником класса TWinControl. Следовательно, потребуется выполнить динамическое приведение (dynamic_cast) типа свойства Owner (TComponent*) к типу TWinControl*.

Листинг 11.6. Определение компонента для создания подчиненного класса для элемента управления на основе функций Windows API

```
class PACKAGE TSubClassComponent : public TComponent
{
private:
   void* PrevControlWndProc;
   void* NewComponentWndProc;

protected:
   virtual void __fastcall WndProc(TMessage& Message);
   virtual void __fastcall Loaded();

public:
   __fastcall TSubClassComponent(TComponent* Owner);
   __fastcall ~TSubClassComponent();
};
```

В листинге 11.7 показан код методов, объявленных в листинге 11.6.

Листинг 11.7. Код компонента, используемого при создании подчиненного класса для элемента управления на основе Windows API

```
//---------------------------------------------------------//
// КОНСТРУКТОР

__fastcall
   TSubClassComponent::TSubClassComponent(TComponent* Owner)
                                      : TComponent(Owner),
                                        PrevControlWndProc(0),
                                        NewComponentWndProc(0)
{
   TWinControl*WinControl = dynamic_cast<TWinControl*>(Owner);
   if(!ComponentState.Contains(csDesigning) && WinControl)
   {
      NewComponentWndProc = MakeObjectInstance(WndProc);
   }
}

//---------------------------------------------------------//
// ДЕСТРУКТОР

__fastcall
   TSubClassComponent::~TSubClassComponent()
{
   TWinControl* WinControl = dynamic_cast<TWinControl*>(Owner);
```

```
   if(NewComponentWndProc != 0 && WinControl)
   {
      // Восстановление прежней оконной процедуры WndProc
      SetWindowLong(WinControl->Handle,
                    GWL_WNDPROC,
                    reinterpret_cast<LONG>(PrevControlWndProc));
      PrevControlWndProc = 0;
      FreeObjectInstance(NewComponentWndProc);
   }
}

//-----------------------------------------------------------//
// МЕТОД : WndProc() (Оконная процедура)

void __fastcall
   TSubClassComponent::WndProc(TMessage& Message)
{
   //************************************
   // Здесь выполняется обработка сообщений
   //************************************

   // Затем вызывается метод WndProc() с целью обработки
   // сообщений для оконных элементов управления
   if(PrevControlWndProc != 0)
   {
      TWinControl*WinControl = dynamic_cast<TWinControl*>(Owner);

      Message.Result =
   CallWindowProc(reinterpret_cast<FARPROC>(PrevControlWndProc),
                  WinControl->Handle,
                  Message.Msg,
                  Message.WParam,
                  Message.LParam);
   }
}

//-----------------------------------------------------------//
// МЕТОД : Loaded()

void __fastcall TSubClassComponent::Loaded()
{
   TComponent::Loaded();

   TWinControl* WinControl = dynamic_cast<TWinControl*>(Owner);

   if(!ComponentState.Contains(csDesigning) && WinControl)
   {
      // Замена оконной процедуры WndProc собственной
      PrevControlWndProc
```

```
      =reinterpret_cast<void*>
         (SetWindowLong(WinControl->Handle,
          GWL_WNDPROC,
          reinterpret_cast<LONG>(NewComponentWndProc)));
   }
}

//---------------------------------------------------------//
```

Код в листинге 11.7 не нуждается в излишних комментариях. Достаточно разобраться в принципах работы функций `SetWindowLong()` и `CallWindowProc()`, остальное же достаточно очевидно. Рассмотрим функцию `SetWindowLong()`, которая имеет следующее объявление.

```
LONG SetWindowLong(HWND hWnd, int nIndex, LONG dwNewLong);
```

Эта функция изменяет характеристики окна, указанного с помощью первого параметра, `hWnd`. Сама характеристика определяется значением второго параметра, `nIndex`. Этим параметром определяется количество констант, включая `GWL_WNDPROC`. Передача `GWL_WNDPROC` функции означает необходимость изменения функции `WndProc()`. Значение третьего параметра определяется изменяемой характеристикой окна (который определяется вторым параметром). В данном случае нам нужно изменить оконную процедуру, поэтому в качестве аргумента для этого параметра передается указатель на новую оконную процедуру. Возвращаемое этой функцией значение зависит от второго параметра. Так как изменяется оконная процедура, то эта функция возвращает указатель на предыдущую оконную процедуру, чтобы ее можно было восстановить впоследствии в деструкторе компонента.

В листинге 11.6 указатели на новую и предыдущую оконные процедуры объявлены с типом `void*`. Так как третий параметр `SetWindowLong()` принимает значение типа `LONG`, то необходимо выполнить приведение (`reinterpret_cast`) типа `void*` к типу `LONG`. Такое преобразование типа необходимо также выполнить для возвращаемого значения функции `SetWindowLong()`. Новая оконная процедура инсталлируется в методе `Loaded()` компонента, чтобы гарантировать ее полную загрузку и готовность к обработке сообщений.

Функция `CalllWindowProc()` вызывается в коде метода `WndProc()` нашего компонента. Он имеет решающее значение для всего компонента. Инсталляция собственной оконной процедуры для управления сообщениями владельца компонента выполняется достаточно просто. Однако при этом следует обеспечить корректную обработку тех сообщений, которые не обрабатываются в нашем компоненте. Этого можно достичь, возвратив необрабатываемые сообщения из метода `WndProc()` исходной оконной процедуре наследника класса `TWinControl*`. Эта функция `CallWindowProc()` имеет следующее объявление.

```
LRESULT CallWindowProc(FARPROC lpPrevWndFunc, // указатель
                                              // оконной
                                              // процедуры
                       HWND hWnd,             // дескриптор окна
                       UINT Msg,              // сообщение
                       WPARAM wParam,         // первый параметр
                                              // сообщения
                       LPARAM lParam)         // второй параметр
                                              // сообщения
```

Первый параметр содержит указатель оконной процедуры, которая предназначена для обработки сообщения. Он имеет тип `FARPROC`, поэтому необходимо выполнить приведение

(reinterpret_cast) типа void* к типу FARPROC. Затем передается дескриптор окна, которое получает сообщение (в данном случае это дескриптор наследника TWinControl и владельца нашего компонента). Остальные три параметра содержат значение полученного сообщения. Возвращаемое значение этой функции является результатом обработки сообщения и хранится в свойстве Message.Result.

На первый взгляд код мониторинга сообщений в другом окне кажется очень сложным, но чтобы разобраться в нем, достаточно понять принцип работы двух Windows API функций: SetWindowLong() и CallWindowProc().

Теперь, обсудив способы использования функций Windows API при создании подчиненного класса для оконной процедуры элемента управления, рассмотрим подход на основе библиотеки VCL. Каждый наследник класса TControl имеет свойство WindowProc (в данном случае основное внимание будет уделено наследникам класса TWinControl). Это свойство содержит указатель на метод WndProc() элемента управления. Свойство WindowProc имеет тип TWndMethod и объявлено в файле $(BCB)\Include\Vcl\Controls.hpp следующим образом.

```
typedef void __fastcall (__closure *TWndMethod)
   (Messages::TMessage& Message);
```

Как показано в предыдущем разделе, оконная процедура используемая наследниками класса Tcontrol, имеет следующий вид.

```
void __fastcall WindowProcedureName(TMessage& Message);
```

Это именно та функция, на которую указывает свойство WindowProc. Чтобы указать новую оконную процедуру для этого элемента управления, нужно задать соответствующее значение свойства WindowProc. Для этого необходимо сначала сохранить прежнее значение WindowProc, чтобы при удалении компонента можно было восстановить исходную оконную процедуру элемента управления. Кроме того, этот сохраненный указатель можно использовать для вызова исходной оконной процедуры (метод WndProc()) и выполнения необходимой обработки сообщений.

В листингах 11.8 и 11.9 показано определение и реализация класса для простого компонента, в котором используется данный подход.

Листинг 11.8. Определение компонента при создании подчиненного класса для элемента управления на основе библиотеки VCL

```
class PACKAGE TVCLSubClassComponent : public TComponent
{
private:
   TWndMethod ControlWndProc;

protected:
   virtual void __fastcall WndProc(TMessage& Message);
   virtual void __fastcall Loaded();

public:
   __fastcall TVCLSubClassComponent(TComponent* Owner);
   __fastcall ~TVCLSubClassComponent();

__published:
};
```

Листинг 11.9. Код компонента при создании подчиненного класса для элемента управления на основе библиотеки VCL

```
//-----------------------------------------------------------//
// КОНСТРУКТОР

__fastcall
 TVCLSubClassComponent::TVCLSubClassComponent(TComponent* Owner)
   : TComponent(Owner),
     ControlWndProc(0)
{
}

//-----------------------------------------------------------//
// ДЕСТРУКТОР

__fastcall
 TVCLSubClassComponent::~TVCLSubClassComponent()
{
   if(!ComponentState.Contains(csDesigning))
   {
      TWinControl* WinControl =
         dynamic_cast<TWinControl*>(Owner);
      if(WinControl)
      {
         // Восстановление исходной процедуры WndProc()
         WinControl->WindowProc = WndProc;
      }
   }
}

//-----------------------------------------------------------//
// МЕТОД : WndProc() (Оконная процедура)

void __fastcall
   TVCLSubClassComponent::WndProc(TMessage& Message)
{
   //*********************//
   // Обработка сообщений //
   //*********************//
   if(ControlWndProc)
   {
      // Вызов исходной оконной процедуры WndProc()
      ControlWndProc(Message);
   }
}

//-----------------------------------------------------------//
// МЕТОД : Loaded()
void __fastcall
```

```
   TVCLSubClassComponent::Loaded()
{
   TComponent::Loaded();

   if(!ComponentState.Contains(csDesigning))
   {
      TWinControl*WinControl =
         dynamic_cast<TWinControl*>(Owner);

      if(WinControl)
      {
         // Сохранение старой оконной процедуры WndProc()
         // и указание новой оконной процедуры WndProc()
         ControlWndProc = WinControl->WindowProc;
         WinControl->WindowProc = WndProc;
      }
   }
}

//---------------------------------------------------------//
```

Код в листинге 11.9 гораздо проще, чем в листинге 11.7, хотя их базовые принципы одинаковы. Обратите внимание, как легко задается новая оконная процедура. Для этого достаточно ввести строку `WinControl->WindowProc = WndProc;` в код метода `Loaded()`. Так же просто организован вызов исходной оконной процедуры элемента управления с помощью строки `ControlWndProc(Message);` в методе `WndProc()`. Сообщение `Message` передается по ссылке, так что не требуется установка поля `Result` на основании возвращаемого значения, как это было при выполнении этой операции с помощью функции `CallWindowProc()`.

Код листингов 11.6–11.9 находится в каталоге `SubClass` на прилагаемом к книге компакт-диске в модулях `SubClassComponent` и `VCLSubClassComponent`.

Применение функций обратного вызова в компонентах

Чаще всего компоненты используются в качестве оболочек для функций Windows API. Обычно функциональность таких оболочек API зависит от использования функций обратного вызова, т.е. ваш компонент должен уметь ответить на отдельные обратные вызовы Windows, передавая указатель на подходящую функцию обратного вызова, которая содержится внутри компонента, как параметр в виде отдельной API-функции. При возникновении какого-либо события соответствующая DLL API-функции вызовет имеющуюся функцию обратного вызова. Рассмотрим этот процесс более подробно на примере функций Windows Telephony API (TAPI) и Video Capture API.

Для организации вызова с помощью TAPI обычно выполняются три действия: инициализируется используемая линия (`lineInitialize()`), открывается эта линия (`lineOpen()`), а затем организуется вызов в этой линии (`lineMakeCall()`). На самом деле нужно выполнить большее количество действий, но в данном разделе обсуждаются функции обратного вызова, а не TAPI функции.

На заметку

Действительно, функция `lineInitialize()` уже устарела и вместо нее следует использовать функцию `lineInitializeEx()`. Однако для упрощения здесь используется функция `lineInitialize()`. Рекомендуется внимательно изучить эти три функции для работы с телефонными подключениями на примерах файлов Win32 SDK, которые входят в состав C++Builder. Обратите внимание, что новый интерфейс TAPI 3.0 очень отличается от прежнего интерфейса TAPI 1.4 (используемого в данном примере) и интерфейса TAPI 2.0. Более подробную информацию о них можно получить в файлах справки Win32 SDK.

Принцип работы этого вызова можно проследить с помощью функции обратного вызова. Для этого необходимо задать подходящую функцию обратного вызова в качестве параметра функции `lineInitialize()`. Объявление этой функции имеет следующий вид.

```
long lineInitialize(HPLINEAPP lphLineApp,
                    HINSTANCE hInstance,
                    LINECALLBACK lpfnCallback,
                    LPCSTR lpszAppName,
                    LPDWORD lpdwNumDevs);
```

А объявление функции обратного вызова имеет такой вид.

```
void CALLBACK CallbackFunctionName(DWORD hDevice,
                                   DWORD dwMsg,
                                   DWORD dwCallbackInstance,
                                   DWORD dwParam1,
                                   DWORD dwParam2,
                                   DWORD dwParam3)
```

На заметку

Обратите внимание на использование макроса `CALLBACK`, который определен как `__stdcall`. Это позволяет сообщить компилятору о том, что для функции следует использовать более эффективную стандартную последовательность вызовов "Standard" с фиксированным количеством передаваемых ей параметров. По умолчанию используется последовательность вызовов "C" (`_cdecl`), где количество передаваемых параметров может меняться, что делает менее эффективным очистку стека после вызова функции. Для функций обратного вызова Windows в целях повышения переносимости, вместо непосредственного использования ключевого слова `__stdcall`, рекомендуется использовать макрос `CALLBACK`. Обычно эту последовательность вызова используют другие функции Windows API, но при этом применяется макрос `WINAPI`.

Игнорируя временно параметр `dwCallBackInstance` (к обсуждению которого мы вернемся позже), можно передать эту функцию функции `lineInitialize()` следующим образом.

```
HLINEAPP LineAppHandle;
DWORD NumberOfTAPIDevices;

lineInitialize(&LineAppHandle,
               HInstance,
               CallbackFunctionName,
               "MyTAPIApplication",
               &NumberOfTAPIDevices );
```

Дополнительные параметры имеют следующее назначение: `&LineAppHandle` используется как параметр для других функций в работе с телефонными подключениями для подтверждения успешной инициализации линии; параметр `HInstance` является экземпляром текущего

процесса; строка MyTAPIApplication — имя приложения; а параметр &NumberOfTapiDevices содержит количество имеющихся устройств TAPI.

Еще один параметр, CallbackFunctionName, также имеет очень большое значение. Он содержит имя функции обратного вызова. При создании компонента может возникнуть препятствие. Прототип нашей функции обратного вызова имеет следующий вид.

```
void CALLBACK CallbackFunctionName(DWORD hDevice,
                                   DWORD dwMsg,
                                   DWORD dwCallbackInstance,
                                   DWORD dwParam1,
                                   DWORD dwParam2,
                                   DWORD dwParam3)
```

Именно такой тип указателя функции необходим для lineInitialize(). Напомним, что суть написания этого кода заключается в создании компонента, который окружает все используемые API-функции. Допустим теперь, что функция обратного вызова вызывается в случае возникновения некоторых событий для адекватного отклика на них. В компоненте C++Builder обычно предусмотрено инициирование событий, а пользователь компонента может создать собственный обработчик каждого из них. Например, функция обратного вызова будет вызвана TAPI после подключения. Обычно в таком случае требуется вызвать событие OnCallConnected (или аналогичное ему событие). Для вызова событий обычно требуется доступ к закрытым и защищенным функциям и данным. Действительно, для выполнения практически любой операции в функции обратного вызова требуется осуществить доступ к закрытым и защищенным функциям и данным вашего компонента. В таком случае функция обратного вызова должна быть либо дружественной функцией, либо членом-функцией вашего компонента.

На заметку

Дружественной (friend) функцией класса называется функция, которая обладает доступом к закрытым и защищенным функциям и данным класса, но не является частью этого класса. Для объявления дружественной функции класса следует переопределить эту функцию в определении класса, предоставляющего дружественный доступ с помощью ключевого слова friend. Учтите, что пользователи этого класса имеют открытый доступ к такой функции.

Однако определение функции обратного вызова как дружественной делает ее открытой. Ясно, что нежелательно делать открытой функцию обратного вызова. Еще один способ решения этой задачи заключается в создании закрытого или защищенного функции-члена самого компонента. Однако этот способ связан с другой проблемой. Для функции lineInitialize() нельзя передать указатель члена-функции, поскольку в ней используется 4-байтовый указатель функции, а не 8-байтовый указатель члена-функции. Для определения функции обратного вызова в качестве закрытого или защищенного члена компонента необходимо создать статичный класс-член. Например, для компонента TTAPIComponent можно использовать следующее (хотя и не полное) определение.

```
class PACKAGE TTAPIComponent : public TComponent
{
private:
   static void CALLBACK CallbackFunctionName(DWORD hDevice,
                           DWORD dwMsg,
                           DWORD dwCallbackInstance,
                           DWORD dwParam1,
                           DWORD dwParam2,
                           DWORD dwParam3 );
```

```
// Другие закрытые функции и данные

protected:
// Защищенные функции и данные

public:
   __fastcall TTAPIComponent(TComponent*Owner);
   __fastcall ~TTAPIComponent();

// Другие открытые функции и данные

};
```

Так как для каждого экземпляра компонента существует только одна функция обратного вызова, то необходимо знать метод определения экземпляра этого компонента для заданной функции обратного вызова. Для этого предназначен параметр `dwCallbackInstance` типа `DWORD` (4-байтный).

При инициализации TAPI-подключения устанавливается версия используемого TAPI-интерфейса (`lineNegotiateAPIVersion()`) и нужные параметры вызова (за счет заполнения структуры `LINECALLPARAMS` необходимыми значениями). Теперь это подключение можно открыть с помощью функции `lineOpen()` для размещения вызова. Вот как выглядит объявление функции `lineOpen()`.

```
long lineOpen(HLINEAPP hLineApp,
              DWORD dwDeviceID,
              LPHLINE lphLine,
              DWORD dwAPIVersion,
              DWORD dwExtVersion,
              DWORD dwCallbackInstance,
              DWORD dwPrivileges,
              DWORD dwMediaModes,
              LPLINECALLPARAMS const lpCallParams);
```

Обратите внимание на параметр `dwCallbackInstance`, который аналогичен параметру `dwCallbackInstance` в функции обратного вызова. При вызове функции обратного вызова DLL-модулем TAPI этот параметр передается ей в неизменном виде. Он имеет ключевое значение и ниже будет рассмотрен подробно. Код вызова метода `lineOpen()` имеет следующий вид.

```
HLINE Line;
LINECALLPARAMS CallParameters;

// Установка параметров CallParameters

lineOpen(LineAppHandle,          // на основе lineInitialize
         LINEMAPPER,             // нет выделенного lineID -
                                 // см. TAPI SDK
         Line,                   // дескриптор открытого подключения
         APIVersion,             // версия TAPI
         0,                      // нет расширенной версии
         reinterpret_cast<DWORD>(this),
         LINECALLPRIVILEGE_NONE, // только для исходящих звонков
         0,                      // не требуется, только для
                                 // исходящих звонков
         &CallParameters);       // параметры звонка
```

Наибольший интерес вызывает следующий параметр.

```
reinterpret_cast<DWORD>(this)
```

Он содержит результат приведения типа указателя this вызывающего компонента TTAPIComponent к типу DWORD. Это допустимо, так как указатель this и тип DWORD имеют размер 4 байта. Следовательно, таким образом передается указатель на компонент, который вызывает функцию lineOpen(). В коде этого компонента функция обратного вызова имеет следующий вид.

```
void CALLBACK TTAPIComponent::CallbackFunctionName(DWORD hDevice,
                                DWORD dwMsg,
                                DWORD dwCallbackInstance,
                                DWORD dwParam1,
                                DWORD dwParam2,
                                DWORD dwParam3 )
{
   TTAPIComponent* TAPIComponent
      = reinterpret_cast<TTAPIComponent*>(dwCallbackInstance);

   // Использование TAPIComponent-> для доступа к функциям и данным
   // компонента TTAPIComponent, который открыл подключение
   // (с помощью метода lineOpen()),
   // и, вероятно, инсталлировал функцию обратного вызова
   // (с помощью метода lineInitialize())
}
```

При вызове TAPI DLL-модулем функции обратного вызова параметр dwCallbackInstance содержит указатель на экземпляр компонента TTAPIComponent, который открыл телефонное подключение. Получив указатель на корректный экземпляр компонента, достаточно просто выполнять операции с членами-функциями и членами-данными. Кроме того, так как функция обратного вызова является членом-функцией, то никаких ограничений доступа не существует.

Обратите внимание: если создан класс TTAPIComponent, то все экземпляры такого класса-компонента будут совместно использовать функцию, которая является оболочкой функции lineInitialize(). Иначе говоря, ее можно сделать статической. Дело в том, что все экземпляры совместно используют одну и ту же статическую функцию обратного вызова, а другие параметры метода lineInitialize() определяются для приложения, а не для экземпляра компонента. С другой стороны, каждому компоненту нужно предоставить его собственную оболочку-функцию lineOpen(), так как она должна передать собственный указатель this в качестве параметра функции lineOpen(). Следовательно, функция-оболочка функции lineOpen() должна быть обычным членом-функцией.

Описанная выше методика использования функций обратного вызова Windows для Telephony API в равной степени может применяться для других функций обратного вызова Windows.

В следующем примере для иллюстрации функций обратного вызова используется Video Capture API. Более подробную информацию о функциях или макросах можно найти в разделе “Video Capture” в справке Win32 SDK. Однако для понимания этого примера не требуется глубокое знание Video Capture API.

При захвате видеокадра с помощью функций Video Capture API можно использовать функцию обратного вызова для указания имеющегося захваченного кадра. Для этого необходимо создать окно захвата с помощью функции capCreateCaptureWindow(). Эта функция возвращает дескриптор типа HWND созданного окна захвата. Затем окно захвата подключается к подходящему драйверу захвата видеокадра с помощью макроса capConnectDriver. Дескриптор окна захвата, созданный функцией

capCreateCaptureWindow(), требуется для последующих вызовов функций захвата видеокадра. В ходе дальнейшего обсуждения мы будем называть его hCaptureWindow. Для захвата кадров и получения обратного вызова необходимо выполнить следующие действия: инсталлировать подходящую функцию обратного вызова для DLL-модуля, ответственного за захват видеокадра, и запустить сам процесс захвата. Функция обратного вызова, необходимая для захвата видеокадров, имеет следующий вид.

```
LRESULT CALLBACK VideoStreamCallback(HWND hCapWin,
                                     LPVIDEOHDR FrameInfo );
```

Как и в предыдущем примере, для создания функции обратного вызова в компоненте ее необходимо определить в виде статической функции-члена. Определение класса для компонента TVideoCaptureComponent будет иметь следующий вид.

```
class PACKAGE TVideoCaptureComponent : public TComponent
{
private:
   static LRESULT CALLBACK VSCallbackFunctionName(HWND hCapWin,
                               LPVIDEOHDR FrameInfo );

   // Другие закрытые функции и данные

protected:
   // Защищенные функции и данные

public:
   __fastcall TVideoCaptureComponent(TComponent* Owner);
   __fastcall ~TVideoCaptureComponent ();

   // Другие открытые функции и данные

};
```

На заметку

В этом случае возвращаемое значение функции обратного вызова имеет тип LRESULT. Этот тип определен с помощью ключевого слова typedef в файле $(BCB)\Include\windefs.h для типа LONG_PTR, который, в свою очередь, объявлен в файле $(BCB)\Include\basetsd.h — файле с определениями базовых типов Windows. Отметим, что тип LONG_PTR имеет ту же длину, что и указатель, т.е. на 32-разрядных компьютерах он имеет длину 32 бита (4 байта). Это значение со знаком.

Для инсталляции здесь используется макрос capSetCallbackOnVideoStream с передачей дескриптора окна захвата и функции обратного вызова в качестве параметров.

```
capSetCallbackOnVideoStream(hCaptureWindow,
                            VSCallbackFunctionName);
```

Этот макрос возвращает значение TRUE или FALSE, указывая на успешную или неуспешную инсталляцию функции обратного вызова. В первом случае нам потребуется только начать захват изображений с помощью вызова макроса capCaptureSequenceNoFile с передачей дескриптора окна захвата в качестве параметра.

```
capCaptureSequenceNoFile(hCaptureWindow);
```

Макрос в случае успешного выполнения возвращает значение TRUE. Обратите внимание: макрос capCaptureSequenceNoFile используется потому, что предполагается выполнение не-

которых действий с захваченными кадрами, например передача их другому компьютеру. В этом случае не обязательно сохранять захваченные изображения в файле.

Приступим теперь к созданию кода функции обратного вызова. Здесь следует использовать иной подход, чем в случае функций TAPI. Во-первых, в этой функции обратного вызова не нужно передавать данные-экземпляр. Это значит, что нельзя непосредственно определить компонент, для которого предназначена эта функция обратного вызова. Однако можно определить окно захвата этого вызова, так как при инсталляции функции обратного вызова в качестве одного из параметров передается дескриптор окна захвата. Для определения нужного экземпляра компонента необходимо последовательно перебрать текущие экземпляры `TVideoCaptureComponent` и найти окно захвата компонента с соответствующим параметром типа `HWND` (`hCapWin`) функции обратного вызова. При этом предполагается наличие только одного окна захвата для каждого компонента, в противном случае (что маловероятно) придется поддерживать целый вектор дескрипторов окон захвата. Для последовательного перебора текущих экземпляров компонента `TVideoCaptureComponent` нужно создать статический класс-контейнер, содержащий указатели на каждый доступный экземпляр компонента `TVideoCaptureComponent`. Подходящим контейнером является стандартный контейнер `vector` библиотеки C++ Standard Template Library. Он позволяет поддерживать массив указателей, обеспечивая простоту удаления его элементов и доступа к ним. Для возможности его использования следует объявить: статический член-вектор, статическую переменную для следующего доступного индекса в векторе указателей экземпляров, а также переменную для хранения индекса указателя `this` текущего экземпляра. Определение этого класса приводится в листинге 11.10.

Листинг 11.10. Определение класса для компонента захвата видеокадра

```
class PACKAGE TVideoCaptureComponent : public TComponent
{
private:
   static LRESULT CALLBACK VSCallbackFunctionName(HWND hCapWin,
                            LPVIDEOHDR FrameInfo );
   static std::vector<TVideoCaptureComponent*>
      VideoCaptureInstances;
   static unsigned NextAvailableIndex;

   unsigned ThisInstanceIndex;

// Другие закрытые функции и данные

protected:
// Защищенные функции и данные

public:
   __fastcall TVideoCaptureComponent(TComponent* Owner);
   __fastcall ~TVideoCaptureComponent ();

// Другие открытые функции и данные

};
```

В конструкторе `TVideoCaptureComponent` указатель `this` добавляется в вектор, а в деструкторе для него задается значение 0. Исходный код компонента захвата показан в листинге 11.11.

Листинг 11.11. Код компонента захвата видеокадра

```
// После директив #include и перед другим кодом

   std::vector<TVideoCaptureComponent*>
      TvideoCaptureComponent::VideoCaptureInstances;
   unsigned TVideoCaptureComponent::NextAvailableIndex = 0;

// КОНСТРУКТОР
__fastcall TVideoCaptureComponennt::
↳TVideoCaptureComponent(TComponent* Owner)
    : TComponent(Owner)
{
   // Код конструктора.

   // Теперь добавим указатель на вектор VideoCaptureInstances.
   // 1 - Сохранение индекса для этого компонента.
   ThisInstanceIndex = NextAvailableIndex;

   // 2 - Включение указателя this в этот вектор.
   if(ThisInstanceIndex == VideoCaptureInstances.size())
   {
      VideoCaptureInstances.push_back(this);
   }
   else VideoCaptureInstances[ThisInstanceIndex] == this;

   // 3 - Определение следующего доступного индекса.
   NextAvailableIndex = 0;

   while( NextAvailableIndex < VideoCaptureInstances.size()
          && VideoCaptureInstances[NextAvailableIndex] != 0 )
   {
      ++NextAvailableIndex;
   }
}

// ДЕСТРУКТОР

__fastcall TVideoCapture::~TVideoCapture()
{
   VideoCaptureInstances[ThisInstanceIndex] == 0;

   // Код деструктора.
}
```

Вот как вкратце можно описать код в листинге 11.11. Если вектор заполнен, новое значение добавляется в конце вектора с помощью члена-функции `push_back()`. В противном случае значение указателя `this` располагается в незанятом элементе (т.е. в элементе равном нулю). Затем с помощью следующего цикла определяется следующий доступный индекс.

```
while(NextAvailableIndex < VideoCaptureInstances.size()
      && VideoCaptureInstances[NextAvailableIndex] != 0 )
{
   ++NextAvailableIndex;
}
```

Обратите внимание: выражение

```
VideoCaptureInstances[NextAvailableIndex] != 0
```

будет оценено только в том случае, если значение `NextAvailableIndex` меньше, чем размер вектора, т.е. если первое выражение цикла `while` возвратит `true`. Если все элементы вектора содержат указатель на экземпляр компонента `TVideoCaptureComponent`, то значение `NextAvailableIndex` будет равно размеру вектора и функция `push_back()` будет использована для создания следующего указателя экземпляра.

При удалении экземпляра соответствующий ему элемент вектора, содержащего указатели экземпляры, задается равным нулю. Это делается с помощью переменной-члена `ThisInstanceIndex`, значение которой задается в конструкторе при добавлении указателя `this` в вектор. Этот элемент теперь может быть переписан следующим экземпляром. Используя указатель `this`, можно поддерживать вектор указателей, добавляя и удаляя в него указатели в любом порядке.

Наконец, рассмотрим саму функцию обратного вызова. Ее код устроен достаточно просто с использованием цикла `for` для поиска корректного экземпляра `TVideoCaptureComponent` для параметра `HWND hCapWin`.

```
LRESULT CALLBACK TVideoCaptureComponent::VideoStreamCallback(
                                          HWND hCapWin,
                                          LPVIDEOHDR FrameInfo)
{
   // Подготовка данных Frame с помощью указателя FrameInfo

   for(int i=0; i<VideoCaptureInstances.size(); i++)
   {
      if(hCaptureWindow ==
         VideoCaptureInstances[i]->hCaptureWindow)
      {
         // Выполнение операций только с экземпляром.
      }
   }
}
```

Завершая это обсуждение, отметим, что указатель `LPVIDEOHDR` указывает на структуру `VIDEOHDR`, которая содержит всю информацию о захваченном кадре, включая указатель на данные кадра. Она определена в файле `$(BCB)\Include\vfw.h`.

Ситуация, проиллюстрированная на примере функций Telephony API, встречается наиболее часто. Изучив приведенные в этом разделе примеры, вы легко справитесь с аналогичными проблемами в других функциях API.

Выбор фундаментальных типов свойств

В этом разделе представлены рекомендации, касающиеся выбора фундаментального типа для заданного свойства. Под фундаментальными типами подразумеваются такие встроенные

типы, как int, unsigned int, double, float, long double, char, bool и т.д. Это обсуждение может в равной степени быть отнесено и к интерфейсу компонента (или любого класса).

Начнем рассмотрение с целочисленных типов: char, short int, int и long int, а также беззнаковых значений.

Тип int

Тип int является базовым целочисленным типом. При создании целочисленного свойства тип int следует всегда использовать в качестве типа свойства, за исключением особых случаев. Причина такого поведения станет очевидной после прочтения следующих разделов.

Тип long int

В C++Builder тип long int имеет длину 32 бита, т.е. такую же, как и тип int. Следовательно, тип long int следует избегать использовать в качестве свойства, так как он усложнит восприятие интерфейса компонента.

Тип short int

Для чего используется свойство типа short int? Чаще всего для ограничения диапазона доступных значений для типа short int (от –32768 до 32767). Для сравнения: диапазон типа int существенно бо́льше (от –2147483648 до 2147483647). В таких случаях предполагается, что значение за пределами диапазона short int никогда не потребуется, а свойство типа short int выбирается потому, что этот способ более эффективен. Однако такой подход обладает некоторыми недостатками. Во-первых, использование типа short int не исключает возможности присвоения значения типа int этому свойству и обрезания присвоенного значения. Следовательно, присвоенное значение может оказаться совсем не таким, каким его ожидает увидеть пользователь. При использовании типа short int нельзя проследить за этой ситуацией. Кроме того, во внутреннем представлении тип short int преобразуется в тип int, поэтому никакой ощутимой выгоды от применения типа short int в качестве типа свойства получить не удастся.

Следует избегать использования типа short int в качестве типа свойства. При необходимости использования только некоторой части диапазона допустимых значений типа int следует явным образом проверять присваиваемое свойству значение в set-методе, выявляя в нем возможные ошибки выхода за пределы заданного диапазона. В зависимости от свойства, можно организовать обработку исключительной ситуации EOutOfRangeError. Взаимодействие с классами исключений библиотеки VCL можно организовать так, как показано в листинге 11.12. Этот код содержится в каталоге OutOfRange на прилагаемом к книге компакт-диске в файле OutOfRangeException.h.

Листинг 11.12. Определение класса исключения EOutOfRangeError

```
class EOutOfRangeError : public EIntError
{
public:
   inline __fastcall EOutOfRangeError(const AnsiString Msg)
      : EIntError(Msg) {}

   inline __fastcall EOutOfRangeError(const AnsiString Msg,
                                      const System::TVarRec* Args,
                                      const int Args_Size)
      : EIntError(Msg, Args, Args_Size) {}
```

```
   inline virtual __fastcall EOutOfRangeError(int Ident)
      : EIntError(Ident) {}

   inline __fastcall EOutOfRangeError(int Ident,
                                const System::TVarRec* Args,
                                const int Args_Size)
      : EIntError(Ident, Args, Args_Size) {}

   inline __fastcall EOutOfRangeError(const AnsiString Msg,
                                   int AHelpContext)
      : EIntError(Msg, AHelpContext) {}

   inline __fastcall EOutOfRangeError(const AnsiString Msg,
                                const System::TVarRec* Args,
                                const int Args_Size,
                                int AHelpContext)
      : EIntError(Msg, Args, Args_Size, AHelpContext) {}

   inline virtual __fastcall EOutOfRangeError(int Ident,
                                           int AHelpContext)
      : EIntError(Ident, AHelpContext) {}

   inline virtual __fastcall
      EOutOfRangeError(System::PResStringRec ResStringRec,
                       const System::TVarRec* Args,
                       const int Args_Size,
                       int AHelpContext)
     : EIntError(ResStringRec, Args, Args_Size, AHelpContext) {}

public:
   inline __fastcall virtual ~EOutOfRangeError(void) {}
};
```

В set-метод этого свойства следует ввести следующий код.

```
void __fastcall TComponentName::PropertySetMethod(int NewValue)
{
   if(FValue != NewValue)
   {
      if(NewValue < MinValue || NewValue > MaxValue)
      {
         AnsiString ErrorMessage;
         ErrorMessage.sprintf(
            "%d Out of Range : %d <= Value <= %d", Value,
            MinValue, MaxValue);
         throw EOutOfRangeError(ErrorMessage);
      }

      // Здесь выполняются другие задачи.

   }
}
```

Единственный случай, когда следует использовать тип `short int`, — необходимость двухбайтовой переменной (например, для двухбайтовой маски).

Тип `unsigned int`

Использование типа `unsigned int` в качестве типа свойства кажется достаточно разумным, но при более подробном и тщательном изучении становится очевидным, что он не подходит для большинства случаев. Рассмотрим сначала причины использования типа `unsigned int` в качестве типа свойства. Наиболее очевидной из них является ограничение вводимых значений только положительными значениями. Другая причина — необходимость использования максимально большого положительного значения. Кроме того, есть еще одна причина, которая подробно рассматривается ниже.

Для ограничения вводимых значений только положительными значениями вряд ли стоит использовать тип `unsigned int`. Действительно, значения такого свойства всегда будут положительны, но передаваемое этому свойству значение не обязательно может быть и отрицательным. При передаче отрицательного значения ошибка не возникает, а значение просто преобразуется к типу `unsigned int` с получением громадного значения. Не существует никакого способа определения этой нежелательной ситуации, а потому всегда существует вероятность ее появления. Для ограничения диапазона значений свойства только положительными значениями лучше всего использовать свойство типа `int` и инициировать исключительную ситуацию, если введенное значение меньше нуля. Кроме того, можно использовать подход, предложенный дли типа `short int`.

Вторая причина использования типа `unsigned int` — увеличение диапазона допустимых значений — вряд ли обоснована. Если максимальное значение 2147483647 недостаточно велико для данного свойства, то вряд ли максимальное значение 4294967295 также будет достаточно велико, потому что оно всего лишь в два раза больше. В большинстве случаев такие большие значение не используются. В противном случае следует рассмотреть возможность использования типа `__int64` (который не рассматривается здесь). Если внимательно изучить библиотеку VCL и ее опубликованные свойства, то можно заметить, что тип `unsigned int` используется для них крайне редко. Он применяется для свойства `Interval` (Пауза) класса `TTimer`. Использование типа `unsigned int` для этого свойства связано с той же проблемой, что и использование этого типа для любого другого свойства. Однако маловероятно, что в качестве нового значений этому свойству будет передано отрицательное значение. В противном случае пауза будет гораздо продолжительнее.

Большинство других свойств в библиотеке VCL имеет тип `unsigned int`, так как они представляют такие значения Windows API типа `DWORD`, как например дескрипторы. В этом случае тип `unsigned int` выбран для типа свойства, потому что он эквивалентен типу `DWORD` (т.е. длиной 4 байта и не имеет знака). Такова третья причина использования `unsigned int` в качестве типа свойства. В большинстве случаев можно избежать применения типа `unsigned int` в качестве типа, эквивалентного `DWORD` Windows API. Например, тип `DWORD` чаще всего используется как маска. В интерфейсе это лучше реализовать с помощью типа набора `Set`. Таким образом, могут быть изменены многие переменные. В некоторых случаях использование типа `unsigned int` оправданно, например при сохранении переменной (такой как дескриптор Windows), которая только считывается или присваивается, но не используется в выражениях. Особенно большое значение имеет исключение свойств типа `unsigned int` в выражениях. Дело в том, что при использовании переменной типа `unsigned` в выражении другие переменные также будут преобразованы в значения типа `unsigned` еще до оценки всего выражения. Если переменная имела отрицательное значение, результат этой оценки будет ошибочным. Это может привести к негативным последствиям. Это еще одна причина, по которой

не рекомендуется использовать unsigned int в качестве типа свойства. Следовательно, нужно уделить особое внимание предупреждениям компилятора типа Comparing signed and unsigned values (Сравнение знаковых и беззнаковых значений).

Старайтесь избегать использования типа unsigned int, если, конечно, возможно. В противном случае следует иметь в виду вероятность возникновения возможных побочных эффектов.

Во время создания этой книги в коде метода SetValue() редактора свойства TIntegerProperty была замечена ошибка. В нем неправильно определялись минимальное и максимальное допустимые значения для свойства типа unsigned int, а именно 0 и -1, соответственно. (Ошибка возникла в результате неправильного использования типа long int для хранения этих значений. Тип long int имеет знак и тот же размер, что и тип int, а максимальное значение типа unsigned int имеет ту же битовую структуру, что и значение -1 типа int.) Очевидно, что это неверно. Кроме того, из-за этого невозможно вводить значения типа unsigned int для свойств компонентов, реализованных на языке C++. Пакет EnhancedEditors из главы 10 содержит редактор свойств для типа unsigned int, который можно инсталлировать в IDE-среде для исправления этой ошибки.

Тип unsigned short

Все, сказанное о типах short int и unsigned int, относится и к unsigned short. Этот тип не следует использовать по упомянутым выше причинам.

Тип unsigned long

В C++Builder этот тип такой же, как unsigned int, поэтому в отношении его справедливы те же соображения, которые высказывались для типа unsigned int.

Тип char, unsigned char и signed char

Для типа char применимы те же рекомендации, что и для типа short int, т.е. его не следует использовать для ограничения диапазона вводимых значений. Если вы собираетесь использовать для этой цели тип char, то лучше вместо него использовать тип int. Свойство типа char следует использовать только в том случае, если вам нужна однобайтовая переменная или свойство должно быть символьным (как например, свойство PasswordChar класса TMaskEdit). Обратите внимание, что по умолчанию тип char имеет знак, но его можно использовать и без знака. Однако при использовании отрицательных значений или значений больше чем 127 могут возникнуть проблемы. Для типа unsigned char применимы те же соображения, что и для типа unsigned short int.

Во время создания этой книги существовало противоречие в поведении редактора свойства TCharProperty. Для свойств типа char, используемых в библиотеке VCL, допускаются значения в диапазоне от 0 до 255. Однако для пользовательских свойств типа char диапазон ограничен значениями от −128 до +127, потому что тип char является знаковым типом (по умолчанию). Кроме того, метод GetValue() не сохраняет знак введенного значения, а преобразует его к беззнаковому эквиваленту. Например, при вводе значения -1 оно будет отображено как #255, потому что эти значения имеют одинаковое битовое представление. Более корректно реализованные редакторы свойств для типов char, signed char и unsigned char содержатся в пакете EnhancedEditors в главе 10. При инсталляции они переопределят принятый по

умолчанию редактор свойств типа char. Для типа char можно вводить любые значения от -128 до 255, но сохранено будет только допустимое значение: если тип char имеет знак, то #255 = #-1; если не имеет знака — верно обратное. Для явно заданных типов signed char или unsigned char диапазон будет ограничен только корректными значениями.

Тип double, long double и float

При необходимости использования в интерфейсе компонента значений с плавающей запятой, прежде всего следует использовать тип double. Он занимает то же место среди типов значений с плавающей запятой, что и тип int среди целочисленных типов. Он предназначен для передачи в качестве подходящего аргумента любого допустимого значения с плавающей запятой.

Тип long double следует использовать только в тех случаях, когда необходима точность, в противном случае используйте тип double.

Применение типа float в интерфейсе компонента может привести к возникновению проблем, аналогичных тем, которые встречаются при использовании в интерфейсе типа short int. Значение типа double может быть без каких-либо предупреждений преобразовано к типу float, в результате чего может появиться некорректное значение свойства.

Заключительные замечания по поводу выбора типа свойства

В большинстве случаев в качестве типов свойств следует использовать int, char и double. Случаи, когда требуется использовать другие типы, возникают не очень часто. Особенно внимательно следует применять беззнаковые типы (например, при использовании типа unsigned int для представления свойств типа DWORD). Убедитесь в том, что такой поход является наиболее оптимальным. В этом разделе рассмотрены типы свойств, используемых в интерфейсе компонента. Более подробное описание способов использования разных фундаментальных типов можно найти в книге *Large-Scale C++ Software Design*, Lakos [1996].

Управление режимом использования компонента

Создавая компонент, следует тщательно продумать его функционирование в режиме выполнения и создания. В главе 9 уже описывались флаги ComponentState для определения режима использования компонента. Если компонент используется при создании приложения, то задается флаг csDesigning, как показано в следующем коде.

```
if(ComponentState.Contains(csDesigning))
{
   // Во время создания ...
}
else
{
   // Во время выполнения ...
}
```

Этот очень важно для некоторых компонентов. Рассмотрим, например, компонент захвата видеокадра с использованием Video Capture API-интерфейса, который упоминался выше в этой главе в разделе об использовании функций обратного вызова.

Одно из преимуществ управления режимом работы компонента, особенно взаимодействующего с аппаратным обеспечением, является то, что при создании приложения можно проверить корректность значений для данного аппаратного обеспечения, не компилируя код.

Упрощение таких экспериментов является одним из несомненных достоинств среды разработки C++Builder. В этих случаях действительно могут возникать различные проблемы. Возвращаясь к примеру с компонентом захвата видеокадра, проиллюстрируем одну такую проблему. Для захвата видеокадра необходимо создать окно захвата и подсоединиться к драйверу захвата. После этого можно приступать к чтению и записи доступных параметров захвата, которые предоставлены поставщиком аппаратного обеспечения. Эта информация о возможностях драйвера имеет огромное значение. Однако нужно учитывать следующую проблему: в определенный момент времени драйвер может быть связан только с одним окном захвата. Поэтому при подключении драйвера видеокарты для захвата видеокадра во время создания приложения последующая попытка запуска и отладки приложения с этим компонентом потерпит неудачу, так как драйвер уже связан с окном захвата. Эта на первый взгляд простая проблема имеет множество решений, которые зависят от выбранного способа функционирования интерфейса в режиме создания приложения. Если вам нужно иметь возможность подключения и просмотра значений, то следует предусмотреть способ отключения драйвера для запуска и отладки приложения. Для достижения этого не существует какого-то одного простого способа, поэтому здесь кратко описывается одно из возможных решений.

Иногда в режиме создания приложения вам необходимо только быть уверенным в корректности значения свойства, причем задавать его не нужно, достаточно знать, что оно может быть задано. Значение свойства в таком случае должно быть сохранено до следующей попытки его изменения. Это хранимое значение свойства затем используется для указания значения свойства во время выполнения приложения при потоковой передаче свойств компонента. При изменении значения достаточно подключиться к драйверу захвата видеокадра, для чего требуется получить подтверждение от драйвера. Следовательно, все остальное время он может оставаться неподключенным. В таком случае проверить режим работы компонента можно с помощью флага `csDesigning`. Если компонент используется в режиме создания приложения, то в исходном состоянии нам не потребуется подключаться к драйверу захвата видеокадра. Если значение свойства меняется (для этого требуется подтверждение со стороны драйвера), можно подключиться к драйверу, проверить значение, обновить значение свойства и отключиться. В Windows API для присвоения целого набора связанных свойств обычно используются большие структуры set. Такой же способ применяется и в Video Capture API. Таким образом, наиболее разумный подход заключается в создании в компоненте подходящей структуры и организации способа связанного изменения значений свойств на основе этого набора. Так как при каждом изменении значения отдельного свойства меняется весь набор значений, то имеет смысл изолировать такой код в методе `Update()`. Он может вызываться каждый раз при изменении связанного свойства. Вот как выглядит структура кода этого метода.

```
void __fastcall TComponentName::SetProperty(int NewValue)
{
   if(FValue != NewValue)
   {
      FValue = NewValue;
      Structure.SomeName = FValue;

      if(ComponentState.Contains(csDesigning))
      {
         // Подключение к драйверу
         UpdateValues();
         // Отключение от драйвера
      }
      else
      {
```

```
            UpdateValues();
        }
    }
}
```

В этом коде предполагается, что в компоненте объявлена структура `Structure`, а также то, что исходные значения свойств заданы в конструкторе компонента. Кроме того, при использовании компонента во время выполнения приложения он подключается к драйверу захвата видеокадра в конструкторе и отключается в деструкторе.

Определение режима работы компонента не является проблемой, так как относится к созданию самого компонента. Однако этому часто приходится уделять особое внимание, чтобы выяснить необходимость такого различия в поведении компонента.

Фрэймы

Библиотека VCL — вероятно, самый существенный элемент среды C++Builder. По сути, она является каркасом, на котором основана эта RAD-среда. Предлагая возможности для визуальной разработки, библиотека VCL существенно облегчает программирование управления пользовательским интерфейсом. Однако, как показано в предыдущих двух главах, этот комплимент не относится к процессу создания компонентов. Дело в том, что в библиотеке VCL не предусмотрен механизм визуальной разработки пользовательских компонентов. Действительно, с момента появления C++Builder версии 1.0 разработчики предлагали специалистам Borland создать такой инструмент. Создание фрэймов можно назвать первым шагом в этом направлении. Как будет показано в следующем разделе, с их помощью существенно упрощается процесс создания пользовательского компонента.

Что такое фрэйм

Стандартное окно верхнего уровня принято называть *формой* (*form*). Термин *фрэйм* (*frame*) используется для описания объекта класса `TFrame` или его наследников. Концептуально фрэйм можно считать дочерней формой, хотя на самом деле фрэйм гораздо ближе связан с окном просмотра, имеющим полосы прокрутки (ScrollBox). Рассмотрим этот вопрос более внимательно.

Класс `TFrame` является прямым наследником класса `TCustomFrame`, который предназначен только для публикации отдельных свойств класса `TCustomFrame` и не содержит кода реализации. Класс `TCustomFrame`, в свою очередь, является наследником класса `TScrollingWinControl`. Классы `TCustomForm` и `TScrollBox` являются наследниками класса `TScrollingWinControl`. Эта связь показана на рис. 11.1.

Класс `TScrollingWinControl`, прямой наследник класса `TWinControl`, расширяет родительский класс за счет введения горизонтальной и вертикальной полосы прокрутки. Класс `TCustomForm` расширяет класс `TScrollingWinControl` за счет поддержки тех характеристик, которые свойственны окнам верхнего уровня. Класс `TScrollBox` расширяет класс `TScrollingWinControl` только за счет свойства `BorderStyle`. Однако класс `TCustomFrame` не содержит никаких дополнительных свойств или функций по сравнению с родительским классом.

Короче говоря, фрэйм представляет собой нечто большее, чем просто объект `TScrollingWinControl`. Он является дочерним окном с возможностями прокрутки содержимого. Во время создания приложения фрэйм больше всего напоминает объект `TForm`. А во время выполнения приложения фрэйм больше похож на объект `TScrollBox`, у которого свойство `BorderStyle` имеет значение `bsNone`.

Класс TCustomFrame

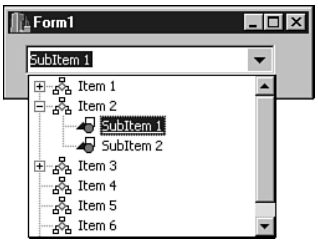


Рис. 11.1. Иерархия наследования классов TFrame, TForm и TScrollBox

Как уже говорилось выше, класс TFrame является наследником класса TCustomFrame и предназначен только для опубликования некоторых свойств своего родительского класса. Чтобы разобраться, что представляет собой первый из двух этих классов, рассмотрим кратко структуру класса TCustomFrame.

В конструкторе TCustomFrame свойству Width присваивается значение 320, а свойству Height — значение 240. Также внутри конструктора задаются следующие флаги состояния для свойства ControlState: csSetCaption, csAcceptsControls, csCaptureMouse, csClickEvents и csDoubleClicks. На самом деле для наследников класса TControl все остальные флаги состояния, кроме csAcceptsControls, задаются автоматически. Именно флаг csAcceptsControls указывает на использование объекта-фрэйма в качестве контейнера элементов управления. Кроме того, аналогично форме (или модулю данных) при создании приложения, класс TCustomFrame также может содержать невизуальные компоненты. Так как эти невизуальные компоненты должны быть переданы в потоке в файл .DFM вместе с объектом-фрэймом, то класс TCustomFrame расширяется (за счет переопределения) динамической функции DYNAMIC GetChildren(), в которой прямой вызов функции обратного вызова с возвращаемым типом TGetChildProc организован для каждого невизуального компонента. Обратите внимание, что такой же способ используется в классах TCustomForm и TDataModule.

Списки ActionLists поддерживаются в классе TCustomFrame с помощью закрытых функций AddActionList() и RemoveActionList(). Они предназначены для добавления и удаления объектов ActionList (содержащихся в классе-наследнике TFrame) в (или из) внутреннего массива ActionList родительской формы. Каждая из этих функций вызывается, соответственно, из переопределенных функций SetParent() (при изменении родительской формы) и Notification() (при создании или удалении ActionList).

Работа с фрэймами при создании приложения

При добавлении нового класса-наследника TFrame в проект во время создания приложения IDE-среда представляет экземпляр этого класса в окне конструктора форм Form Designer (объект TWinControlForm). Работа с ним осуществляется точно так же, как и при создании класса-наследника TForm. Например, можно управлять аспектами класса-наследника Tframe, изменяя его опубликованные свойства. Действительно, за исключением свойства TScrollBox::BorderStyle, свойства (включая типы событий) принимаемого по умолчанию наследника класса TFrame идентичны свойствам класса TScrollBox.

Управление классом-наследником TFrame с помощью конструктора форм позволяет учитывать изменения самого класса. Это идентично работе с классом-наследником TForm в конструкторе форм Form Designer. Например, при опускании нового компонента в форму во время создания приложения заголовочный файл класса-наследника TForm обновляется для отражения этого изменения (аналогичный результат можно получить при работе с конструктором модуля данных Data Module Designer). При изменении характеристики нашего класса-наследника TFrame в его конструкторе форм внесенные изменения будут отражены в заголовочном файле класса или в соответствующем файле .DFM. Это отличается от способа работы с большинством других компонентов, когда манипуляции во время создания приложения

влияют только на отдельный экземпляр класса компонента. Например, при опускании компонента-панели TPanel в форму во время создания приложения и изменении свойства Color этой панели нужно только изменить один экземпляр класса TPanel, а не сам класс. Это именно то, что нужно при использовании компонентов: во многих ситуациях не гарантируется создание класса пользовательского компонента для каждого экземпляра отдельного компонента. Например нет никакой необходимости создавать класс TRedPanel только для использования окрашенного в красный цвет варианта панели TPanel.

К счастью, в IDE-среде C++Builder 5 предусмотрено несколько механизмов, которые позволяют во время создания приложения управлять отдельным экземпляром класса-наследника TFrame. Например можно выбрать пиктограмму Frames во вкладке Standard палитры компонентов Component Palette и создать экземпляр класса-наследника TFrame для такого элемента управления контейнера, как, например, форма, панель или даже другой фрэйм. Альтернативный метод заключается в добавлении класса-наследника TFrame в палитру компонентов Component Palette в качестве шаблона компонента. Последняя задача может быть выполнена с помощью выбора команды Add To Palette из контекстного меню, которое отображается на экране после щелчка правой кнопкой мыши на фрэйме в его конструкторе форм. Третий подход заключается в замене класса-наследника TFrame в пакете компонента. Позже мы обсудим этот вопрос.

Тут важно отметить, что любые изменения, вносимые во время создания приложения в экземпляр класса-наследника TFrame, влияют только на этот отдельный экземпляр. С другой стороны, манипуляции с классом-наследником TFrame во время создания приложения будут отражены во всех экземплярах этого класса. В C++Builder 5 для различия этих двух парадигм вводится флаг csDesignInstance в свойстве ComponentState. Любые изменения (кроме удаления компонента) в отдельном экземпляре класса могут быть сделаны с помощью простых манипуляций с самим экземпляром. На самом деле, IDE-среда устроена немного сложнее. Если свойство изменено в каком-либо экземпляре класса-наследника TFrame, то последующие изменения этого свойства того же класса не будет отражено в измененном экземпляре. Например, при изменении класса-наследника TFrame с помощью размещения компонента TPanel в фрэйме в окне конструктора форм все экземпляры класса-наследника TFrame будут отражены в новой панели. Однако при изменении свойства Color этой панели в отдельном экземпляре нашего фрэйма и последующем вторичном изменении свойства Color соответствующей панели в классе-наследнике TFrame новое значение свойства Color панели, заданное в экземпляре фрэйма, не будет изменено вторично. Иначе говоря, IDE-среда не будет отменять изменения, которые были предварительно внесены в отдельный экземпляр класса фрэйма.

Работа с фрэймами при выполнении приложения

В работе с экземпляром класса-наследника TFrame при выполнении приложения нет ничего особенного, так как это всего лишь другой вид наследника класса TWinControl. Более того, как уже говорилось, фрэйм при выполнении приложения напоминает окно просмотра с полосами прокрутки (ScrollBox). Действительно, без усовершенствований времени создания, которые представлены в IDE-среде и классе TCustomFrame, вряд ли была бы замечена разница в использовании фрэйма вместо окна просмотра.

Однако с точки зрения управления ресурсами фрэйм имеет более сложную структуру, чем большинство других элементов управления. Например, при размещении компонента TImage в форме, загрузке в компонент TImage изображения размером 1 Мбайт и создании десятка копий компонента TImage размер приложения существенно увеличится. Но если создать новый фрэйм, опустить в него компонент TImage в режиме конструктора форм (иначе говоря, изменить класс-наследник TFrame), загрузить в него изображение размером 1 Мбайт и создать

десяток экземпляров этого фрэйма вместо отдельных объектов TImage, то размер приложения практически не увеличится. Дело в том, что все экземпляры фрэйма будут совместно использовать только одну копию откомпилированного изображения. И наоборот, при использовании десятка копий компонента TImage десяток копий этого изображения будут откомпилированы в виде ресурсов приложения.

Создание класса-наследника TFrame

Ясно, что создание класса-наследника TForm (или TDataModule) значительно упрощается при использовании IDE-среды, особенно при использовании инструментов визуальной разработки. По этой же причине класс-наследник TFrame создается так же просто. И наоборот, для создания класса-наследника TScrollBox требуется вручную создавать соответствующий код.

Очень часто новый компонент создается исключительно для поддержки некоторой функциональной возможности. Наиболее распространенный пример — какие-либо действия, выполняемые во время создания приложения, например расширение виртуальной функции CreateParams() (этим объясняется огромное количество вызовов члена-функции RecreateWnd() в библиотеке VCL). В процессе работы с объектом-фрэймом функция CreateParams() легкодоступна просто потому, что IDE-среда автоматически создает объявление класса-наследника. Это справедливо и для создания пользовательских компонентов, а значит при работе с фрэймами не придется упаковывать и регистрировать класс компонента. Это существенно упрощает процесс отладки. Рассмотрим следующий пример, который иллюстрирует это.

Практический пример: применение фрэймов для создания всплывающего окна

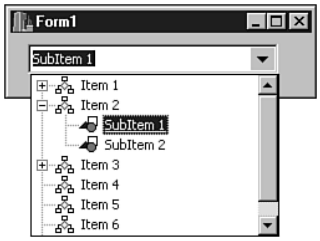


Рис. 11.2. Использование фрэйма для создания всплывающего окна с элементом управления TreeView

Во многих современных приложениях используются временные всплывающие окна, которые содержат разные типы элементов управления. Например, многие кнопки панелей инструментов Microsoft Office 2000 содержат стандартные всплывающие меню, на самом деле являясь пользовательскими окнами без заголовка. Еще один распространенный пример такого окна представляет собой раскрывающийся список окна прокрутки со списком позиций для выбора. Попробуем создать собственную версию раскрывающегося окна с помощью класса-наследника TCustomFrame (рис. 11.2). Интерфейс этого класса приводится в листинге 11.13, а полная реализация этого класса содержится в проекте Project1.bpr в каталоге PopupFrame на прилагаемом к книге компакт-диске.

Листинг 11.13. Файл TPopupFrame.h для класса-наследника TFrame

```
//---------------------------------------------------------------//
#ifndef PopupFrameUnitH
#define PopupFrameUnitH
//---------------------------------------------------------------//
#include <Classes.hpp>
#include <Controls.hpp>
#include <ComCtrls.hpp>
```

```
//-----------------------------------------------------------//

class TPopupFrame : public TFrame
{
__published:
   TTreeView *TreeView1;
   void __fastcall TreeView1MouseMove(TObject *Sender,
                                      TShiftState Shift,
                                      int X, int Y);

   void __fastcall TreeView1MouseUp(TObject *Sender,
                                    TMouseButton Button,
                                    TShiftState Shift,
                                    int X, int Y);

private:
   TNotifyEvent OnCloseUp_;
   MESSAGE void __fastcall CMMouseEnter(TMessage& AMsg)
   {
      ReleaseCapture();
   }
   MESSAGE void __fastcall CMMouseLeave(TMessage& AMsg)
   {
      if (Visible)SetCapture(TreeView1->Handle);
   }

protected:
   virtual void __fastcall CreateParams(TCreateParams& AParams)
   {
      TFrame::CreateParams(AParams);
      AParams.Style = AParams.Style | WS_BORDER;
      AParams.ExStyle = AParams.ExStyle | WS_EX_PALETTEWINDOW;
   }
   virtual void __fastcall CreateWnd()
   {
      TFrame::CreateWnd();
      ::SetParent(Handle,GetDesktopWindow());
      SNDMSG(TreeView1->Handle,WM_SETFOCUS,0,0);
   }
   DYNAMIC void __fastcall VisibleChanging()
   {
      TFrame::VisibleChanging();
      if (Visible)ReleaseCapture();
      else SetCapture(TreeView1->Handle);
   }

public:
   BEGIN_MESSAGE_MAP
   VCL_MESSAGE_HANDLER(CM_MOUSEENTER, TMessage, CMMouseEnter)
   VCL_MESSAGE_HANDLER(CM_MOUSELEAVE, TMessage, CMMouseLeave)
```

```
END_MESSAGE_MAP(TFrame)

public:
   __fastcall TPopupFrame(TComponent* AOwner);
   __property TNotifyEvent OnCloseUp =
      {read = OnCloseUp_, write = OnCloseUp_};
};
//---------------------------------------------------------//
extern PACKAGE TPopupFrame *PopupFrame;
//---------------------------------------------------------//
#endif
```

Всплывающее окно на само деле является дочерним объектом окна рабочего стола (т.е. для него задан бит стиля WS_CHILD). Оно создается на основе комбинации битов расширенного стиля WS_EX_TOOLWINDOW и WS_EX_TOPMOST (представленных WS_EX_PALETTEWINDOW). Бит WS_CHILD при активизации всплывающего окна предотвращает потерю активности самой формы. Бит WS_EX_PALETTEWINDOW предотвращает перекрытие всплывающего окна другими окнами (EX_TOPMOST) и появление его в списке активных приложений, который отображается на экране при нажатии комбинации клавиш <Alt+Tab> (EX_TOOLWINDOW).

Для работы с битами стиля следует переопределить виртуальную функцию CreateParams().

```
virtual void __fastcall CreateParams(TCreateParams& AParams)
{
   TFrame::CreateParams(AParams);
   AParams.Style = AParams.Style | WS_BORDER;
   AParams.ExStyle = AParams.ExStyle | WS_EX_PALETTEWINDOW;
}
```

Сначала вызывается член-функция CreateParams() родительского класса, чтобы класс TWinControl мог добавить (вверх по иерархической структуре классов) бит WS_CHILD. К члену данных Style аргумента типа TCreateParams добавляется бит WS_BORDER, а к члену данных ExStyle — бит WS_EX_PALETTEWINDOW.

Для назначения окна рабочего стола в качестве родителя всплывающего окна следует переопределить виртуальную функцию CreateWnd().

```
virtual void __fastcall CreateWnd()
{
   TFrame::CreateWnd();
   :: SetParent(Handle, GetDesktopWindow());
   SNDMSG(TreeView1->Handle, WM_SETFOCUS, 0, 0);
}
```

Сначала вызывается функция CreateWnd() родительского класса, чтобы класс TWinControl мог зарегистрировать (вверх по иерархической структуре классов) этот класс в Windows и создать окно с помощью функции CreateWindowHandle(). Для указания окна рабочего стола в качестве родителя всплывающего окна используется API-функция SetParent() вместе с API-функцией GetDesktopWindow(). Это действие гарантирует, что содержимое всплывающего окна не будет обрезано по границам его обычного родителя (т.е. формы). Кроме того, дочернему элементу управления TreeView посылается сообщение WM_SETFOCUS для создания иллюзии получения фокуса ввода с клавиатуры.

Помимо управления стилем и указания родителя всплывающего окна, следует организовать связанные с мышью события, особенно те, которые приводят к закрытию всплывающего окна. Для этого необходимо использовать сообщения `CM_MOUSENTER` и `CM_MOUSELEAVE` библиотеки VCL и расширить виртуальную функцию `VisibleChanging()`. С этой целью можно воспользоваться уже готовыми событиями `TTreeView::OnMouseMove` и `TTreeView::OnMouseUp`. Соответствующим образом в элементе управления TreeView предоставляется и отменяется перехват сообщений от мыши так, что обработчик события `OnMouseUp` может быть запущен, даже если кнопка мыши отпущена при положении курсора вне границ всплывающего окна. Точнее, если всплывающее окно не перехватило сообщение от мыши, сообщения `WM_*BUTTONUP` не будут посылаться при размещении курсора вне клиентской части всплывающего окна.

На самом деле каждый шаг по созданию класса всплывающего окна может быть выполнен и без использования фрэймов. Однако цель этого примера состоит в демонстрации применения фрэймов для значительного упрощения процесса создания такого класса. Например нам не потребуется вручную объявлять и присваивать соответствующие функции событиям `OnMouseMove` и `OnMouseUp` элемента управления TreeView. Эта задача становится очень обременительной, если компонент содержит несколько разных дочерних компонентов, а каждый из них имеет большое количество обработчиков событий. Несколько более сложный пример класса-наследника `TFrame`, а именно класса `TMCIWndFrame`, можно найти на прилагаемом к книге компакт-диске в проекте `Proj_VideoDemo.bpr` в каталоге `VideoDemo` для главы 25 о методах работы с мультимедиа-технологиями.

Наконец, следует отметить, что виртуальная или динамическая функции могут переопределяться только при выполнении приложения. Например, в листинге 11.5 переопределяется динамическая функция `VisibleChanging()`. В этом случае наша переопределенная версия не будет вызвана при создании приложения. Это не относится к тому случаю, когда компонент размещается в пакете времени создания и регистрируется для использования в IDE-среде. Действительно, если класс-наследник `TFrame` регистрируется как компонент времени создания, то может потребоваться использовать флаг `csDesigning` свойства `ComponentState`.

Наследование от класса-наследника `TFrame`

С наследниками класса-наследника `TFrame` во время создания приложения также можно выполнять визуальные операции. Например, при создании наследника нашего класса `TPopupFrame` он также будет открыт в собственном окне конструктора форм с отображением атрибутов его родительского класса. Упомянутые выше правила наследования также применимы здесь, т.е. последующие изменения родительского класса при создании приложения будут отражены в наследнике, но не наоборот.

Однако IDE-среда не очень подходит для визуального управления наследованием фрэймов. Т.е. эту простую процедуру вряд ли можно назвать дружественной для пользователя. Для создания наследника класса-наследника `TFrame`, который может быть спроектирован визуальными средствами, следует выполнить следующие действия.

1. Создать в проекте родительский класс-наследник `TFrame` (например, `TFrame2`).
2. Выбрать команду меню File⇨New Frame для создания в проекте нового класса-наследника `TFrame` (например, `TFrame3`).
3. Отредактировать код в заголовочном файле и файле с исходным кодом для нового класса-наследника `TFrame` (`Unit3.h`, `Unit3.cpp`), заменяя все случаи исходного родительского класса (`TFrame`) именем нового родительского класса (например, `TFrame2`).
4. Отредактировать файл `.DFM` нового класса-наследника, заменяя ключевое слово `object` ключевым словом `inherited`.

Повторное использование фрэймов

Так же как и формы, фрэймы можно включать в репозиторий объектов для повторного использования в будущих проектах. Фрэймы также можно распространять среди других разработчиков, передавая им исходный код (интерфейс и код реализации) и файл `.DFM`. Более того, как уже упоминалось, класс-наследник `TFrame` может быть размещен в пакете времени создания приложения и зарегистрирован в IDE-среде, как и любой другой класс пользовательского компонента. Например, для регистрации нового класса `TMyFrame` наследника класса `TFrame` достаточно использовать макрос `PACKAGE` в соответствующих местах, а потом определить знакомые функции `ValidCtrCheck()` и `Register()` в файле с исходным кодом класса.

```
static inline void ValidCtrCheck(TMyFrame*)
{
   new TMyFrame(NULL);
}

namespace Myframeunit
{
   void __fastcall PACKAGE Register()
   {
      TComponentClass classes[1] == {__classid(TMyFrame)};
      RegisterComponents("Samples", classes, 0);
   }
}
```

Однако при работе с фрэймом в составе пакета для создания приложения следует иметь в виду следующие возможные трудности. Во-первых, при использовании редактора пакетов IDE-среда потребует открытия файла `.DFM` класса-наследника `TFrame` в таком виде, как если бы он был стандартным объектом-формой. Соответственно, потребуется изменить содержимое файла .DFM. Например, для обычного класса-наследника `TFrame` оно будет иметь следующий вид.

```
object Frame2: TFrame2
   Left = 0
   Top = 0
   Width = 320
   Height = 240
   TabOrder = 0
end
```

С другой стороны, файл .DFM для того же класса-наследника `TFrame`, но исправленного IDE-средой будет выглядеть иначе.

```
object Frame2: TFrame2
   Left = 0
   Top = 0
   Width = 320
   Height = 240
   Color = clBtnFace
   Font.Charset = DEFAULT_CHARSET
   Font.Color = clWindowText
   Font.Height = -11
   Font.Name = 'MS Sans Serif'
   Font.Style = []
   OldCreateOrder = True
```

```
   PixelsPerInch = 96
   TextHeight = 13
end
```

Помимо удаления строки для свойства `TabOrder`, IDE-среда добавила строки, которые относятся к наследникам класса `TForm`, но не к наследникам класса `TFrame` (например, `OldCreateOrder`). Более того, после регистрации класса-наследника `TFrame` последующие попытки открытия файла с исходным кодом (и его DFM-файла) приведут к упомянутому выше изменению содержимого DFM-файла (независимо от использования редактора пакетов Package Editor). Обратите внимание на то, что если DFM-файл был непреднамеренно изменен, его можно вернуть к прежнему состоянию, добавив свойство `TabOrder`, а потом удалив все ненужные строки.

Еще один побочный эффект регистрации класса-наследника `TFrame` заключается в том, что содержащимися в нем компонентами нельзя автоматически управлять при создании приложения. Например, во время создания приложения при включении шаблона компонента или добавлении его в проект содержащимися в экземпляре класса-наследника `TFrame` компонентами можно манипулировать по отдельности. Однако при работе с экземпляром времени создания приложения зарегистрированного класса-наследника `TFrame` эти компоненты уже не будут доступны, и придется применять другие меры (например, отдельные редакторы свойств).

Заключительные замечания по поводу применения фрэймов

Общеизвестно, что функциональности времени создания приложения, обеспечиваемой фрэймами, недостаточно для по-настоящему визуальной разработки. На этом этапе фрэймы, вероятно, будут более полезны для управления группами элементов управления, а не для создания сложных компонентов. Действительно, для эффективного использования системных ресурсов вряд ли стоит применять класс-наследник `TFrame`. Напомним, что класс `TFrame` является наследником класса `TWinControl`, поэтому для каждого его экземпляра необходим отдельный дескриптор окна. Тем не менее фрэймы представляют собой шаг в верном направлении, а сама эта концепция представляется достаточно перспективной. В следующих версиях C++Builder она, несомненно, будет усовершенствована.

Распространение компонентов

Одним из наиболее важных вопросов в работе создателя компонентов является распространение компонентов. Эта казалось бы простая задача насыщена многими подвохами. В этом разделе даются рекомендации, позволяющие сократить до минимума вероятность возникновения этих проблем. Мы начнем обсуждение этого вопроса с рассмотрения структуры пакетов, которые содержат компоненты. При разработке пользовательских компонентов можно создать три типа пакетов: времени создания, времени выполнения, а также двойного применения пакета — времени создания и выполнения. При распространении компонентов всегда следует передавать пакеты двух типов, т.е. времени создания и времени выполнения, даже если они и не содержат каких либо пользовательских свойств или редакторов. Если процесс создания одного или нескольких компонентов еще продолжается, то разумней и удобней распространять пакет двойного применения. Мы будем рассматривать использование пары пакетов (времени выполнения и времени создания), хотя для сравнения будет упоминаться и двойственный пакет. Кроме того, здесь также рассматривается распространение пакета времени создания. Помимо выбора структуры пакета, также нужно выбрать удачное

соглашение об именах для модулей внутри пакета и самих компонентов. Также следует решить, стоит ли создавать разные компоненты для использования их с разными версиями компилятора. Эти и другие вопросы подробно рассматриваются в следующих разделах.

Упаковка компонентов

После проверки работоспособности компонентов следует их корректно упаковать. Наиболее корректный подход основан на разделении кода времени создания и времени выполнения. Правильный способ с использованием пакетов основан на создании как минимум двух пакетов: времени выполнения и времени создания. Для этого нужно выполнить следующие действия.

1. Создать пакет времени выполнения, содержащий *только* исходный код одного или нескольких компонентов. Не включать в него код регистрации или код для интерфейса времени создания (например, код редакторов свойств и компонентов).
2. Создать пакет времени создания, содержащий *только* код регистрации одного или нескольких компонентов и по желанию код интерфейса времени создания. Не включать в него исходный код одного или нескольких компонентов, а добавить вместо него в список `Requires` пакета библиотеку импорта (файл `.bpi`) для одного или нескольких компонентов пакета времени выполнения.

Пакет времени создания инсталлируется в IDE-среде. На рис. 11.3 показана взаимосвязь между этими двумя пакетами.

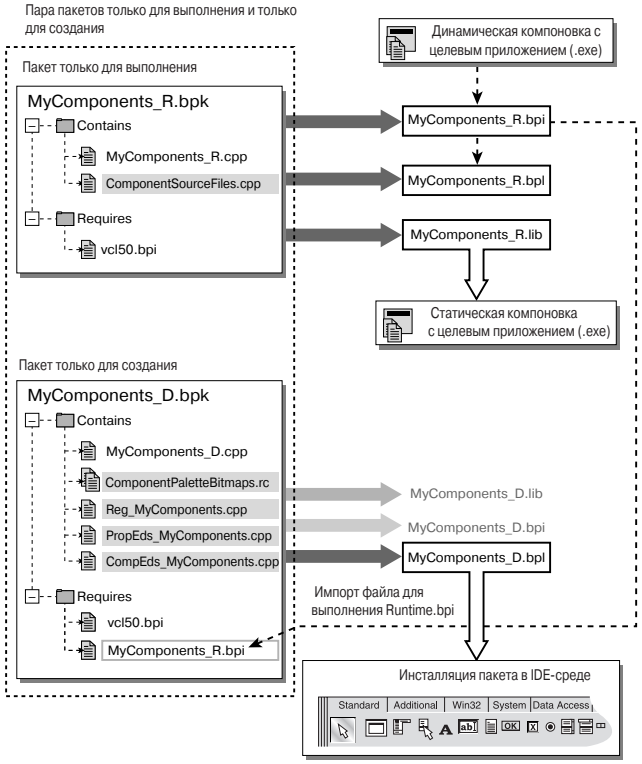


Рис. 11.3. Схема взаимодействия пакетов для выполнения и для создания приложения

Как показано на рис. 11.3, пакет времени создания используется только для создания файла `.bpl` для инсталляции его в IDE-среде. Пакет времени выполнения используется для создания файлов, которые будут использованы приложением. В действительности один или несколько пакетов времени выполнения могут использоваться вместе с одним пакетом времени создания (иначе говоря, могут быть представлены в разделе `Requires` пакета времени создания). Следовательно, весь код можно полностью удалить из пакета времени создания, создавая на его основе пакет регистрации.

Для сравнения пакет двойного применения для создания/выполнения показан на рис. 11.4.

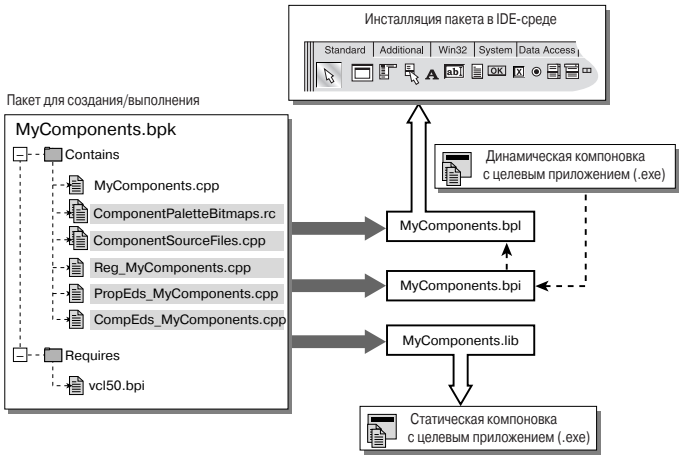


Рис. 11.4. Структура пакета двойного применения для создания/выполнения

Ясно, что код, необходимый для регистрации, а также для редакторов свойств и компонентов, будет без особой надобности включен в приложения, где используется двойственный пакет, как показано на рис. 11.4. Для простых пакетов без редакторов свойств и компонентов это может быть вполне приемлемо. Причем тестирование одного или нескольких компонентов с помощью двойственного пакета может оказаться еще более удобным.

Приступим теперь к рассмотрению многопакетной модели, простейшим вариантом которой является пара пакетов времени выполнения/создания, показанная на рис. 11.3 (другие возможности будут кратко описаны в конце этого раздела). Рассмотрим теперь этот подход более подробно.

Пакет времени выполнения создается первым. Для указания его как пакета времени выполнения установите флажок Runtime Only во вкладке Description диалогового окна параметров компонента Options. Этот пакет должен содержать только исходный код одного или нескольких компонентов. Процесс создания компонентов подробно описывается в главе 9, а также выше в этой главе. Предположим, что код компонента уже создан и протестирован, а потому может быть включен в пакет (на рис. 11.3 он показан в виде файла `ComponentSourceFiles.cpp`, включенного в раздел `Contains`). Необходимые этому компоненту библиотеки импорта (файлы `bpi`) следует включить в список `Requires`. При этом следует включить только те библиотеки импорта, которые нужны для успешной сборки. Напомним, что пакеты, упомянутые в списке `Requires`, компонуются во время компиляции приложения, с которыми они будут использоваться. После успешной сборки пакета времени выполнения будут получены три файла (если только это действие не будет отменено в результате снятия флажка Generate .lib File во вкладке Linker диалогового окна параметров компонента

Options): bpl-файл, bpi-файл, а также lib-файл. Убедитесь в их наличии, потому что все они потребуются в дальнейшем (см. файлы MyComponents_R.bpl, MyComponents_R.bpi и MyComponents_R.lib на рис. 11.3). Lib-файл нужен не всегда, а только в тех случаях, когда компоненты должны быть статически скомпонованы с приложением.

Теперь, после создания пакета времени выполнения и файла импорта, можно приступить к работе над пакетом времени создания. Этот пакет будет содержать код регистрации и код редакторов свойств и компонентов, которые необходимы вашим компонентам. На рис. 11.3 это модули Reg_MyComponents.cpp, PropEd_MyComponents.cpp и CompEd_MyComponents.cpp, соответственно, которые располагаются в разделе Contains. Раздел Requires включает библиотеку импорта пакета времени выполнения (файл MyComponents_R.bpi на рис. 11.3). Предположим, что код всех редакторов свойств и компонентов уже создан (подробное описание этого процесса можно найти в главе 10), и нам остается только написать код регистрации. Комбинируя информацию о создании функции Register() из глав 9 и 10, можно получить общую структуру, показанную в листинге 11.14.

Листинг 11.14. Общий вид функции Register() в модуле регистрации

```
//---------------------------------------------------------------//
#include <vcl.h>
#pragma hdrstop
//---------------------------------------------------------------//
#include "Reg_MyComponents.h"
#include "PropEd_MyComponents.h"
#include "CompEd_MyComponents.h"

#include "ЗаголовочныеФайлыКомпонентов.h"
//---------------------------------------------------------------//
#pragma package(smart_init)
//---------------------------------------------------------------//

namespace Reg_mycomponents
{
   void __fastcall PACKAGE Register()
   {
      // Регистрация компонентов пакета

      // Например...
      RegisterComponents("SomePalettePage",
                         OPENARRAY(TMetaClass*,
                                   (__classid(TComponent1),
                                    __classid(TComponent2),
                                    __classid(TComponent3))));

      // Регистрация редакторов свойств пакета.
      // См. главу 10 с примерами такого кода.

      // Регистрация редакторов компонентов пакета.
      // См. главу 10 с примерами такого кода.
```

```
      // Регистрация фильтров свойств пакета.
      // См. главу 10 с примерами такого кода.

   }
}
//---------------------------------------------------------//
```

При сборке этого пакета будут созданы три файла, хотя в данный момент bpi-файлы и lib-файлы не имеют большого значения. Наибольший интерес представляет bpl-файл (файл `MyComponents_D.bpl` на рис. 11.3). Это библиотека пакета, которую IDE-среда использует для организации доступа к компонентам при создании приложения.

При создании пакетов следует проверить исходный код (Edit⇨Option⇨Source) и удалить ненужные строки в файле библиотеки в разделах `<LIBRARIES>` и `<SPARELIBS>`. В противном случае пользователи вашего пакета будут безуспешно пытаться найти его lib-файл. Даже если этот файл не требуется для корректного функционирования пакета, пользователям не удастся использовать пакет без него, если в пакете имеется ссылка на этот файл.

На рис. 11.5 показаны файлы, которые должны распространяться для каждого используемого пакета.

Помимо упомянутых выше файлов (`.bpl`, `.bpi` и `.lib`), также следует распространять заголовочные файлы (`.h`) для каждого модуля, упомянутого в разделе `Contains` пакета выполнения. Кроме того, следует распространять заголовочные файлы модулей, используемых в пакете времени создания (за исключением заголовочного файла самого модуля регистрации, который обычно пуст и не используется). Однако обычно приходится распространять также исходный код модуля регистрации, чтобы пользователи компонента могли проследить за изменениями, которые вносятся в IDE-среду. Обратите внимание, что rc-файлы показаны как файлы ресурсов, которые используются для хранения изображений пиктограмм в палитре компонентов Component Palette. Для этой цели также могут использоваться res-файлы или dcr-файлы. Этот вопрос более подробно рассматривается ниже в этой главе в разделе о создании изображений пиктограмм в палитре компонентов. Учтите также, что при распространении пользовательских редакторов свойств и компонентов можно дать пользователю возможность создавать на их основе новые производные редакторы.

Как уже говорилось в главе 2 в разделе, посвященном использованию пакетов, lib-файл пакета по сути представляет собой набор объектных файлов (.obj) модуля пакета. Распространяя lib-файл пакета времени выполнения (который содержит исходный код одного или нескольких компонентов) либо объектные файлы одного или нескольких компонентов, можно по желанию пользователя организовать их статическую компоновку с приложением. Lib-файл обладает тем преимуществом, что все объектные файлы уже содержатся в одном файле, что упрощает их сопровождение.

Размещение распространяемых файлов

Место размещения распространяемых файлов на компьютере пользователя обуславливает простоту их использования. Применяя для этого используемые по умолчанию каталоги C++Builder (рис. 11.6), вы избавите пользователя от необходимости редактировать параметры каталогов проекта. Большинство пользователей предпочитают не размещать компоненты сторонних разработчиков в этих каталогах. Однако знание структуры каталогов C++Builder может оказаться очень полезным.

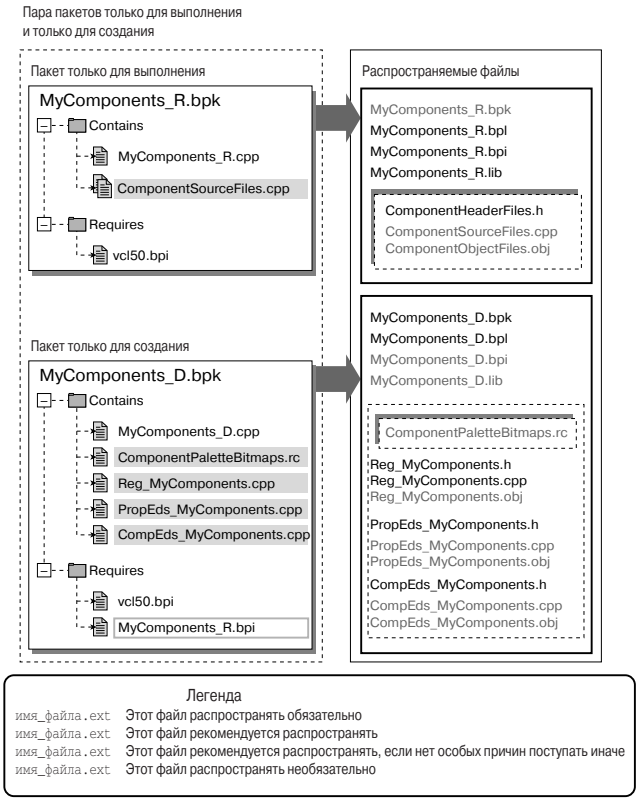


Рис. 11.5. Файлы, необходимые для распространения пакета

Рис. 11.6. Структура используемых по умолчанию каталогов C++Builder

Обратите внимание на использование соглашения об именах на рис. 11.3–11.5. Кроме того, здесь не показаны модули с исходным кодом, поскольку их расположение учитывается только при компиляции пакета. Простейшим альтернативным вариантом размещения всех файлов одного пакета является размещение их в одном каталоге, за исключением библиотеки времени выполнения (.bpl), которая может находиться в каталоге `Windows\System` или эквивалентном ему каталоге. Это значительно упрощает добавление путей в параметрах установки каталогов проекта. Компоновщик должен уметь найти bpi-файлы и lib-файлы пакета времени выполнения (при условии использования пары пакетов времени выполнения/времени создания), поэтому каталог с ними должен быть включен в состав глобального параметра Library Path (Tools⇨Environment Options⇨Library). Его также можно изменить, редактируя ключ реестра `HKEY_CURRENT_USER\Software\Borland\C++Builder\5.0\Library\Search Path`, где 5.0 — номер версии C++Builder (он может быть равен 1.0, 3.0 или 4.0). Напомним, что, если каталог (или несколько каталогов) с вашими компонентами находится за пределами основного каталога C++Builder, то строку `$(BCB)` можно использовать для представления этого каталога во всех нужных путях.

Именование пакетов и модулей в них

Именование пакетов и модулей имеет очень большое значение для подготовки компонентов к распространению. Предполагая, что для упаковки компонентов используется пара пакетов времени выполнения/создания (или аналогичная модель), необходимо именовать пакеты времени выполнения таким образом, чтобы их можно было отличить от пакетов времени создания. В имена пакетов времени выполнения обычно добавляют суффикс `_R` или `_RTP`, а в имена пакетов только времени создания — суффикс `_D` или `_DTP`. Символ подчеркивания используется по желанию (на рис. 11.3–11.5 он показан только для придания большей наглядности символам `D` и `R`).

Кроме того, в имя пакета нужно включить номер версии библиотеки VCL, которая использовалась для создания пакета. Для пакетов, созданных с помощью C++Builder версии 5, этот номер будет равен `50` (таков номер версии библиотеки VCL в C++Builder 5). Это позволяет легко определять номер версии среды C++Builder, для которой предназначен пакет. В именах пакетов на рис. 11.3–11.5 номер версии не показан только для удобства изложения материала, но в реальных пакетах его обязательно следует указать. Например, в именах пакетов времени выполнения можно добавить суффикс `_R50`, а в именах пакетов времени создания — суффикс `_D50`. Для пакетов библиотеки VCL принято соглашение о замене символа V из аббревиатуры VCL символом D (с получением строки DCL) в именах пакетов только времени создания. В обоих случаях номер версии добавляется в конце имени пакета. Например, имена пакетов для компонентов доступа к данным и компонентов управления данными будут иметь вид `VCLDB50.bpl` и `DCLDB50.bpl` для пакетов времени выполнения и пакетов времени создания, соответственно.

Кроме отражения в именах пакетов режима их использования (при создании или выполнении) и номера версии библиотеки VCL, для которой они предназначены, имена должны быть уникальными. Нельзя инсталлировать или использовать пакет, имя которого совпадает с именем уже существующего пакета.

Именование модулей в пакете имеет такое же большое значение, как и именование самого пакета. Следует помнить, что модули, экспортируемые из пакета, *должны быть уникальными* в рамках всего приложения, где они используются (нельзя использовать модули с одинаковыми именами в нескольких пакетах, которые одновременно используются приложением), включая IDE-среду.

Тут следует рассмотреть два отдельных случая именования. Во-первых, именование модулей, которые обычно входят в состав пакета времени создания (модули с кодом реги-

страции и модули с редакторами свойств и компонентов). Во-вторых, именование модулей, которые обычно входят в состав пакета времени выполнения (модули с исходным кодом компонентов).

В имена модулей пакета времени создания следует включить имя пакета. Это гарантирует уникальность имени модуля. Например, модуль, содержащий код регистрации, может иметь имя `Registration_PackageName.cpp`. По вашему усмотрению можно изменить порядок слов, заменить слово `Registration` словом `Reg` (и т.д.), а также вставить символы подчеркивания. Самое главное при этом — добиться понятности и уникальности имени модуля. Так как имя пакета уникально, его включение в имя модуля служит гарантией уникальности последнего. Следовательно, пакет времени создания с именем `NewComponentsD50` может содержать модули со следующими именами.

```
Reg_NewComponentsD50
RegNewComponentsD50
Registration_NewComponentsD50
RegistrationNewComponentsD50
```

Вариантов именования может быть очень много, но лучше всего выбрать какой-то один из них и строго придерживаться его. Требование уникальности имени исключает очевидные варианты выбора имен для модулей, содержащих редакторы свойств и компонентов. Например, имена `PropertyEditors.cpp` и `ComponentEditors.cpp` — пример крайне неудачного именования. Простейший способ, гарантирующий уникальность имен, был приведен выше: добавить имя пакета в имя модуля. Однако, наряду с этим вариантом, существует еще один альтернативный способ уникального именования модулей, используемых для редакторов свойств и компонентов. Вместо обычной директивы `#pragma package(smart_init)` для этих модулей следует использовать директиву `#pragma package(smart_init, weak)`. В результате это модули будут слабо запакованы в пакет времени создания. На самом деле имя модуля в этом случае уже не будет иметь никакого значения, потому что сам модуль уже не добавляется в состав пакета времени создания. Вместо этого по мере необходимости осуществляется непосредственный доступ к коду модуля. Никаких конфликтов возникать не будет при условии уникальности имен классов редакторов свойств и компонентов.

Второй случай связан с именованием модулей пакета времени выполнения. Если каждый модуль используется для хранения отдельного компонента, то имеет смысл в качестве имени модуля использовать имя компонента без начального символа T. Так как имена компонентов должны быть уникальными (см. следующий раздел), то применение такого соглашения практически гарантирует уникальность имени модуля. Для модулей, содержащих несколько компонентов, разумным способом именования является выбор такого имени, которое бы отражало весь набор компонентов модуля с добавлением имени пакета, т.е. аналогично способу именования модулей пакетов времени создания. Вероятность совпадения имен будет практически исключена, если строго придерживаться этих рекомендаций.

Именование компонентов

Выбор подходящего имени компонента имеет огромное значение. Во-первых, имя компонента должно быть уникальным, а во-вторых, оно должно отражать назначение компонента.

Задание уникального имени представляет собой достаточно сложную задачу. Существует много компонентов, решающих одну и ту же задачу, например инкапсулирующих работу последовательного порта компьютера. Для решения этой задачи разработчиками создано исключительно большое количество вариантов компонента `TComport`. Поэтому разработчики

очень серьезно относятся к проблеме распространения своих компонентов, обычно прибавляя подпись к имени компонента в виде инициалов после символа T и перед именем компонента. Их принято задавать в верхнем или нижнем регистре, что позволяет визуально игнорировать их. Некоторые разработчики считают, что буквы в верхнем регистре визуально проще игнорировать, потому что они легче воспринимаются. Использование букв аналогично подходу, используемому для именования перечислимых типов в свойствах компонентов (см. главу 3). Выбор букв может быть совершенно произвольным или, например, основываться на названии фирмы. Например, в фирме под названием *Components for Builder* можно использовать обозначение `cfb`. В примере с компонентом `TComport` имя компонента может быть преобразовано в `TcfbComport`. Оно не очень привлекательно, но вполне допустимо, если пользователь не имеет доступа к исходному коду компонента. Вероятность создания другим разработчиком компонента с таким же именем очень мала. Если же пользователь имеет доступ к исходному коду компонента, его конкретный вид не имеет большого значения, потому что он может поменять его на более удачное имя.

Выбор имени, которое адекватно отражало бы назначение компонента, иногда требует тщательного обдумывания нескольких вариантов. При этом важно придерживаться традиционных имен. Например, следует избегать таких имен, как TCOM для компонента, который инкапсулирует последовательный порт, потому что пользователь может предположить, что компонент каким-то образом связан с COM-программированием. Сложность выбора подходящего имени часто свидетельствует о неудачном проектировании компонента. В таком случае следует подумать об изменении структуры такого компонента.

Распределение пакетов времени создания

До сих пор рассматривалось распространение компонентов в составе пакета времени создания и одного или нескольких пакетов времени выполнения. В этом разделе кратко рассматривается способ распределения компонентов с использованием модели пакета времени создания. В таком случае пользователь вынужден статически компоновать объектные файлы (obj) ваших компонентов непосредственно с приложением, в котором они используются. Это усложняет применение компонентов, но это не самое главное в данном случае. При создании в форме приложения небиблиотечного компонента IDE-среда выполняет две задачи: включает заголовочный файл модуля, который содержит определение компонента, а также добавляет директиву `#pragma link "имя_модуля"` в файл реализации формы (где `имя_модуля` — имя модуля, реализующего этот компонент). Это означает необходимость статической компоновки компонента с приложением. Для выполнения этой задачи пользователю не придется выполнять никакой дополнительный код. При условии, что компоновщик может найти все необходимые объектные файлы (.obj), эта схема будет работать вполне удовлетворительно. На рис. 11.7 показана структура этой модели распространения пакета времени создания.

Следует отметить, что вместо распространения самих объектных файлов (.obj), можно распространять файл статической библиотеки (.lib), содержащий объектные файлы компонентов. Этого можно добиться одним из следующих способов. Во-первых, можно создать пакет времени выполнения, содержащий только компоненты. После сборки lib-файл будет содержать все необходимые obj-файлы этих компонентов. Остальные файлы пакета — bpl-файл и bpi-файл — не потребуются. Файл библиотеки (.lib) можно будет распространять вместо отдельных объектных файлов (.obj). Еще одна возможность заключается в создании объектных файлов (.obj) непосредственно в пакете времени создания с помощью макроса `USEOBJ`. После этого можно распространять файл библиотеки (.lib) времени создания. В любом слу-

чае, если компоновщик не сможет найти необходимый файл библиотеки (.lib), придется отредактировать имя статической библиотеки в строке <SPARELIBS>, как показано ниже.

```
<SPARELIBS value="VCL50.lib MyStaticLibrary.lib"/>
```

Этот файл будет успешно скомпонован, если он находится в одном из каталогов с файлами библиотек, указанных в параметрах проекта.

Следует отметить, что при использовании подхода на основе пакета времени создания потребуется поддерживать только один пакет. При создании пакетов для разных версий C++Builder (см. следующий раздел) это может существенно упростить вам жизнь. Объектные файлы компонентов можно генерировать, компилируя компоненты для разных версий C++Builder. При этом предполагается, что код структурирован таким образом, чтобы он мог компилироваться для каждой версии C++Builder. Эту модель упаковки можно использовать в тех случаях, когда по какой-либо причине вам не хочется распространять библиотеку времени выполнения для динамической компоновки. Хотя в целом подход на основе пары пакетов для выполнения/создания обладает преимуществом, и рекомендуется использовать именно его.

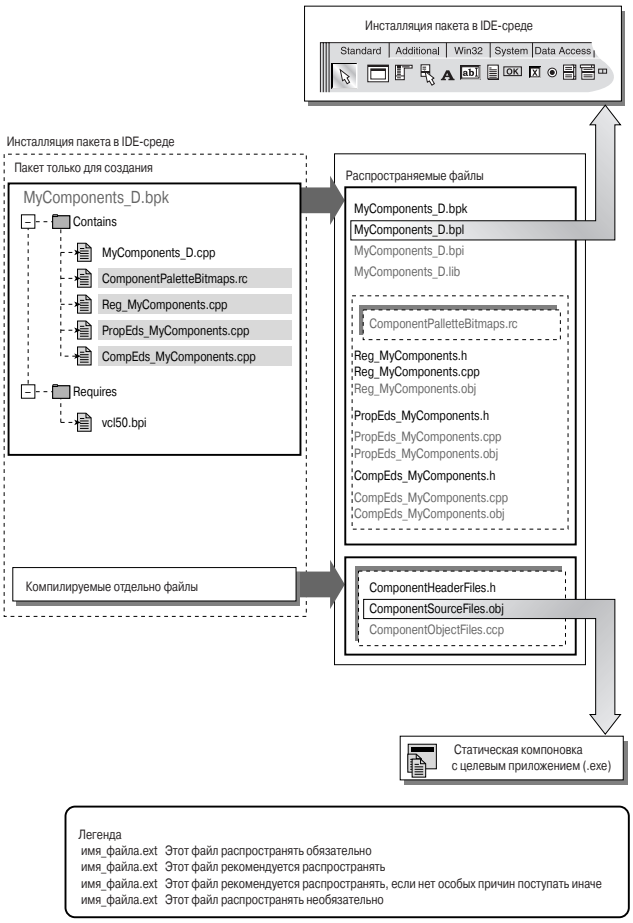


Рис. 11.7. Модель упаковки на основе пакета времени создания

Распространение компонентов для разных версий C++Builder

Если вы распространяете компоненты как поставщик, то должны предоставить версии компонентов для максимально возможного количества версий C++Builder (в настоящее время для четырех версий: 1, 3, 4 и 5). Конечно, в некоторых случаях это невозможно сделать, поскольку такие компоненты могут содержать функциональные возможности, имеющиеся только в определенных версиях компилятора. Однако, если предположить, что компоненты могут использоваться для нескольких версий C++Builder (в большинстве случаев это возможно), то придется создать несколько версий компонента. Для распространения компонентов для разных версий C++Builder требуется отдельно откомпилировать код для каждой версии. Следует также создать пакет для компонентов, инсталлируемых для версий 3, 4 и 5 (в первой версии C++Builder компоненты инсталлируются непосредственно в файле `CMPLIB32.CCL`; а пакеты для этого не используются). При этом следует учитывать все сказанное выше. Для этого используются соответствующие библиотеки импорта для каждой версии C++Builder и создается пакет. Все было бы прекрасно, если бы исходный код компонента компилировался для каждой версии компилятора. Это маловероятно, а потому именно эта часть работы вызывает наибольшие проблемы при распространении компонентов для разных версий. Вряд ли практично иметь отдельные листинги с кодом компонентов для каждой версии компилятора, поэтому обычно используется альтернативный способ. В нем для всех версий C++Builder используются одинаковые модули. Чтобы учесть различия между версиями компилятора, определяется версия компилятора и с помощью директив препроцессора выбирается код, предназначенный для каждой отдельной версии компилятора. Этот и другие вопросы, связанные с учетом версии компилятора, рассматриваются в следующих разделах.

Определение версии компилятора при компиляции

Каждая версия C++Builder обладает определенным номером версии компилятора. Проверяя его значение, можно выборочно компилировать разные фрагменты кода. В листинге 11.15 представлен метод использования директив `#defines` для работы с разными версиями C++Builder.

Листинг 11.15. Установка `#defines` для разных версий C++Builder

```
#ifndef VERSION_DEFINES
#define VERSION_DEFINES

   #if(__TURBOC__ == 0x550) // C++Builder 5
   #define CPPBUILDER_VERSION_5
   #endif

   #if(__TURBOC__ == 0x540) // C++Builder 4
   #define CPPBUILDER_VERSION_4
   #endif

   #if(__TURBOC__ == 0x530) // C++Builder 3
   #define CPPBUILDER_VERSION_3
   #endif
```

```
   #if(__TURBOC__ == 0x520) // C++Builder 1
   #define CPPBUILDER_VERSION_1
   #endif

#endif
```

Включая показанный в листинге 11.15 код в заголовочный файл модуля компонента или размещая его в другом заголовочном файле file, можно создать приведенный ниже код.

```
#ifdef CPPBUILDER_VERSION_5
// Регистрация фильтров свойств,
// которые поддерживаются только в версии 5
#endif
```

Используя директивы `#ifdef/endif` и `#ifndef/endif`, можно селективно удалять фрагменты кода для разных версий C++Builder. В главе 3 уже говорилось, что следует избегать использовать директивы препроцессора, за исключением крайних случаев. В данном примере мы имеем дело как раз с таким случаем, поэтому использование директив препроцессора может существенно упростить вам задачу.

Функция `ValidCtrCheck()`

Эта функция используется для определения наличия виртуальных функций в инсталлируемых компонентах во время компиляции. Компоненты с виртуальными функциями нельзя инсталлировать в IDE-среду. Используемая для этого функция в C++Builder версии 1 имеет другой вид.

```
static inline TComponentName *ValidCtrCheck()
{
   return new TComponentName(NULL);
}
```

В C++Builder версий 3, 4 и 5 функция `ValidCtrCheck()` выглядит иначе.

```
static inline void ValidCtrCheck(TComponentName *)
{
   new TComponentName(NULL);
}
```

Применение пакетов и C++Builder версии 1

В компиляторе версии 1 не предусмотрено использование пакетов, поэтому макрос `PACKAGE` не следует использовать после ключевого слова `class` в определении компонента. Не следует его размещать и перед `Register()` в функции регистрации. Кроме того, директива `#pragma package(smart_init)` не будет найдена компилятором в исходном коде компонента.

Так как пакеты не используются компилятором версии 1, то вам придется распространять только заголовочные и объектные файлы компонента вместе с отдельным модулем регистрации (по желанию).

Применение наборов `Set` в компонентах

Применение наборов `Set` в версиях 1 и 3 компилятора отличается от его применения в версиях 4 и 5 компилятора.

В пакете MJFSecurity Малкольма Смита (Malcolm Smith) C++Builder версии 3 (его можно найти по адресу http:\\www.mjfreelancing.com) содержится следующий код заголовочного файла.

```
#include <sysdefs.h>

enum TFailedShareRegKey {fsrNone, fsrInstalledDate,
                         fsrRegUser, fsrRegOrgn,
                         fsrRegCode, fsrRunCount,
                         fsrUserDefined};

typedef Set<TFailedShareRegKey, fsrNone,
            fsrUserDefined> TFailedShareRegKeys;

typedef void __fastcall (__closure *TLoadErrorEvent)
   (TObject *Sender, TFailedShareRegKeys FailedKeys,
    bool &Terminate);
```

В файле с исходным кодом используется код, аналогичный показанному в листинге 11.16.

Листинг 11.16. Исходный код, иллюстрирующий использование набора `Set`

```
TFailedShareRegKeys FailedKeys;
FailedKeys <<fsrNone;

// ... пример кода чтения параметров реестра:

if(MyReg->ValueExists(KeyNames->InstalledDate))
   FInstalledDate = MyReg->ReadDate(KeyNames->InstalledDate);
else
{
   FailedKeys >> fsrNone;
   FailedKeys << fsrInstalledDate;
}

if(MyReg->ValueExists(KeyNames->Username))
   FRegisteredName = MyReg->ReadString(KeyNames->Username);
else
{
   FailedKeys >> fsrNone;
   FailedKeys << fsrRegUser;
}

// ... и многое другое
// ... затем в коде вызова события:

if(FOnLoadError)
{
   bool Terminate = TerminateOnLoadError;
   FOnLoadError(this, FailedKeys, Terminate);
}
```

Код в листинге 11.16 позволяет создать набор `Set`, который определяет возможные причины неудачной загрузки. Набор `Set` передается в качестве параметра событию `OnLoadError`, с помощью которого пользователь может проверить эту информацию и действовать соответствующим образом.

Этот код будет успешно работать в версиях 4 и 5 C++Builder, но не в версиях 1 и 3. В версиях 1 и 3 требуется использовать следующее объявление.

```
template class TFailedShareRegKeys;
```

Это явное объявление принуждает компилятор компилировать все методы класса `Set`.

В C++Builder версий 4 и 5 уже содержится явное объявление класса `Set` в файле `$(BCB)\Include\Vcl\Sysmac.h`, включенное косвенно в строке `#include <system.hpp>`.

```
template<class T, unsigned char minEl, unsigned char maxEl>
class RTL_DELPHIRETURN Set;
```

Окружая директивами препроцессора строку `template class TFailedShareRegKeys;`, ее можно выборочно компилировать для версий 1 и 3 и игнорировать для версий 4 и 5.

Изображения в палитре компонентов Component Palette

При обсуждении пакетов изображения в палитре компонентов рассматривались как файлы ресурсов (`.rc` — resource script). Причина этого заключается в том, что использование файлов ресурсов (или, что более предпочтительно, одного файла ресурсов с описанием всех изображений) позволяет добиться большей гибкости при создании изображения для палитры компонентов. Механизм создания файлов ресурсов подробно рассматривается в главе 10 в разделе об использовании заданных изображений в пользовательских редакторах свойств и компонентов. Там также рассмотрены ограничения редактора изображений Image Editor в C++Builder. Конечно, создание изображений палитры компонентов с помощью более мощного графического инструмента и добавление их вручную с помощью файла ресурсов позволяет более эффективно использовать пользовательские палитры. Это позволяет улучшить внешний вид изображений компонентов в палитре компонентов. Более подробные сведения об этом можно найти в главе 10.

Рекомендации по проектированию распространяемых компонентов

Компоненты для собственного или внутреннего употребления и компоненты для пользователей создаются по-разному. Причина этого в том, что вы не представляете, как будут использовать ваш компонент пользователи. Поэтому вы должны проектировать компонент, имея в виду следующее.

- Не скрывать от пользователя функции компонента, которые, *возможно*, ему не потребуется. Если структура компонента не пострадает от наличия этой функции, ее следует сделать доступной. Всегда найдется тот, кому она пригодиться.
- При создании событий в компонентах следует убедиться в том, что события инициируются для всех возможных событий, которые пользователю нужно обрабатывать. Неспособность организовать отклик на некоторые события, потому что они не были учтены, может серьезно уменьшить полезность компонента.

- Не принуждайте пользователя полагаться на компоновку как на метод получения дополнительной функциональности за счет связывания нескольких компонентов. Например, при создании компонента с захватом входного потока от звуковой платы и компонента с отображением этих данных имеет смысл скомпоновать их вместе с процессом, управляемым этими компонентами. Однако если это единственный способ использования компонентов, то по сути они могут оказаться бесполезными. Дело в том, что захваченные звуковой платой данные следует всегда делать доступными для отображения или обработки любыми другими средствами, которыми располагает пользователь.
- Постарайтесь создать интуитивно понятный интерфейс, особенно для времени создания. У большинства пользователей это сформирует первое представление о вашем компоненте. Тщательно продумайте использование редакторов свойств и компонентов. Грамотное применение фильтров свойств (категорий свойств) также может упростить работу с компонентом, обладающим множеством свойств.
- Наконец, можно создать абстрактный базовый класс вашего компонента, как, например, компоненты типа `TCustom` в C++Builder. Это не всегда может пригодиться, но окажет существенную пользу многим пользователям вашего компонента.

Другие вопросы распространения компонентов

Следует также сказать о настройке редакторов компонента для управления гиперссылкой на ваш Web-узел, отображение логотипа вашей фирмы или любой другой информации (например, версии компонента). Информацию по этому поводу можно найти в главе 10.

Далее нужно решить вопрос о том, будет ли ваш компонент бесплатным или условно бесплатным. Хотите ли вы распространять вместе с ним его исходный код или нет? Следует ли запаковать компоненты таким образом, чтобы они инсталлировались с предложением лицензионного соглашения? И так далее. Эти вопросы выходят за рамки этого раздела. Однако их следует учитывать при распространении компонентов. Более подробная информация по этому поводу приводится в главе 28 о распространении программного обеспечения и в главе 29 об инсталляции и обновлениях программного обеспечения.

Заключительное замечание касается распространения сопроводительной документации для всех компонентов, кроме наиболее простых. Она может поставляться в форме справочных файлов, электронных документов или даже расширенных инструкций и комментариев в заголовочных файлах самих компонентов. Предпочтительнее справочные файлы и печатная версия электронной документации. Не имеет никакого значения, насколько удачным получился ваш компонент, потому что без соответствующей документации он будет практически бесполезным. Следует создать и распространить пример кода и приложения, демонстрирующего способ использования этого компонента. Кроме того, очень полезным было бы создание файла с ответами на часто возникающие вопросы. Более подробно процесс создания справочных файлов и документации описывается в главе 27.

Резюме

Эта глава посвящена созданию пользовательских компонентов. Здесь кратко рассмотрен процесс создания интерфейса компонента, хотя на несколько другом уровне, чем в предыдущей главе. Основное внимание уделено таким специальным вопросам, как использование функций обратного вызова и обработке сообщений в компонентах. Здесь рассмотрены фрэймы и показано, как они упрощают процесс визуальной разработки компонентов. Обсуждено распространение компонентов, хотя в этой части главы нам удалось рассмотреть только не-

большую часть этой обширной темы. На самом деле полному описанию этого предмета требуется посвятить несколько глав книги.

Область создания компонентов охватывает много дисциплин. Как всегда, опыт, тщательное внимание к подробностям и знания должны помочь вам в создании качественных и надежных компонентов. Создание компонентов представляет собой интересную и благодарную задачу. При этом нельзя недооценивать смысл создания чего-то осязаемого и полезного. Успех создания качественного компонента во многом зависит от затраченного времени и усилий.

ЧАСТЬ

II

Обмен информацией, базы данных и программирование Web

ПРОГРАММИРОВАНИЕ ОБМЕНА ИНФОРМАЦИЕЙ

ПРОГРАММИРОВАНИЕ WEB-СЕРВЕРА

ПРОГРАММИРОВАНИЕ БД-ПРИЛОЖЕНИЙ

Глава

12

Программирование обмена информацией

Кит Тернбулл II
Марк Дэйви

Немногие из людей склонны жить на необитаемом острове, точно так же редко какая программа обходится без общения с другими. Сеть коммуникаций — это кровеносная система человеческой цивилизации, а средства обмена информацией — неотъемлемый компонент подавляющего большинства компьютерных программ. Диапазон этих средств огромен — от передачи информации на периферийные устройства до подключения к другим программам через сеть Internet.

В этой главе будут освещены темы, касающиеся программирования обмена информацией через последовательные порты. Вы познакомитесь с множеством сетевых протоколов, доступных программисту через компоненты FastNet, которые входят в состав C++Builder. Внимательно изучив материал этой главы, вы сможете в дальнейшем пользоваться всеми инструментами, необходимыми для включения в программу эффективных средств обмена информацией с любой существующей программой.

Обмен информацией по последовательным каналам

Для разработки программ, обеспечивающих интерфейс с внешними устройствами, важнейшее значение имеет понимание принципов передачи информации по последовательным каналам. В мире программных систем отдельный компьютер — это только маленький компонент большой системы, в которой интеграция компонентов имеет первостепенное значение. Для подавляющего большинства приложений вполне достаточно тех возможностей, которыми обладают последовательные порты компьютера. В этом разделе мы только вкратце остановимся на основных вопросах программирования передачи данных через последовательный порт.

На заметку

Фрагменты программного кода, которые будут приведены в этом разделе, носят учебный характер и не предназначены для прямого копирования в разрабатываемые программы. Они заимствованы из больших и довольно сложных программ, работающих напрямую с определенными устройствами. Поэтому мы снабдили эти фрагменты пространными комментариями.

В состав пакета Win32 API входит множество функций поддержки работы с последовательными каналами передачи данных. Но вряд ли имеет смысл включать в программу функции низкого уровня управления аппаратурой, если вы собираетесь создавать программные продукты, работающие в различных модификациях операционной системы Windows. Все функции такого рода из состава Win32 API перечислены в системе оперативной справки к этому пакету в разделе “About Communications”.

Вряд ли нужно кого-либо убеждать в том, какую важную роль играет в наше время Internet. Широкое распространение этой сети не в последнюю очередь объясняется тем, что она “покоится” на жестком, надежном и хорошо документированном фундаменте. Это обстоятельство часто упускают из виду. Internet как система базируется на множестве протоколов. Описывая методику разработки программ обмена информацией мы рассмотрим не только различные виды протоколов, но и методы управления состояниями, архитектуру систем коммуникаций, методы синхронизации потоков и методы буферизации.

Протокол обмена информацией

Протокол — это соглашение, которое закрепляет определенный порядок выполнения операций в ходе обмена информацией. Фактически для любого мало-мальски серьезного приложения вам потребуется не просто отдельный протокол, а *стек протоколов*.

Стек протоколов обмена информацией представляет собой набор взаимосвязанных протоколов, функционирующих на разных *уровнях*. В своих разработках вам чаще всего придется иметь дело с тремя уровнями: *физическим (Physical)*, *транспортным (Transport)* и *прикладным (Application)*.

Стек протоколов можно рассматривать как своего рода иерархию классов. Роль базового класса играет протокол нижнего уровня, поддерживающий большинство базовых функций (в нашем случае — работу с физическими устройствами на уровне передачи отдельных битов). Чем более специализированным является класс-наследник, тем к более верхнему уровню относится соответствующий этому классу протокол. В конце этой цепочки стоит интерфейс приложений — ваш "конкретный" класс.

На самом нижнем уровне — физическом — можно использовать широко известный протокол RS-232, который регламентирует передачу данных порциями по 8 бит с одним стоповым без контроля по паритету — просто и без особых претензий. Этот протокол регламентирован международным стандартом и единственная проблема для конструктора — обеспечить правильное подключение проводов в кабеле.

Далее следует транспортный уровень. Протокол этого уровня регламентирует последовательность обмена сообщениями в процессе передачи данных между приложениями — формат данных, определяющий, где начинается передаваемое сообщение и где оно заканчивается, как отличить правильно переданные данные от искаженных и что нужно делать, если процесс идет не так, как следует.

Протокол прикладного уровня определяет смысл передаваемых данных: является ли сообщение командой, данными, которые нуждаются в дальнейшей обработке, либо сообщением об обнаруженной ошибке. Зная смысл поступившего сообщения, можно на него адекватно отреагировать, а это уже забота приложения.

Процесс разработки протокола можно разделить на 10 этапов.

1. Выработка требований к протоколу.
2. Анализ потенциальных причин появления ошибок.
3. Определение требований к производительности процесса передачи информации.
4. Оценка параметров информационных потоков в обоих направлениях.
5. Выявление общих характеристик сообщений.
6. Выбор подходящей архитектуры.
7. Разработка метода тестирования.
8. Реализация.
9. Тестирование.
10. Повторение этапов 8 и 9 до тех пор, пока программа не будет работать в соответствии с протоколом.

Выбор протокола

Для выбора подходящего протокола нужно в первую очередь иметь четкое представление о том, какого вида информация будет передаваться и какие требования при этом должны быть соблюдены. Эти требования определяют выбор того или иного варианта протоколов нижних уровней — транспортного и физического.

Протокол может быть синхронным или асинхронным, символьным (в кодах ASCII) или двоичным, простым или сложным. Какой бы вариант вы не выбрали, нужно обязательно продумать методику его тестирования. Тщательное тестирование должно подтвердить работоспособность выбранного протокола, причем желательно смоделировать как можно больше вариантов внешних условий, влияющих на процесс обмена. В первую очередь подумайте над возможностью

применения символьных протоколов, использующих кодировку ASCII, поскольку их можно тестировать с помощью стандартных инструментальных средств, например ***HyperTerminal***.

Но в тех приложениях, которые требуют интенсивного обмена информацией или специальных мер защиты, протоколы, базирующиеся на кодировке ASCII, не подходят. В такой ситуации следует ориентироваться на двоичные протоколы, которые предоставляют гораздо более широкие возможности управления процессом обмена и в определенной степени даже упрощают требования к обработке информации. Но отлаживать программы, реализующие протоколы этого класса, значительно сложнее.

Синхронизация процесса обмена на уровне протокола (т.е. использование синхронных протоколов) имеет важное значение для тех приложений, в которых нужно непрерывно следить за достоверностью передаваемой информации. Синхронные протоколы проще в реализации, поскольку отправитель не может отослать очередную порцию до тех пор, пока не получит подтверждение о приеме и отсутствии искажений в ранее переданной порции данных. Но асинхронные протоколы обеспечивают более высокую скорость передачи данных при тех же физических характеристиках канала. Протоколы этого класса имеет смысл использовать в тех приложениях, в которых передаваемые данные не зависят от ответа принимающей стороны (например, передача определенной команды не влияет на то, какая следующая команда будет отослана).

Прикладной протокол

Основное назначение прикладного протокола — обеспечить поддержку функциональных возможностей приложения, т.е. передачу определенных команд устройствам или запросов серверу. Эти запросы имеют смысл в контексте определенного приложения, которое "знает", как реагировать на сообщение, поступившее в ответ.

Прикладной протокол "разбирается" в содержимом пакета сообщений, и в этом протоколе должно быть определено, как приложение будет на него реагировать. В таком протоколе можно организовать опрос состояния устройства, чтобы выяснить, содержится ли достоверная информация в его памяти или установить параметры внутреннего генератора синхронизации устройства. Именно приложение "знает", какие команды воспринимает устройство и в каком формате эти команды нужно передавать.

Нужно составить список всех команд и данных, которые будут передаваться и приниматься, и добавить к нему всю имеющуюся информацию относительно этих команд и данных, включая форматы слов и байт. Тем самым будет создан базис для формирования стека протоколов. Более нижние уровни в нем обеспечивают пересылку этой информации от отправителя к получателю.

В прикладном протоколе должно быть предусмотрено, как будет реагировать система на возможные сбои в процессе обмена — искажение передаваемых данных, неверный ответ на переданную команду или сбои в синхронизации процесса. В принципе, средства обработки таких ошибочных ситуаций могут быть предусмотрены в протоколах разных уровней, а не только прикладного. Желательно предусмотреть определенные меры на нижних уровнях для снижения вероятности сбоев на прикладном уровне.

В процессе выработки требований к прикладному протоколу уточняется объем передаваемых данных и смысл сообщений, которыми обмениваются взаимодействующие программы, а это в дальнейшем значительно упростит работу по реализации программ. Выделение похожих сообщений и подобных методов обработки позволяет более системно организовать наследование в иерархии классов, а это, в свою очередь, будет способствовать созданию более надежного и удобного в сопровождении программного кода.

Транспортный протокол

Сформулировав требования к прикладному протоколу, можно переходить к более нижним уровням. Транспортный протокол отвечает за "упаковку" сообщений и передачу их по назна-

чению. Можно считать, что приложение снабжает средства реализации транспортного протокола блоками сообщений или данных, а их забота — отправить эти посылки и доставить их адресату в целости и сохранности.

Как правило, сообщение можно разбить на секции — заголовок, блок данных, заключительная секция — "хвост". В заголовке содержится информация о параметрах сообщения — его длина и идентификатор. При передаче по последовательному каналу неплохо включить в заголовок и маркер, который даст знать принимающей стороне, что далее последует блок данных.

Обсуждая прикладной протокол, мы предлагали составить список команд и всех связанных с ними полей данных. Теперь мы расширим эту рекомендацию и на транспортный протокол, который выступает в роли почтальона по отношению к данным, подготовленным для передачи.

Сообщение может иметь, например, такой формат:

```
<msgStartCharacter>;<msgLength>;<msgID>;<msgDataBlock>;
↳<msgCheckSum>;<msgEndCharacter>
```

Здесь каждый элемент в угловых скобках, например `<msgDataBlock>`, обозначает компонент сообщения.

Выбор определенного формата позволяет приступить к разработке алгоритма реализации интерфейса. Получатель сообщения должен следить за появлением маркера начала сообщения — компонента `msgStartCharacter`. Получив его, можно с уверенностью сказать, что следующий компонент `msgLength` содержит информацию о длине блока — количество байт. Если в процессе приема порции данных длиной в `msgLength` байт встретится код `msgStartCharacter`, принимающая сторона будет рассматривать его не как маркер начала следующего сообщения, а как очередной элемент данных. После того как блок данных заданной длины принят, его компоненты `<msgID>;<msgDataBlock>` можно передать более верхнему уровню приложения. Программа реализации транспортного протокола также должна взять на себя определенные функции обнаружения сбоев и реакции на них в процессе передачи. В частности, если в течение заданного отрезка времени (тайм-аута) пришло менее `msgLength` символов данных, эта программа должна игнорировать все дальнейшие символы до тех пор, пока не обнаружит следующий маркер начала сообщения `msgStartCharacter`.

На уровне транспортного протокола также можно определять корректность принятых данных, т.е. отсутствие в них искажений. Один из наиболее распространенных способов — анализ контрольной суммы `<msgCheckSum>`, которая включается в формат сообщения. Контрольная сумма формируется передающей стороной, причем для этого могут использоваться разные способы. Самый простой — суммировать байты сообщения с помощью операции XOR (исключительное ИЛИ), но иногда применяются и более сложные алгоритмы. Выбор алгоритма — прерогатива разработчика. Чем более сложный алгоритм формирования контрольной суммы применяется, тем больше времени уходит на оценку достоверности принятого сообщения. Вероятность появления искажений в процессе передачи определяется характеристиками приложения. Если приложения работают в "тепличных" условиях лаборатории или офиса, длина кабелей невелика и используется физический протокол RS232, то вполне достаточно суммирования операцией XOR. Если же данные передаются по длинным каналам связи (например, через Internet), следует подумать о более сложных алгоритмах.

Если будет обнаружен какой-либо сбой в процессе передачи (например, ошибка тайм-аута), анализ контрольной суммы может стать дополнительным аргументом в пользу игнорирования полученного сообщения. Один из возможных вариантов дальнейших действий в этом случае — использование механизма повторной передачи сообщения.

В простейшем варианте контрольная сумма формируется суммированием с помощью операции XOR всех байт данных (например тех, которые входят в состав компонента `msgDataBlock`). При суммировании можно использовать 8-, 16- или 32-битовый формат данных. Общепринято не считать старший бит знаковым, т.е. рассматривать все данные в формате `unsigned`.

Процессор обработки состояний

При реализации протокола в виде программы, последняя должна следить за отправкой сообщений и ожидать соответствующей реакции — ответа — от другой стороны. Это необходимо для того, чтобы знать, принято ли переданное сообщение, а от ответа, в свою очередь, зависит, что будет передаваться дальше. Кроме того, в процессе передачи могут возникнуть "нештатные" ситуации, на которые нужно адекватно реагировать. Все эти функции возлагаются на компонент программы, который мы назвали "процессором обработки состояний".

Многим программистам приходилось разрабатывать подобные процессоры, хотя они об этом и не подозревают. Думаю, вы не раз включали в свои программы кнопки, которые блокировались или разблокировались в зависимости от состоянию других компонентов экранной формы (очень часто это приходится делать с кнопками OK и Cancel). Программа, которая это делает и есть простейший процессор обработки состояний. Она значительно усложняется, когда приходится управлять процессом передачи данных, но смысл работы программы тот же.

Производительность против надежности

В зависимости от назначения конкретного приложения, разработчик должен выбрать, что представляет для него наибольший интерес — производительность, надежность или точность, и чем можно пожертвовать (конечно, только в определенных пределах). В идеале хотелось бы получить и то, и другое, но в жизни чем-то всегда приходится жертвовать. Как достичь желаемого, и нет ли тут золотой середины?

Когда речь идет о передаче информации, чем быстрее канал, тем лучше, но при этом одновременно и повышается вероятность появления ошибок. Если в ПК используется стандартный COM-порт RS-232, то проблемы, связанные с длиной кабеля, начинают проявляться при передаче данных на расстояние свыше 50 м. Причина здесь кроется в том, что порт RS-232 не предназначен для работы на более длинный канал связи, в отличие от порта, в котором используются стандарты RS422 или R485. Если вам нужно передавать данные в пределах 4 км, то воспользуйтесь стандартом RS422, и канал будет работать прекрасно.

Лично я предпочитаю иметь дело с надежными данными, а уж исходя из этого выбирать наивысшую скорость передачи, но такой подход требует использования достаточно сложных алгоритмов и самого совершенного оборудования. Если требуется передавать данные через локальный модем, то основные заботы вам принесет качество телефонной линии.

Очень важно учитывать при этом характер информационных потоков. Если в приложении нужно получать большие массивы данных в ответ на сравнительно короткие сообщения, то основное внимание придется уделить принимающей стороне. Если же приложение само должно отсылать большие объемы данных и получать в ответ короткие подтверждения, на передний план выходит передающая сторона. В любом случае можно оптимизировать систему таким образом, чтобы она наилучшим образом соответствовала конкретным условиям работы (т.е. подобрать размеры буферов, уровни приоритета программных потоков и т.д.).

Сбои и ошибки

Как бы вы ни старались, избежать сбоев в работе системы передачи информации не удастся. Причины возникающих сбоев могут также поразить любого (лично я уже перестал им удивляться).

Ошибки могут возникнуть в данных, в выполнении алгоритма, предусмотренного протоколом, или в интерпретации получаемых сообщений, причем даже там, где, казалось бы, все абсолютно ясно. Лучше заранее смириться с этим, чем пытаться задним числом искать виновного после того, как программа будет написана.

Нужно заранее подумать о возможности появления ошибок подключения и синхронизации, искажения данных при передаче, ошибочной интерпретации сообщений и продумать

адекватные меры реагирования на них. С каждым видом ошибок связаны свои проблемы, и для их разрешения используются специальные методы. В этой книге мы рассмотрим основные виды ошибок — ошибки синхронизации (тайм-аута), искажение данных, а также и проблему интерпретации сообщений.

Архитектура

Теперь можно задуматься и над архитектурой разрабатываемого последовательного интерфейса. Во-первых, вам обязательно придется выделить для интерфейса отдельный поток. Этого можно избежать только в том случае, если создается очень простая терминальная система, но лично я и в этом случае выделил бы в ней отдельный поток для интерфейса.

Без разделения потоков вам не удастся воспользоваться операционной средой Windows. Таким образом, вопрос состоит не в том, разделять ли потоки, а в том, до каких пределов целесообразно их разделять и как после этого управлять процессом в целом.

Как по мне, выбор архитектуры — самая интересная стадия разработки системы. Многопотоковая организация позволяет получить прекрасные результаты, если с умом ее использовать. Вы сможете создавать высокоэффективные приложения, если досконально разберетесь в сути этой концепции.

Если предполагается использовать в приложении асинхронный протокол, то можно отсылать и получать совершенно независимые сообщения, для чего нужно организовать независимые потоки для их передачи и приема. Если же требуется синхронизация на каком-то уровне (например, посылается сообщение и ожидается ответ на него), то можно выбрать другой вариант — использовать единый поток, в рамках которого выполнять и прием, и передачу.

Обычно я организую два потока — “read” для приема и “write” для передачи. Такая организация позволяет быстро реагировать на поступающие данные и распознавать ситуацию, когда в ответ на отосланное сообщение не поступил ответ. Передача больших массивов данных в обе стороны при этом никак не сказывается на производительности приложения в целом, поскольку в это время приложение может решать другие задачи.

Методы синхронизации потоков

Если вы остановились на многопотоковой организации вычислительного процесса в приложении, то необходимо озаботиться проблемой взаимной увязки потоков. Для того чтобы дать знать потоку, что пора взяться за дело, я использую механизм обработки событий. Это просто и эффективно. Поток ожидает сообщений об определенных событиях. Например, поток `write` имеет в своем составе обработчик события “сообщение подготовлено; пожалуйста, отошли его”. Данные, которые подлежат передаче, передаются в поток `write` через общедоступный протокол сообщений (вызов функции). В результате данные через критическую секцию попадают в закрытое (private) пространство данных потока, откуда они выбираются для передачи. Одновременно поток `write` сигнализирует потоку `read`, что данные отосланы, а последний запускает механизм отслеживания тайм-аута.

Поток `read` постоянно ожидает от операционной системы сигнала о том, что можно прочесть поступившее сообщение. Получив этот сигнал, поток `read` анализирует длину сообщения, считывает его во внутренний буфер и сигнализирует приложению о том, что данные получены, вызовом `PostMessage()`. Это, в свою очередь, запускает последовательность операций, предусмотренных протоколом.

Для того чтобы поток `read` мог сигнализировать приложению о том, что получено сообщение или данные, я пользуюсь механизмом передачи сообщений между окнами. Такое решение вполне резонно, поскольку, если в потоке `write` произойдет что-либо непредвиденное,

то возникнет ошибка тайм-аута потока `read`, которая возбудит сообщение о сбое процесса обмена, и приложение узнает об этом от процесса `read`.

Неплохо также "вложить" эти потоки в стек протокола (или класс) или по крайней мере скрыть от них приложение, не допустив непосредственного вмешательства. Это означает, что внутренний механизм управления состояниями протокола будет защищен от вмешательства извне.

В этом классе полученные данные копируются во внутренний буфер, что позволяет работать с сообщениями переменной длины. Если длина сообщений фиксирована, то можно модифицировать поток `read` таким образом, чтобы он считывал блоки фиксированной длины.

Именно класс протокола заботиться об отсылке сообщения. Он "знает", чего следует ожидать и сколько должно быть данных. Однако такое возможно не всегда. Мне приходилось иметь дело с приложениями, в которых протоколом предусматривалась передача в обе стороны сообщений переменной длины. Класс протокола также обрабатывает поступившие данные и после завершения обработки посылает приложению сообщение "получено сообщение, к котором имеются данные". Если внутри этого класса были созданы какие-либо структуры или объекты, не забудьте удалить их, после того как завершите работу с ними.

Буферизация

Буферизация, как правило, редко упоминается при анализе средств обмена данными, но в процессе разработки системы этот аспект не стоит выпускать из поля зрения. Для отсылаемых данных буферизацию лучше не использовать, если только не приходится отсылать большое количество данных. Если программа предназначена для устройства управления, которое часто отсылает сообщения небольшой длины подчиненным устройствам, то лучше всего обходиться вообще без буфера. Я часто устанавливаю при настройке Windows значение 0 для параметра Device Driver TX Buffer. Я выделяю маленький приемный буфер для того, чтобы входной поток не слишком часто "дергал" программу, хотя для приложений, которые имеют дело со скоростями передачи менее 56 Кбод в секунду в этом, вероятно, и нет необходимости, даже в сравнительно медленных переносных компьютерах (лептопах).

Что действительно необходимо, так это сохранять принятые данные в FIFO буфере. Это можно без труда организовать с помощью класса `AnsiString` (я обращаюсь с ним, как с классом `String`), добавляя новые данные в конец и удаляя обработанные данные из начала буфера. Этот метод достаточно эффективен и работает очень быстро. Можно использовать и класс `String` при работе с 8-битовыми двоичными данными, но здесь придется поработать с потоками Blob-объектов, если вы захотите использовать базы данных для хранения информации.

Немало хороших средств для работы с очередями содержится и в библиотеке STL, и об их использовании тоже имеет смыл подумать. Но при этом может пострадать производительность программы. Учтите также, что при работе со сложными протоколами обмена очень важно сохранять контроль над процессами обработки данных в ходе отладки и тестирования программы.

Некоторые соображения, касающиеся последовательного обмена

Главное в проектировании системы обмена информацией — спецификация протоколов. Имея четко сформулированные протоколы всех уровней, можно составить достаточно ясное

представление о структуре потоков в программе. А уже после этого нужно определиться со взаимной увязкой потоков и организацией буферов.

Структурная организация иерархии протоколов в значительной степени зависит от специфики разрабатываемого приложения, выбранных способов обработки нештатных ситуаций (ошибок в ходе обмена информацией), требований к надежности, точности, производительности и обслуживанию. Многие программисты подчас забывают, что разработанный ими программный код будет работать достаточно долго после завершения проекта, а следовательно, кто-то должен будет его сопровождать. Это будет возможно только в том случае, если имеется четкое представление о том, чего вы старались достичь при разработке программы и какие способы достижения цели выбрали.

Постарайтесь досконально изучить технологию организации потоков в программе. Вы увидите, что большинство операционных систем поддерживает большой набор средств синхронизации потоков, таких как события, критические секции и передача сообщений.

На заметку

В каталоге `SerialIO` компакт-диска, прилагаемого к этой книге, вы найдете проект *cd5Book.bpr*, в котором используется последовательный обмен информацией. Программы в этом проекте демонстрируют, как можно структурировать систему обмена информацией. Вы также сможете на примере познакомиться с тем, как открывать, закрывать, считывать и записывать данные через последовательный порт, как организовать большое количество потоков, взаимодействие между потоками и буферизацию данных.

Протоколы Internet — SMTP, FTP, HTTP, POP3

- В последние годы на рынке программных продуктов ведущие позиции заняли приложения класса клиент/сервер, работающие в сети Internet. Создание таких приложений стало возможным только благодаря использованию общедоступных протоколов. Канули в Лету времена, когда считалось высшим шиком использовать в приложении протокол собственной "конструкции". В этом разделе вы сможете познакомиться с компонентами C++Builder, позволяющими разработчику быстро создать приложение, в котором используется какой-либо из существующих стандартных протоколов. Сначала мы познакомим вас с компонентами, которые поддерживают работу с сетями, а потом рассмотрим приложение, в котором используются протоколы HTTP, POP, SMTP и TCP/IP.

Сетевые компоненты в составе C++Builder

В C++Builder версий 3 и 4 все программные компоненты, предназначенные для работы с сетью Internet, были сосредоточены на одной вкладке Internet. В пятой версии C++Builder разработчик компонентов — фирма Inprise — разделила такого рода компоненты на две группы. Первая группа представлена на вкладке Internet. В нее входят компоненты `TClientSocket` и `TServerSocket`, а также разнообразные компоненты вида `PageProducer`, которые используются для создания приложений Web-серверов. Полный перечень компонентов на вкладке Internet представлен в табл. 12.1. Вторая группа компонентов выведена на вкладку FastNet. В нее включены ActiveX-компоненты FastNet, которые можно использовать при разработке самых разнообразных сетевых приложений. Полный список компонентов этой группы вы найдете в табл. 12.2.

Таблица 12.1. Компоненты на вкладке Internet

Компонент	Назначение
TClientSocket	Инкапсулирует Winsock для использования в клиентском приложении
TServerSocket	Инкапсулирует Winsock для использования в приложении-сервере
TWebDispatcher	Позволяет использовать модуль данных в качестве расширения Web-сервера
TPageProducer	Используется для преобразования HTML-шаблона в HTML-документ
TQueryTableProducer	Используется для преобразования объекта TQuery в HTML-таблицу
TDataSetTableProducer	Используется для преобразования объекта TDataSet в HTML-таблицу
TDataSetPageProducer	Используется для включения в HTML-документ результатов, полученных объектом TDataSet
TCppWebBrowser	Инкапсулирует Web-броузер Internet Explorer для использования его в клиентских приложениях

Таблица 12.2. Компоненты на вкладке FastNet

Компонент	Назначение
TNMDayTime	Используется для получения даты и времени от Internet-сервера службы времени
TNMMsg	Используется для передачи простого текстового сообщения в кодировке ASCII по сети Internet или Intranet с помощью протокола TCP/IP
TNMMsgServ	Используется для создания сервера обработки сообщений, переданных объектом класса TNMMsg
TNMEcho	Используется работы с эхо-сервером Internet в процессе отладки приложений
TNMFTP	Используется для создания клиентского приложения FTP
TNMHTTP	Устанавливает соединение с сервером HTTP в сети World Wide Web
TNMNNTP	Используется для работы с серверами новостей Internet и Intranet
TNMStrm	Отсылает потоки на сервер потоков через Internet или Intranet
TNMStrmServ	Создает сервер потока, который может получать потоки по сети Internet или Intranet
TNMPOP3	Используется при создании клиентского приложения для чтения сообщений, переданных по электронной почте с использованием протокола POP3
TNMSMTP	Используется при создании клиентского приложения для передачи сообщений по электронной почте с использованием протокола SMTP
TNMTime	Получает время от сервера времени Internet
TNUDP	Используется при реализации пользовательского протокола передачи датаграмм (User Datagram Protocol — UDP) для пересылке датаграмм по сети Internet или Intranet
TNMURL	Выполняет преобразование URL в строку и строки в URL

Окончание табл. 12.3

Компонент	Назначение
TNMUUProcessor	Выполняет кодирование/декодирование MIME-документов или документов в нейтральном формате (uuencoded)
Tpowersock	Базовый класс для многих сокет-компонентов на вкладке FastNet
TNMGeneralServer	Базовый класс для разработки многопотоковых Internet-серверов
TNMFinger	Получает информацию о пользователе из Internet с помощью протокола Finger

Приложение ChatServer

Первой в качестве примера мы рассмотрим программу поддержки "болтовни" через сеть Internet. Эта программа позволяет пользователю подключиться к центральному серверу и передавать сообщения другим пользователям, принимающим участие в обсуждении. Все программные компоненты этого проекта вы найдете в папке Chat\Server на компакт-диске, который прилагается к этой книге. Имя файла проекта — ChatServer.bpr.

Создание приложения

Создайте новый проект.

1. Выберите в меню File⇨New. При этом будет создана пустая экранная форма Form1 и файл программного кода Unit1.cpp.
2. Добавьте в экранную форму Form1 компонент TMemo и присвойте его свойству name значение **LogMemo**.
3. Добавьте в экранную форму компонент TPanel и присвойте значение **alTop** его свойству align. Свойству Caption этого компонента присвойте значение **blank**
4. Свойству align элемента LogMemo присвойте значение **alClient**.
5. Установите две кнопки (компоненты TButton) на панель Panel1; Одной присвойте наименование **StartButton**, а второй — **StopButton**.
6. Добавьте в форму компонент TLabel и установите в нем текст надписи **Port**; добавьте также компонент TEdit, которому присвойте наименование **PortEdit**.
7. Добавьте в форму компонент TServerSocket (его вы найдете на вкладке Internet) и присвойте ему наименование **MyServer**. Компоновка элементов управления на поле экранной формы должна выглядеть примерно так, как показано на рис. 12.1.
8. Выберите в меню команду File⇨Save All. Когда на экране откроется диалоговое окно Save, укажите в нем каталог, в котором будут размещаться файлы проекта. Я рекомендую назвать этот каталог Server, а проекту дать имя ChatServer.

Рис. 12.1. Компоновка элементов управления в экранной форме приложения-сервера

Заполнение списка пользователей

Теперь сформируем список — объект класса TStringList, в котором будут перечислены пользователи, подключенные к серверу в текущий момент. Чтобы сделать это, добавьте в файл заголовка Unit1.h после ключевого слова private: строку

```
TStringList *ConnectedList;
```

После этого откройте в окне редактора файл Unit1.cpp и добавьте в него программный код метода OnCreate() класса экранной формы:

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    ConnectedList = new TStringList();
}
```

Запуск и останов работы сервера

Следующий шаг — добавить программный код в объекты кнопок Start и Stop. Эти кнопки должны запускать и останавливать работу сервера. В обработчик события OnClick() объекта StartButton добавьте следующий код:

```
void __fastcall TForm1::StartButtonClick(TObject *Sender)
{
    MyServer->Port = PortEdit->Text.ToIntDef(1971);
    Caption = MyServer->Port;
    MyServer->Active = true;
    StopButton->Enabled = true;
    StartButton->Enabled = false;
    PortEdit->Enabled = false;
}
```

Первый оператор устанавливает порт для сервера. Поскольку PortEdit->Text является объектом класса AnsiString, воспользуйтесь методом ToIntDef() этого класса. Это позволит установить значение по умолчанию для порта в том случае, если в строке PortEdit->Text окажется нечто, что нельзя преобразовать в целое число.

Далее добавьте программный код в обработчик события OnClick() кнопки StopButton:

```
void __fastcall TForm1::StopButtonClick(TObject *Sender)
{
        MyServer->Active = false;
        StartButton->Enabled = true;
        StopButton->Enabled = false;
        PortEdit->Enabled = true;
}
```

Теперь программа способна установить номер порта, который будет использовать сервер, а также запустить сервер в работу и остановить его. Номера портов позволяют единственной системе, идентифицированной по значениям свойств Host или Address, работать с множеством подключений одновременно. Примерами номеров портов по умолчанию могут быть 80 — для Web-серверов и 23 — для серверов Telnet. Для нашего сервера ChatServer значением номера порта по умолчанию будет 1971. Номера портов могут иметь значения в диапазоне от 1 до 9999.

Управление подключениями

Теперь займемся организацией подключения пользователей к нашему серверу. Программа должна соответствующим образом реагировать на регистрацию пользователей на сервере и их отключение от сервера. Добавьте следующий код обработки события OnClientConnect:

```
void __fastcall TForm1::MyServerClientConnect(TObject *Sender,
     TCustomWinSocket *Socket)
{
    LogMemo->Lines->Add(Socket->RemoteAddress +
          " has connected.");
    ConnectedList->AddObject(Socket->RemoteAddress, Socket);
}
```

Тем самым в журнал LogMemo добавляется строка с адресом клиента, который собирается подключиться к серверу. Затем в список ConnectedList добавляется сокет, через который подключается клиент, причем в качестве идентификатора используется адрес клиента.

В программу обработки события OnClientDisconnect добавьте следующий программный код:

```
void __fastcall TForm1::MyServerClientDisconnect(TObject *Sender,
     TCustomWinSocket *Socket)
{
    LogMemo->Lines->Add(Socket->RemoteAddress +
        " has disconnected.");
    SendMessage(Format("%s has disconnected.",
      OPENARRAY(TVarRec, (ConnectedList->Strings[
          ConnectedList->IndexOfObject(Socket)]))),
         "Server");
    ConnectedList->Delete(ConnectedList->IndexOfObject(Socket));
}
```

Этот код добавляет строку в журнал LogMemo, которая регистрирует отключение клиента от сервера. Затем вызывается метод SendMessage(), который мы рассмотрим чуть позже. Эта функция информирует всех подключенных в текущий момент клиентов о том, что один из "собеседников" покинул форум.

Обработка имен пользователей и передача сообщений

Теперь осталось добавить программу обработки имени пользователя и отсылки сообщения пользователям, подключенным к серверу. Начнем с того, что добавим в программу новый метод SendMessage(). Для этого сначала включите в секцию public: файла Unit1.h объявление метода:

```
void __fastcall SendMessage(AnsiString aMessage,
      AnsiString aFrom);
```

В файл Unit1.cpp включите программный код реализации метода SendMessage(), который получает два параметра класса AnsiString — aMessage и aFrom:

```
void __fastcall TForm1::SendMessage(AnsiString aMessage,
          AnsiString aFrom)
{
    for(int i=0;i<MyServer->Socket->ActiveConnections;i++)
    {
        if(aFrom == "Server")
            MyServer->Socket->Connections[i]->SendText(
                Format("%s",OPENARRAY(TVarRec,(aMessage))));
```

```
        else
            MyServer->Socket->Connections[i]->SendText(
                Format("%s said: %s",OPENARRAY(TVarRec,
                      (aFrom, aMessage))));
    }
}
```

Эта функция опрашивает в цикле все подключенные объекты `Socket` и отсылает через них сообщение вида *`User`* `said:` *`text`* (<Пользователь> заявил: <текст>). Если, например, пользователь `John` передал на сервер **`I am the walrus`** (Я — морж), то через все подключенные сокеты сервер разошлет сообщение `John said: I am the walrus` (Джон заявил: "Я — морж"). Если же автором заявления будет сам сервер (параметр `aFrom` содержит текст `"Server"`), то вводная часть *`User`* `said:` в рассылаемое сообщение не включается.

Сообщения, поступающие на сервер от клиентов, можно разделить на две категории: реплики, предназначенные для рассылки остальным участникам форума, и "приватное" сообщение, предназначенное только серверу, в котором пользователь передает свое имя.

Обработка события `OnClientRead` выполняется программой, представленной в листинге 12.1.

Листинг 12.1. Обработка события `OnClientRead`

```
void __fastcall TForm1::MyServerClientRead(TObject *Sender,
      TCustomWinSocket *Socket)
{
    AnsiString TextIn, CurrentName;
    int iIndex;

    TextIn = Socket->ReceiveText();
    iIndex = ConnectedList->IndexOfObject(Socket);
    if(iIndex == -1)
        return;

    TStringList *UserName = new TStringList();

    if(TextIn.Pos("UserName=") == 1)
    {
        // Определение имени пользователя
        UserName->Text = TextIn;
        ConnectedList->Strings[iIndex] =
              UserName->Values["UserName"];
        SendMessage(Format("%s has connected.",
                    OPENARRAY(TVarRec,
                    (UserName->Values["UserName"]))),"Server");
    }
    else
    {
        // Передача сообщения
        CurrentName = ConnectedList->Strings[iIndex];
        SendMessage(TextIn, CurrentName);
    }

    delete UserName;
}
```

Сначала в этой программе с помощью метода ReceiveText объекта Socket считывается принятый текст в переменную типа AnsiString. Затем определяется, включен ли этот объект в список подключенных клиентов ConnectedList; он должен там присутствовать, но лучше лишний раз проверить. Затем проверяется, не содержится ли в полученном сообщении текст "UserName=". Если дело обстоит именно так, то вслед за ним в сообщении должно идти имя пользователя. Этот второй компонент сообщения извлекается с помощью свойства Name=Value класса TStringList и помещается в список ConnectedList. Последний этап процедуры — рассылка всем остальным клиентам извещения о том, что к "беседе" подключился новый участник с таким-то именем.

Если же окажется. что сообщение начинается не с текста "UserName=", то полученный текст рассылается всем прочим участникам обсуждения с помощью метода SendMessage().

Итак, мы завершили разработку программы сервера, который поддерживает коллективное обсуждение через сеть Internet. Сервер позволяет множеству клиентов оперативно обмениваться текстовыми сообщениями. Позже мы добавим в эту программу возможность работы с протоколом HTTP, что позволит запускать его с помощью Web-броузера и просматривать на экране список подключенных клиентов. А сейчас сохраните проект — выберите в меню команду File⇨Save All.

Клиентское приложение для сервера ChatServer

В этом разделе мы рассмотрим программу клиентского приложения, которое будет работать с описанным выше сервером ChatServer. Это приложение будет использовать протокол TCP/IP. С помощью этого простого клиентского приложения пользователь сможет указать свое имя, выбрать сервер и порт, к которому собирается подключиться, отсылать и получать сообщения. При желании эту программу можно усовершенствовать — добавить в нее сопровождение списка пользователей, частных форумов и опции форматирования текстов. Все компоненты этого проекта находятся в папке Chat\Client на компакт-диске, который прилагается к этой книге. Имя файла проекта — Client.bpr.

Создание приложения

Начните работу с создания нового приложения в среде C++Builder. Для этого в меню File выберите команду New Application — будет сформирована пустая экранная форма и модель программного кода. Согласитесь с предлагаемыми по умолчанию именами этих компонентов программы и выберите в меню команду File⇨Save All. Я советую назвать новую папку Client, а файлу проекта присвоить имя Client.bpr. Далее займемся компоновкой экранной формы. Выполните следующие операции.

1. Поместите в экранную форму такие компоненты: два компонента TMemo, которым присвойте имена **MemoIn** и **MemoOut** и компонент TPanel, свойство Caption которого очистите.
2. Значения свойств align этих новых компонентов настройте следующим образом: для компонента MemoOut установите значение **alBottom**, для компонента Panel1 — **alTop**, а для компонента MemoIn — **alClient**.
3. Компоненту MemoIn присвойте статус "только для чтения", поскольку этот компонент будет только выводить сообщения, поступившие от сервера. Настройка статуса выполняется присвоением свойству ReadOnly элемента MemoIn значения **True**.
4. Сразу после запуска клиентское приложение еще не подключено к серверу программы. Поэтому имеет смысл в этом режиме заблокировать компонент MemoOut чтобы пользователь не пытался отсылать сообщения, прежде чем подключится к серверу. Присвойте свойству Enabled компонента MemoOut значение **False**.

5. Теперь, покончив с базовыми компонентами пользовательского интерфейса, займемся прочими.

   - Включите в состав Panel1 кнопку (компонент TButton) и присвойте ее свойству name значение **ConnectButton**, а свойству Caption — значение **Connect**.
   - Добавьте в панель еще одну кнопку и присвойте ее свойству name значение **DisconnectButton**, а свойству Caption — значение **Disconnect**.
   - Добавьте в Panel1 три компонента TEdit; присвойте им имена **UserNameEdit**, **PortEdit** и **ServerEdit**. Свойству Text всех трех новых компонентов присвойте значение **blank**.

6. Добавьте в Panel1 три компонента TLabel и установите в них надписи **User Name**, **Port** и **Server**.

Далее добавим в экранную форму Form1 компонент TClientSocket. Присвойте свойству name этого компонента значение **MySocket**. Класс TClientSocket инкапсулирует все функции сокета, необходимые для работы клиентского сетевого приложения. Изображение этого компонента на поле экранной формы напоминает штепсельную розетку, достаточно точно передавая его назначение.

Компоновка элементов управления на поле экранной формы должна выглядеть примерно так, как на рис. 12.2.

Рис. 12.2. Компоновка экранной формы клиентского приложения

Включим в программу обработчик событий OnClick кнопок. Ниже представлена программа обработки события OnClick для кнопки ConnectButton.

```
void __fastcall TForm1::ConnectButtonClick(TObject *Sender)
{
    MySocket->Address = ServerEdit->Text;
    MySocket->Port = PortEdit->Text.ToIntDef(1971);
    MySocket->Active = true;

    ConnectButton->Enabled = false;
    DisconnectButton->Enabled = true;
    MemoOut->Enabled = true;
}
```

Эта функция настраивает компонент MySocket соответственно содержимому полей Server и Port экранной формы, которые должны быть предварительно заполнены пользователем. Программа присваивает свойству Address компонента MySocket IP-адрес хоста. Метод ToIntDef(int) класса AnsiString обеспечивает установку порта по умолчанию в случае, если пользователь ввел в поле Port (элемент PortEdit) неверное значение. После настройки свойств Address и Port программа инициирует подключение — присваивает свойству Active компонента MySocket значение True. Далее блокируется кнопка ConnectButton, поскольку подключение уже произошло, и разблокируется кнопка DisconnectButton. Последняя операция — разблокирование компонента MemoOut, что позволяет пользователю вводить текст сообщения.

Обеспечив таким образом запуск клиентского приложения, нужно включить в программу средства прекращения сеанса связи с сервером. Для этого добавьте следующий программный код обработки события OnClick кнопки DisconnectButton.

```
void __fastcall TForm1::DisconnectButtonClick(TObject *Sender)
{
    MySocket->Active = false;

    ConnectButton->Enabled = true;
    DisconnectButton->Enabled = false;
    MemoOut->Enabled = false;
}
```

Функция разрыва соединения еще проще, чем функция подключения. Она прерывает соединение с сервером, присваивая значение **False** свойству Active компонента MySocket, разблокирует кнопку ConnectButton (это позволяет пользователю при необходимости повторно подключиться к серверу), блокирует кнопку DisconnectButton и поле MemoOut.

Передача серверу имени пользователя и других сообщений

Следующий этап — включение в программу функции отсылки на сервер имени, которое выбрал для себя пользователь, а также введенных им сообщений.

После подключения к серверу клиентское приложение должно передать информацию о том, кто вступает в общий "разговор". Добавьте следующий программный код обработки события OnConnect объекта MySocket:

```
void __fastcall TForm1::MySocketConnect(TObject *Sender,
     TCustomWinSocket *Socket)
{
    MemoIn->SetFocus();
    MemoIn->Lines->Add("Connected ...");
    MemoOut->SetFocus();
    MySocket->Socket->SendText(Format("UserName=%s",
           OPENARRAY(TVarRec, (UserNameEdit->Text))));
}
```

Сначала программа устанавливает фокус ввода экранной формы на элемент управления MemoIn. Это позволяет привлечь внимание пользователя к полю с сообщением о том, что подключение в серверу состоялось. Текст этого сообщения вводится в поле MemoIn вторым оператором программы. После этого фокус ввода переводится на элемент MemoOut, как бы предлагая пользователю ввести в него текст сообщения. Последний оператор программы передает введенный текст на сервер, причем сообщение форматируется таким образом, что перед введенным в поле текстом вставляется "UserName=". Именно в таком виде сервер воспринимает сообщение, в котором содержится имя только что подключившегося участника обсуждения. Благодаря этому сообщению на сервере будет иметься информация о том, кого представляет данное клиентское приложение.

После того как пользователь введет какой-либо текст в поле MemoOut и нажмет клавишу <Enter>, сообщение передается на сервер. Это выполняется обработчиком события OnKeyPress компонента MemoOut:

```
void __fastcall TForm1::MemoOutKeyPress(TObject *Sender, char &Key)
{
    if(Key == VK_RETURN)
```

```
    {
        MySocket->Socket->SendText(MemoOut->Text);
        MemoOut->Lines->Clear();
        Key = 0;
    }
}
```

Функция проверяет, не было ли вызвано событие нажатием клавиши с виртуальным кодом `VK_RETURN`. Если это так, то содержимое компонента `MemoOut` пересылается на сервер, а `MemoOut` очищается. Последний оператор программы сбрасывает в нуль значение переменной `Key`; тем самым пользователю разрешается начать ввод нового сообщения.

Обработка сообщений, поступивших от сервера

Сообщения, которые поступают от сервера, должны быть отображены для пользователя. Это выполняется обработчиком события `OnRead()` компонента `MySocket`:

```
void __fastcall TForm1::MySocketRead(TObject *Sender,
      TCustomWinSocket *Socket)
{
    MemoIn->SetFocus();
    MemoIn->Lines->Add(Socket->ReceiveText());
    MemoOut->SetFocus();
}
```

Первый оператор этой программы устанавливает фокус ввода экранной формы на компонент `MemoIn`; тем самым пользователю предоставляется возможность просматривать в режиме прокрутки текст, хранящийся в буфере этого компонента. Затем в компонент `MemoIn` вводится текст сообщения, полученного объектом `Socket`. Последний оператор переносит фокус ввода на элемент `MemoOut`.

На этом разработка клиентского приложения завершается. Выберите в меню команду File⇨Save All и сохраните файлы программы.

Для проверки функционирования разработанных приложений выполните следующие операции.

- Запустите на выполнение сервер-приложение `ChatServer`.
- Настройте порт сервера (по умолчанию используется порт 1971).
- Щелкните на кнопке Start в экранной форме приложения сервера.
- Запустите клиентское приложение.
- Установите тот же порт, что и в приложении сервера.
- Установите адрес сервера `127.0.0.1`. Это — адрес локального хоста. Он будет использоваться только тем компьютером, на котором вы сейчас работаете.
- Введите какое-нибудь имя пользователя в поле User Name и щелкните на кнопке Connect.

Приложение для работы с электронной почтой

Протокол почтовой службы *Post Office Protocol* (POP) и простой протокол передачи почты *Simple Mail Transfer Protocol* (SMTP) — это два самых распространенных протокола, которые используются при создании приложений для электронной почты. Протокол POP используется для получения сообщений по электронной почте от POP-серверов,

а протокол SMTP — для отсылки таких сообщений через SMTP-сервер. В этом разделе мы рассмотрим несложное приложение, которое позволяет посылать и принимать сообщения по электронной почте. Все программные компоненты этого проекта вы найдете в папке Mail на компакт-диске, который прилагается к этой книге. Имя файла проекта — Mail.bpr.

Создание приложения

Создайте новый проект. Для этого выберите в меню File⇨New Application. В ответ C++Builder создаст пустую экранную форму Form1 и файл программного кода Unit1.cpp. Сохраните новый проект — выберите в меню команду File⇨Save All. Я рекомендую создать для этого проекта отдельную папку Mail и так же назвать сам проект. Теперь скомпонуйте экранную форму, с помощью которой пользователь будет общаться с приложением.

1. Поместите на поле экранной формы компонент TListView , который вы найдете на вкладке Win32. Свойству align нового элемента управления присвойте значение **alBottom** и оставьте предлагаемое по умолчанию имя компонента ListView1.
2. Добавьте на поле экранной формы две кнопки. Первую назовите **CheckMail** и установите текст надписи на ней **Check Mail**, а вторую назовите **NewButton** и надпись на ней должна быть **New Mail**. Тексты надписей — это значения свойств Caption.
3. Включите в экранную форму три компонента TEdit и назовите их **UserEdit**, **PasswordEdit** и **HostEdit**. Свойству PasswordChar элемента PasswordEdit присвойте значение * или любой другой символ — именно он будет выводиться в этом поле вместо пароля при его вводе пользователем. Сейчас можно и назначить почтовый сервер по умолчанию, который будет работать с этим приложением. Имя сервера по умолчанию (например, **mail.yourserver.com**) можно задать в качестве значения свойства Text элемента HostEdit.
4. Добавьте три компонента TLabel и установите следующие значения свойств Caption для них: **User Name**, **Password** и **Server**.
5. Теперь откройте редактор колонок — дважды щелкните на поле компонента ListView1. Добавьте в него две колонки: первая будет называться **From** (От), а вторая — **Subject** (Содержание).
6. Присвойте свойству ViewStyle компонента ListView1 значение **vsReport**.
7. Включите в экранную форму компонент TNMPOP3 (вы найдете его на вкладке FastNet). На этой стадии разработки экранная форма нового приложения должна выглядеть примерно так, как показано на рис. 12.3.

Включение в приложение компонентов поддержки протокола POP

После того так с компоновкой экранной формы приложения будет в основном покончено, включите в файл заголовка Unit1.h объявление нескольких общедоступных переменных — добавьте в него приведенные ниже операторы.

```
bool bConnected, bSummary
int myId
```

После этого можно приступить к включению в приложение основных функций. Начнем с тех, которые обеспечивают соединение с POP-сервером, аутентификацию подключившегося клиента и загрузку списка поступивших, но еще не прочитанных писем. В обработчик события OnClick компонента CheckMail добавьте программный код из листинга 12.2.

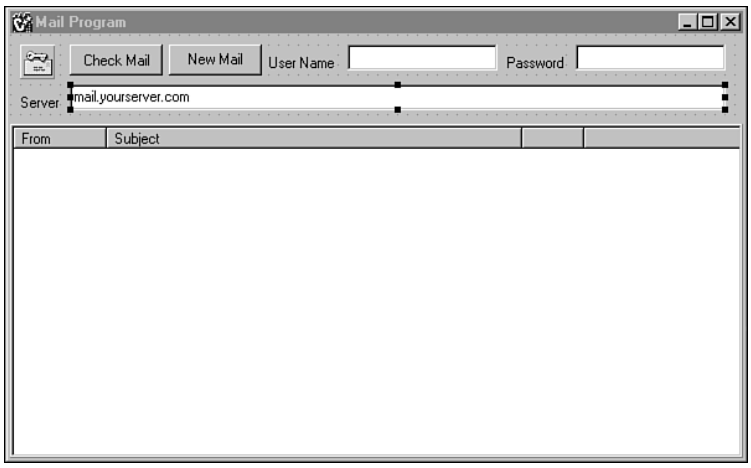


Рис. 12.3. Компоновка диалогового окна приложения Mail

Листинг 12.2. Обработчик события `OnClick` компонента `CheckMail`

```
void __fastcall TForm1::CheckMailClick(TObject *Sender)
{
    bConnected = false;
    NMPOP31->UserID = UserEdit->Text;
    NMPOP31->Password = PasswordEdit->Text;
    NMPOP31->Host = HostEdit->Text;
    NMPOP31->Connect();

    if(NMPOP31->Connected)
    {
        if(NMPOP31->MailCount > 0)
        {
            ListView1->Items->Clear();
            bSummary = true;
            for(int i = 0; i < NMPOP31->MailCount; i++)
            {
                myId = i + 1;
                NMPOP31->GetSummary(myId);
            }
        }
        else
            // "Новые сообщения отсутствуют"
            ShowMessage("No messages waiting");

        NMPOP31->Disconnect();
    }
}
```

Первый оператор в этой функции присваивает общедоступному флагу — переменной `bConnected` — значение **`false`**, а затем устанавливаются значения свойств компонента

NMPOP31. Свойство UserID — это имя пользователя, который обращается к услугам POP-сервера, свойство Password — пароль пользователя, Host — имя POP-сервера, например **mail.myserver.com**.

После установки значений свойств программа подключается к серверу, вызывая метод Connect() объекта NMPOP31. Запросив подключение, программа проверяет, установлено ли свойство Connected объекта класса TNMPOP3, т.е. произошло ли подключение. Если дело обстоит именно так, то начинается обработка списка сообщений, имеющихся для клиента на сервере. Сначала очищаются все элементы объекта списка ListView1, а затем общедоступному флагу — переменной bSummary — присваивается значение **true**. Тем самым обработчик события CheckMailClick извещает остальные компоненты приложения, что считаны сведения о сообщениях, но не сами тексты сообщений.

Далее организуется цикл считывания резюме сообщений с сервера. Количество циклов равно текущему значению свойства MailCount компонента TNMPOP3. Первая операция в теле цикла — присвоение глобальной переменной myId значения, равного увеличенному на 1 номеру цикла. Второй оператор в теле цикла вызывает метод GetSummary() компонента TNMPOP3. Этот метод извлекает текст резюме сообщения, номер которого соответствует значению переменной myId, а также возбуждает событие RetrieveEnd компонента TNMPOP3.

В случае, если на сервере отсутствуют сообщения для подключившегося клиента, программа выводит сообщение No Messages Waiting (Новые сообщения отсутствуют).

Последняя операция в обработчике события CheckMailClick — вызов метода Disconnect() объекта NMPOP31; этот метод отсоединяет клиента от сервера.

Помимо обработчика события Click, в программу нужно добавить и обработчик события RetrieveEnd того же объекта NMPOP31. Включите в модуль реализации текст метода NMPOP31RetrieveEnd, приведенный ниже.

```
void __fastcall TForm1::NMPOP31RetrieveEnd(TObject *Sender)
{
    if(bSummary)
    {
        TListItem *Temp = ListView1->Items->Add();

        Temp->Caption = NMPOP31->Summary->From;
        Temp->SubItems->Add(NMPOP31->Summary->Subject);
        Temp->SubItems->Add(myId);
    }
}
```

Событие RetrieveEnd возбуждается в двух случаях: когда появляется резюме сообщения и когда извлекается само сообщение. Уловить конкретную ситуацию помогает глобальная переменная-флаг bSummary: этому флагу присваивается значение **True**, когда выполняются операции с резюме сообщений, и значение **False**, когда извлекаются сами сообщения. Если обрабатываются резюме сообщений, то программа добавляет очередной элемент в объект списка ListView1. Для этого сначала создается переменная Temp класса TListItem и приравнивается значению, возвращаемому методом ListView1->Items->Add(). Таким образом создается новый элемент списка ListView1. Далее свойству Caption нового элемента списка ListItem присваивается значение отправителя сообщения. Оно извлекается из свойства From объекта Summary в NMPOP31. Затем в качестве компонентов элемента списка добавляются значения свойства Subject объекта Summary в NMPOP31 и текущего ID сообщения (глобальная переменная myId). После выполнения этих операций для всех сообщений в списке ListView1

будут созданы элементы, каждый из которых представляет одно из сообщений, имеющихся для пользователя на почтовом сервере. Манипулируя этим списком, пользователь может выбрать те сообщения, с которыми желает познакомиться подробно (не исключено, что часть сообщений пользователь отправит в корзину, не читая).

Извлечение и просмотр сообщений

Для извлечения и просмотра сообщений понадобится создать в приложении еще одну экранную форму. Выберите в меню File команду New Form — в ответ среда разработки C++Builder сформирует новую пустую экранную форму и присвоит ей имя `Form2`, а также новый модуль программного кода и присвоит ему имя `Unit2`. Скомпонуйте экранную форму следующим образом.

1. Добавьте в экранную форму компонент `TPanel` и присвойте значение **`alTop`** его свойству `align`. Свойству `Caption` этого компонента присвойте значение **`blank`**
2. В состав `Panel1` включите четыре компонента `TLabel`, которым присвойте наименования **`Label1`**, **`Label2`**, **`FromLabel`**, **`SubjectLabel`**.
3. Свойствам `Caption` первых двух элементов надписей присвойте следующие значения: элементу `Label1` — значение **`From`** (От), а элементу `Label2` — значение **`Subject`** (Содержимое). Для двух других элементов пока что оставьте те значения свойства `Caption`, которые предлагаются по умолчанию средой разработки: **`FromLabel`** и **`SubjectLabel`**.
4. Установите на панель `Panel` кнопку (компонент `TButton`) и присвойте ей наименование **`CloseButton`**.
5. Добавьте в экранную форму `Form2` компонент `TMemo` и присвойте его свойству `align` значение **`alClient`**.
6. Свойству `name` компонента `TMemo` присвойте значение **`MailMemo`**. На этой стадии разработки экранная форма `Form2` должна выглядеть как на рис. 12.4.

Рис. 12.4. Компоновка диалогового окна считывания сообщений, поступивших по электронной почте

В модуль этой экранной формы — обработчик события OnClick кнопки CloseButton — нужно добавить единственный оператор Close();

Для того чтобы в этом диалоговом окне можно было просматривать содержимое поступивших сообщений, нужно прежде всего включить модуль Unit2 в главную экранную форму приложения (Form1). Вызовите в окно редактора программного кода файл Unit1.cpp, выберите в меню File команду Include File Header и выберите в открывшемся окне файл Unit2. Это позволит манипулировать копией формы Form2 в форме Form1. Теперь можно написать программный код, который запустит копию формы Form2 и заполнит ее поля информацией, если пользователь пожелает ознакомиться с содержимым присланных сообщений.

Сначала присвойте значение **True** свойству RowSelect списка ListView1 — тем самым пользователю будет позволено выбирать элементы в списке с помощью щелчка мыши в любом месте соответствующей строки, а не только на ее первом элементе. Далее добавьте приведенный ниже программный код в обработчик сообщения OnDblClick элемента управления ListView1.

```
void __fastcall TForm1::ListView1DblClick(TObject *Sender)
{
    if(!NMPOP31->Connected)
        NMPOP31->Connect();
    if(ListView1->SelCount > 0)
    {
        bSummary = false;
        NMPOP31->GetMailMessage(
              ListView1->Selected->SubItems->Strings[1].ToIntDef(0));
    }
    NMPOP31->Disconnect();
}
```

В этом обработчике сначала проверяется, подключено ли приложение в текущий момент к серверу. Если это не так, то вызывается метод Connect() объекта NMPOP31 и выполняется подключение. Далее выясняется, выбраны ли какие-либо элементы в списке ListView; таким способом предотвращается появление ошибки, которая возникает в случае, если в момент выполнения двойного щелчка на поле элемента ListView в нем не было выбрано ни одно сообщение. Если какой-либо элемент списка выбран, то глобальной переменной bSummary присваивается значение **False**, и тем самым остальные компоненты программы (в частности, обработчик события RetrieveEnd объекта NMPOP31) извещаются о том, что извлекаться будет именно содержимое сообщений, а не резюме. В следующей строке вызывается метод GetMailMessage() объекта NMPOP31. В качестве аргумента методу передается идентификатор сообщения, с которым пользователь пожелал ознакомиться. Этот идентификатор извлекается из второго компонента (свойства SubItems) выбранного элемента списка. Метод ToIntDef() класса AnsiString преобразует строковое значение идентификатора в числовое.

Последняя операция в этом обработчике — отключение приложения от сервера. Вызов метода GetMailMessage() возбуждает событие OnRetrieveEnd, но на сей раз пользователь желает просмотреть не резюме, а содержимое сообщения. Поэтому нам придется дополнить разработанный ранее обработчик события OnRetrieveEnd объекта NMPOP31. Добавьте в его программный код фрагмент, выделенный в листинге 12.3 полужирным шрифтом.

Листинг 12.3. Обработчик события `OnRetrieveEnd`

```
void __fastcall TForm1::NMPOP31RetrieveEnd(TObject *Sender)
{
    if(bSummary)
    {
        TListItem *Temp = ListView1->Items->Add();

        Temp->Caption = NMPOP31->Summary->From;
        Temp->SubItems->Add(NMPOP31->Summary->Subject);
        Temp->SubItems->Add(myId);
    }
    // Новый программный код
    else
    {
        TForm2 *Temp = new TForm2(NULL);
        Temp->MailMemo->Lines->Assign(
               NMPOP31->MailMessage->Body);
        Temp->FromLabel->Caption = NMPOP31->MailMessage->From;
        Temp->SubjectLabel->Caption =
               NMPOP31->MailMessage->Subject;
        Temp->Show();
    }
}
```

Первый оператор в добавленном фрагменте создает новый экземпляр класса формы `TForm2`, а затем устанавливаются значения свойств нового объекта `Temp` соответственно реквизитам почтового сообщения, которое пользователь выбрал для просмотра. Сначала вызывается метод `Assign()` объекта, который входит в состав объекта `MailMemo`. Объект `Lines` имеет тип `TStringList`. В результате в объект `MailMemo` копируется тело сообщения — содержимое `Body` объекта `MailMessage`, который, в свою очередь, входит в состав объекта `NMPOP31`. Далее устанавливаются тексты надписей элементов `FromLabel` и `SubjectLabel` (значения свойств `Caption`). Они считываются из свойств `From` и `Subject` объектов `MailMessage` в `NMPOP31`. Последняя операция — вызов метода `Show()` объекта экранной формы `Temp`.

После того как в приложение включен обработчик события, оно готово к приему и отображению для пользователя полученных по электронной почте сообщений. Но прием и отображение сообщений — это только полдела. Нужно обеспечить также и отсылку почтовых сообщений. Этим мы займемся в следующем разделе.

Формирование и отсылка сообщений по электронной почте

Нам потребуется включить в приложение, во-первых, новую экранную форму, в которой можно было бы подготовить сообщение к отсылке, а во-вторых, программный код для подключения к SMTP-серверу и отсылки ему сформированного сообщения. Для включения в приложение еще одной экранной формы выберите в меню File команду New Form — среда разработки в ответ сформирует пустую экранную форму `Form3` и модуль программного кода `Unit3` и включит их в состав проекта. Скомпонуйте экранную форму `Form3`, включив в нее элементы пользовательского интерфейса.

1. Добавьте в экранную форму компонент `TPanel` и присвойте значение **`alTop`** его свойству `align`. Свойству `Caption` этого компонента присвойте значение **`blank`**

2. Добавьте в экранную форму Form3 компонент TMemo и присвойте его свойству align значение **alClient**. Свойству name компонента TMemo присвойте значение **MessageMemo**.
3. В состав Panel1 включите пять компонентов TEdit, которым присвойте наименования **ToEdit**, **FromEdit**, **SubjectEdit**, **CCEdit** и **BCCEdit**. Во всех элементах типа TEdit очистите свойство Text.
4. В состав Panel1 включите еще и пять компонентов TLabel. Свойствам Caption этих элементов присвойте значения **To**, **From**, **Subject**, **CC** и **BCC**.
5. Включите в экранную форму компонент TNMSMTP (вы найдете его на вкладке FastNet). Оставьте значение его свойства name таким, как предлагается по умолчанию средой разработки C++Builder, — **NMSMTP1**.
6. Добавьте на поле экранной формы две кнопки (компонента TButton). Первую назовите **SendButton**, а надпись на ней (свойство Caption) должна быть **Send**, а вторую назовите **CancelButton**, а надпись на ней, соответственно, **Cancel**. На этой стадии разработки экранная форма должна выглядеть так, как показано на рис. 12.5.

Рис. 12.5. Компоновка экранной формы подготовки сообщений к отсылке по электронной почте

Теперь займемся программным кодом Сначала включите в модуль Unit3 директиву включения файла заголовка Unit1.h — воспользуйтесь командой Include File Header меню File и выберите в открывшемся окне файл Unit1. Далее включите в приложение работы с электронной почтой средства поддержки протокола SMTP. В обработчик события OnClick кнопки SendButton добавьте текст программы, представленный в листинге 12.4.

Листинг 12.4. Обработка события OnClick кнопки SendButton

```
void __fastcall TForm3::SendButtonClick(TObject *Sender)
{
    NMSMTP1->PostMessage->ToAddress->Clear();
    NMSMTP1->PostMessage->ToBlindCarbonCopy->Clear();
```

```
    NMSMTP1->PostMessage->ToCarbonCopy->Clear();

    NMSMTP1->PostMessage->ToAddress->CommaText = ToEdit->Text;
    NMSMTP1->PostMessage->FromAddress = FromEdit->Text;
    NMSMTP1->PostMessage->ReplyTo = FromEdit->Text;
    NMSMTP1->PostMessage->ToBlindCarbonCopy->CommaText =
            BCCEdit->Text;
    NMSMTP1->PostMessage->ToCarbonCopy->CommaText = CCEdit->Text;
    NMSMTP1->PostMessage->Body->Assign(MessageMemo->Lines);
    NMSMTP1->PostMessage->Subject = SubjectEdit->Text;


    NMSMTP1->PostMessage->LocalProgram = "My Emailer";

    if(NMSMTP1->Connected)
        NMSMTP1->Disconnect();

    NMSMTP1->UserID = Form1->UserEdit->Text;
    NMSMTP1->Host = Form1->HostEdit->Text;
    NMSMTP1->Connect();
    NMSMTP1->SendMail ();
}
```

Сначала в этом обработчике подготавливается к работе объект NMSMTP1. Основным информационным компонентом в объекте класса TNMSMTP является PostMessage, в котором содержится вся информация, касающаяся сообщения, — его тело и список рассылки (Send To).

Далее устанавливаются компоненты сообщения, предназначенного к отсылке. Первым делом установите значение свойства ToAddress объекта PostMessage, которое имеет тип TStringList и содержит все адреса рассылки. С помощью свойства CommaText объекта ToAddress программа позволяет пользователю назначить множество получателей отсылаемого сообщения в поле To, причем в качестве разделителя используется запятая или пробел. Далее свойству FromAddress присваивается значение, введенное ранее в поле From (элемент управления FromEdit), и устанавливается значение свойства ToBlindCarbonCopy, которое аналогично свойству ToAddress; разница между ними состоит в том, что адреса, перечисленные в ToBlindCarbonCopy, не включаются в поля To и CC почтового сообщения. Следующая операция — установка значения свойства ToCarbonCopy, которое аналогично значению свойства ToBlindCarbonCopy с той только разницей, что перечисленные в нем адреса упоминаются в заголовке почтового сообщения.

Последняя операция — перепись содержимого сообщения из элемента экранной формы MessageMemo в свойство Body объекта PostMessage. Затем устанавливается значение свойства Subject — в него переписывается текст из поля Subject (элемента управления SubjectEdit) экранной формы. В объекте PostMessage остается после этого заполнить только свойство LocalProgram; в это поле можно записать все, что угодно, и обычно оно используется при создании интегрированных программ работы с электронной почтой для хранения маркера отсылки сообщения.

Установив в объекте PostMessage все параметры отсылаемого сообщения, нужно заполнить и другие свойства объекта NMSMTP1, после чего можно будет отослать сообщение. Сначала нужно проверить, не присоединен ли объект NMSMTP1 к серверу, и, если это

так, то разорвать соединение. Далее нужно установить значения свойств UserID и Host — этим свойствам присваиваются значения, введенные в элементы управления главной экранной формы приложения Form1. Последняя операция — подключение приложения к SMTP-серверу и вызов метода SendMail(), который и отвечает за отсылку скомпонованного сообщения.

В этом модуле осталось организовать реакцию приложения на события, извещающие об успешном завершении передачи сообщения (OnSuccess) или о неудачной попытке (OnFailure), и добавить оператор Close в обработчик события OnClick кнопки CancelButton. Текст соответствующих функций приведен ниже.

```
void __fastcall TForm3::NMSMTP1Success(TObject *Sender)
{
    // " Отсылка сообщения выполнена успешно "
    ShowMessage("Send Message Successful");
    Close();
}

void __fastcall TForm3::NMSMTP1Failure(TObject *Sender)
{
    // " Не удалось отослать сообщение "
    ShowMessage("Send Message Failed");
}

void __fastcall TForm3::CancelButtonClick(TObject *Sender)
{
    Close();
}
```

Включите в модуль Unit1 директиву #include "Unit3.h" и добавьте в него программу обработки события OnClick кнопки New Mail экранной формы Form1. Этот новый обработчик события должен выполнять операции, сопряженные с отсылкой нового сообщения по электронной почте. Для включения директивы #include нужно выполнить следующее: открыв в редакторе программного кода файл Unit1.cpp, выбрать команду File⇨Include Unit Header и выделить в открывшемся диалоговом окне Unit3, после чего добавить в модуль следующий текст обработчика события OnClick компонента NewMail:

```
void __fastcall TForm1::NewButtonClick(TObject *Sender)
{
    TForm3 *OutMail = new TForm3(NULL);
    OutMail->ShowModal();
}
```

Этот обработчик события сформирует новый экземпляр класса экранной формы Form3 и передаст управление этому диалоговому окну в модальном режиме. Теперь можно сохранить все файлы проекта и запустить его на выполнение. Примите мои поздравления — вы создали клиентское приложение, которое способно получать и передавать сообщения по электронной почте.

В демонстрационной версии этого приложения сообщение, поступившее по электронной почте, остается на сервере в то время, как пользователь читает его. Большинство коммерческих программных продуктов позволяет либо сохранять послание на сервере, либо удалять его — в зависимости от команды пользователя. Класс TNMPOP3 располагает методом DeleteMailMessage(int Number), который выполняет удаление сообщения.

Разработка HTTP-сервера

Пожалуй, самым распространенным в мире на сегодняшний день протоколом является *HTTP — Hypertext Transfer Protocol* (протокол передачи гипертекстов). Сейчас практически каждый, кто пользуется компьютером, пользуется и Web-броузером, заглядывая на те или иные Web-сайты. Программа-броузер является клиентской программой по отношению к HTTP-серверу, в качестве которого выступает Web-сайт. В этом разделе вы познакомитесь с некоторыми функциональными возможностями HTTP-сервера, которые мы включим в созданное ранее приложение `ChatServer`. В частности вы узнаете, как организовать в программе выполнение следующих операций:

- удаленный просмотр статуса сервера;
- удаленный запуск сервера;
- просмотр текущего списка пользователей.

Включение Web Server Socket в приложение

Начните с того, что снова откройте проект приложения `Chat Server`, который мы рассматривали в одном из предыдущих разделов этой главы. Добавьте в экранную форму главного окна этого приложения второй компонент `TServerSocket` (его можно отыскать на вкладке Internet). Присвойте ему наименование `HttpServer`. Установите в свойстве `Port` этого компонента номер порта по своему выбору. Как правило, для HTML-серверов выбирается порт 80. Если вы уже поддерживаете работу собственного Web-сервера или любого другого сервера, то выберите номер порта, отличный от 80. Для этого примера я выбрал значение 8000. Нужно также определиться с IP-адресом. Это можно сделать, запустив из командной строки программу `ipconfig.exe`. Как правило, компьютер может использовать адрес `127.0.0.1` для обращения к самому себе, т.е. это адрес локального хоста, но если вы работаете без брандмауэра или другого подобного средства обеспечения безопасности, то ваш IP-адрес может быть и другим. Последний штрих: присвойте свойству `Active` объекта `HttpServer` значение `True`.

Обработка запросов к серверу

Web-сервер, как правило, должен реагировать на запросы разных типов и в ответ передавать клиентам файлы и другую информацию. Поскольку в этом примере мы намерены наделить наш сервер только базовыми функциональными возможностями — он будет поддерживать только три функции — вы будете избавлены от необходимости программировать множество разнообразных опций, которые обычно присутствуют в профессиональных приложениях такого типа. Наш сервер будет поддерживать функции `Status`, `Start` и `Users`. Каждую из них мы будем считать отдельным типом запроса к HTML-странице. Запросы от Web-броузера, которые активизируют перечисленные функции, должны иметь следующий вид (полагая, что вы подставите собственный IP-адрес вместо `127.0.0.1`):

- `http://127.0.0.1:8000/Status.htm`
- `http://127.0.0.1:8000/Start.htm`
- `http://127.0.0.1:8000/Users.htm`

Добавьте в обработчик события `OnClientConnect` компонента `HttpServer` текст программы из листинга 12.5.

Листинг 12.5. Обработчик события `OnClientConnect`

```
void __fastcall TForm1::HttpServerClientConnect(TObject *Sender,
      TCustomWinSocket *Socket)
{
    AnsiString aRequest, aResponse;

    aRequest = Socket->ReceiveText();

 if(aRequest.Pos("Status.htm") > 0)
    {
        aResponse =
        "<html><head><title>Status</title></head><body>Status :";
        aResponse += (MyServer->Active) ? "Running" : "Stopped";
        if(MyServer->Active)
        {
            aResponse +=
                  Format("<BR>Port: %d<BR>%d Users Connected",
                     OPENARRAY(TVarRec,(MyServer->Port,
                         MyServer->Socket->ActiveConnections)));
        }
        aResponse += "</body></html>";
    }
    else
    {
        if(aRequest.Pos("Start.htm") > 0)
        {
            if(!MyServer->Active)
                StartButton->Click();
            aResponse = "<html><head><title>Start</title></head>"
                        "<body>Started</body></html>";
        }
        else
        {
            if(aRequest.Pos("Users.htm") > 0)
            {
                AnsiString aHead;

                aResponse = "";
                aHead = "<title>";
                for(int i=0; i< ConnectedList->Count; i++)
                {
                    aResponse += ConnectedList->Strings[i] +
                        "<BR>";
                }
                aHead += ConnectedList->CommaText;
                aHead += "</title>";
                if(aResponse.Length() == 0)
                    aResponse = "No Users Connected";
                aResponse = "<html><head>" + aHead +
                            "</head><body>User List<br>" +
```

```
                            aResponse +
                           "</body></html>";
            }
            else
            {
                aResponse =
   "<html><head><body>Invalid Request</body></html>";
            }
        }
    }

    Socket->SendText (aResponse);
    Socket->Close();
}
```

Первая операция в этой программе — сохранение в переменной типа `AnsiString` запроса, поступившего от броузера. Далее определяется тип запроса. Для этого выясняется, не присутствует ли в тексте запроса наименование одной из заданных страниц. Если дело обстоит именно так, т.е. в тексте запроса обнаружено присутствие литералов `Status.htm`, `Start.htm` или `Users.htm`, то соответствующий запрос обрабатывается. В противном случае формируется ответ, в котором содержится сообщение `Invalid Request` (Ошибочный запрос).

Если запрос касается статуса (т.е. запрос завершается именем страницы `Status.htm`), то программа выясняет, начал ли сервер функционировать. Если он находится в активном состоянии (это выясняется по состоянию члена `Active` объекта `MyServer`), то программа возвращает клиенту сообщение об этом, сопровождаемое номером порта и информацией о количестве пользователей, зарегистрировавшихся в текущий момент на сервере. Если от клиента (броузера) получен запрос типа `start`, то анализируется, не запущен ли сервер ранее, и, если это не так, то сервер активизируется, о чем и извещается клиент. Если же получен запрос на передачу списка текущих пользователей, то программа формирует такой список на основании информации, которая содержится в элементе управления `ConnectedList`.

После того как строка ответного сообщения будет сформирована, программа пересылает ее Web-броузеру, используя для этого метод `SendText()` объекта `Socket`. Последняя операция — вызов метода `Close()` объекта `Socket`; это позволяет Web-броузеру вернуть управление клиенту и обработать возвращенный сервером HTML-текст.

Используя этот сервер в качестве базовой "конструкции", можно организовать выполнение множества разнообразных операций. Например, можно расширить набор запросов и выполнять удаленное администрирование сервера коллективного обсуждения. Можно обеспечить передачу на Web-броузеры страниц с сервера и даже запускать на выполнение CGI-приложения. Можно рассматривать описанный в этом разделе сервер как первую ступеньку на пути к созданию собственных приложений с HTTP-сервером. А сейчас сохраните проект. Когда будет время, откомпилируйте и запустите его на выполнение.

Клиентское приложение, использующее протокол FTP

Многие полагают, что с расширением сети World Wide Web и сферы применения протокола HTTP популярность протокола *FTP — File Transfer Protocol* (Протокол передачи файлов) пошла на убыль. Но это далеко не так. Просто большинство Web-броузеров имеет встроенные средства работы с протоколом FTP, а потому потребность в специализированных кли-

ентских приложениях на базе этого протокола действительно снизилась. Но есть еще множество приложений, построенных исключительно на базе этого протокола, включая и приложения с автоматическим обновлением компонентов.

В этом разделе мы разработаем простое клиентское приложение с использованием протокола FTP на базе компонента TNMFTP, который вы можете отыскать на вкладке FastNet. Все программные компоненты этого проекта вы найдете в папке Ftp на компакт-диске, который прилагается к этой книге. Имя файла проекта — Ftp.bpr.

Создание приложения

Как обычно, первый шаг — создание нового проекта. Для этого выберите в меню File⇨New Application. В ответ C++Builder создаст пустую экранную форму Form1 и файл программного кода Unit1.cpp. Сохраните новый проект — выберите в меню команду File⇨Save All. Я рекомендую создать для этого проекта отдельную папку Ftp и так же назвать сам проект. Далее выполните следующие операции.

1. Добавьте в экранную форму компонент TPanel и присвойте значение **alTop** его свойству align. Свойству Caption этого компонента присвойте значение **blank**. Включите в экранную форму компонент TNMFTP, который находится на вкладке FastNet, и назовите его **MyFtp**.
2. Поместите на поле экранной формы компонент TListView , который вы найдете на вкладке Win32. Свойству align нового элемента управления присвойте значение **alClient**, а свойству name — значение **MyTree**.
3. Поместите на поле экранной формы компонент TImageList, который вы найдете на вкладке Win32. Оставьте предлагаемое по умолчание наименование этого элемента **ImageList1**.
4. Добавьте в Panel1 три компонента TLabel и установите для них следующие значения свойств Caption: **User**, **Password** и **Server**.
5. Добавьте три компонента TEdit, которым присвойте наименования **UserEdit**, **PasswordEdit** и **ServerEdit**. Установите для этих элементов управления значения по умолчанию: соответственно, **anonymous**, **mail@mail.com** и **ftp.yourserver.com**. Свойству PasswordChar элемента PasswordEdit присвойте значение *****.
6. Добавьте в Panel1 три кнопки (компонента TButton) и назовите их **StartButton**, **StopButton** и **UploadButton**. Надписи на кнопках (значения свойств Caption) должны быть, соответственно, **Start**, **Stop** и **Upload**. В результате всех этих операций экранная форма нового приложения должна выглядеть примерно так, как на рис. 12.6.

Теперь в добавьте в элемент ImageList1 изображения. Дважды щелкните на поле элемента ImageList1 и выберите act. Я предпочитаю использовать изображения из папки Program Files\Common Files\Borland Shared\Images\Buttons и рекомендую добавить из нее файлы fldropen.bmp и filenew.bmp. Поскольку эти растровые изображения имеют размер 16×32 пикселей, а компонент Image настроен на размер 16×16, то появится диалоговое окно, в котором запрашивается, не согласны ли вы скорректировать размер изображений. Выберите Yes и удалите ненужную часть изображения из компонента. Теперь у вас два изображения: одно представляет папку (индекс этого изображения равен 0), а второе — файл (индекс этого изображения равен 1).

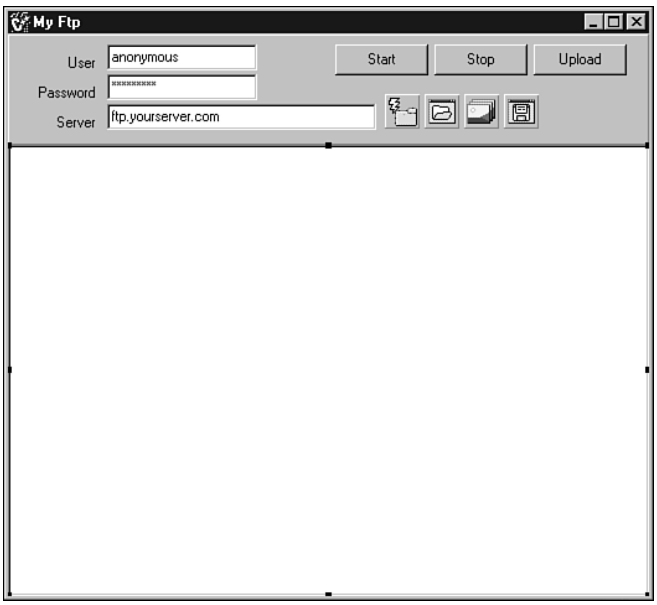


Рис. 12.6. Диалоговое окно программы работы с FTP-сервером

Подключение к FTP-серверу

На этом этапе разработки приложения нужно добавить в него средства подключения к удаленному FTP-серверу. Сначала добавьте следующий программный код в обработчик события `OnClick` кнопки `StartButton`:

```
void __fastcall TForm1::StartButtonClick(TObject *Sender)
{
    MyFtp->Host = ServerEdit->Text;
    MyFtp->UserID = UserEdit->Text;
    MyFtp->Password = PasswordEdit->Text;
    MyFtp->Connect();
    StartButton->Enabled = false;
    StopButton->Enabled = true;
    MyTree->Items->Clear();
    DoList();
}
```

Алгоритм работы этой программы достаточно очевиден — она аналогична программам, которые уже рассматривались в этой главе. Работа начинается с установки свойства `Host` объекта `MyFtp` — этому свойству присваивается выбранный адрес сервера. Затем в объекте `MyFtp` устанавливаются ID пользователя (свойство `UserID`) и пароль (свойство `Password`). Я разрешаю на своем сайте анонимный доступ, который позволяет задать имя пользователя `anonymous`, а в качестве пароля — адрес электронной почты. После этого выполняется подключение к серверу (вызывается метод `Connect()` объекта `MyFtp`), блокируется кнопка Start и разблокируется кнопка Stop. Далее очищается контейнер `Items` объекта `MyTree` и вызывается функция `DoList()`, которую мы рассмотрим чуть позже.

Составление перечня содержимого сервера

Теперь разработаем функцию, которая будет выводить на экран содержимое каталога на сервере. Присвойте свойству ParseList объекта MyFtp значение **True**; это позволит сохранить информацию о каталоге, которая содержится в объекте типа TFTPDirectoryList. Доступ к ней открывается посредством свойства времени выполнения FTPDirectoryList объекта TNMFTP. Далее добавьте в файл Unit1.h, в его секцию общедоступных (public) членов, объявление новой функции:

```
void __fastcall DoList();
```

Затем в конец файла Unit1.cpp добавьте текст программы из листинга 12.6.

Листинг 12.6. Функция DoList()

```
void __fastcall TForm1::DoList()
{
    TTreeNode *Temp, *Root;
    int i;
    TCursor Save_Cursor = Screen->Cursor;

    // Установить курсор в виде песочных часов.
    Screen->Cursor = crHourGlass;
    Root = MyTree->Selected;
    MyFtp->List();
    MyTree->Items->BeginUpdate();
    for(i=0;i<MyFtp->FTPDirectoryList->Attribute->Count;i++)
    {
        Temp = MyTree->Items->AddChild(
                Root, MyFtp->FTPDirectoryList->name->Strings[i]);
        if((MyFtp->FTPDirectoryList->Attribute->Strings[i])[1] ==
                                'd')
        {
            // Папка
            Temp->ImageIndex = 0;
            Temp->SelectedIndex = 0;
        }
        else
        {
            // Файл
            Temp->ImageIndex = 1;
            Temp->SelectedIndex = 1;
        }
    }
    MyTree->AlphaSort();
    MyTree->Items->EndUpdate();
    if(Root)
        Root->Expand(true);
    Screen->Cursor = Save_Cursor;
}
```

Сначала в этой функции сохраняются параметры текущего курсора — перед завершением выполнения функции их нужно будет восстановить. Затем устанавливается новая форма курсора — песочные часы (hourglass). Такая форма курсора общепринята, когда нужно сообщить пользователю, что приложение занято выполнением достаточно продолжительной операции. Далее локальному объекту типа TTreeNode присваивается указатель на выбранный элемент дерева MyTree. Если ни один элемент в древовидном списке не выбран, значение указателя будет NULL, что и нужно нам в такой ситуации. Далее вызывается метод List() объекта MyFtp. (Существует еще один аналогичный метод, Nlist(), который позволяет получить информацию только о наименованиях элементов, но для данного приложения он не подходит.)

Вызов метода BeginUpdate() объекта класса TTreeNodes (последний входит в состав MyTree) предотвращает обновление экрана до тех пор, пока не будет вызван метод EndUpdate(). Благодаря этому значительно повышается скорость вывода списка на экран. Затем организуется цикл просмотра списка Attributes объекта класса TFTPDirectoryList элемента FTPDirectoryList. Поскольку список Attributes и список имен в FTPDirectoryList соответствуют друг другу, тот же цикл можно использовать и для обработки типов и имен элементов. В теле цикла сначала создается новый объект класса TTreeNode, который становится потомком текущего (если указатель на текущий равен NULL, то им является корень дерева —root). Надпись на каждом из новых элементов будет соответствовать содержимому соответствующего элемента в списке имен. Затем проверяется первый символ в элементе списка Attributes; если это — символ d, то объект является папкой (или каталогом, если вам так больше нравится), и соответственно выбирается растр для его отображения на поле древовидного списка (устанавливается значение индекса 0); в противном случае устанавливается значение индекса 1, а значит, выводится изображение пиктограммы файла.

После завершения формирования в цикле списка элементов он сортируется по алфавиту, и вызывается метод EndUpdate(), чем инициируется обновление изображения на экране. Если перед обращением к функции в списке MyTree был выбран некоторый элемент, то его ветвь в древовидной структуре раскрывается. Последняя операция — восстановление формы курсора.

Обработка списка и загрузка файлов

На этом этапе мы добавим в программу функцию сортировки списка файлов по типам, а затем — каждого типа по алфавиту. После этого разработаем функцию, которая будет определять полный путь для каждого узла древовидной структуры. Последняя операция — включение в программу обработки двойного щелчка на пиктограмме выбранного файла и раскрытие соответствующей папки или загрузка соответствующего файла с сервера.

Сначала включим в программу операцию установки свойства SortType объекта MyTree — этому свойству нужно присвоить значение **stText**. В обработчик события OnCompare объекта MyTree добавьте следующий программный код:

```
void __fastcall TForm1::MyTreeCompare(TObject *Sender,
      TTreeNode *Node1, TTreeNode *Node2, int Data,
      int &Compare)
{
    if(Node1->ImageIndex > Node2->ImageIndex)
```

```
        Compare = 1;
    else
        if(Node1->ImageIndex == Node2->ImageIndex)
            Compare = CompareStr(Node1->Text, Node2->Text);
        else
            Compare = -1;
}
```

Сначала анализируются значения свойств `ImageIndex` сравниваемых узлов, содержащих информацию о типе узла, — папки или файла. Если значение `ImageIndex` узла `Node1` больше, то, значит, этот узел представляет в списке файл, и переменной `Compare` присваивается значение `1`. В противном случае проверяется, не равны ли значения индексов — в этом случае оба узла представляют либо файлы, либо папки. В случае равенства вызывается функция `CompareStr()`, которой в качестве аргумента передается текст ярлыка каждого узла. Если же индексы не равны, т.е. индекс узла `Node2` больше индекса узла `Node1`, а значит, узел `Node1` представляет папку, а узел `Node2` — файл, то переменной `Compare` присваивается значение `-1`.

После того как пользователь дважды щелкнет на пиктограмме какого-либо узла на поле списка `MyList`, должна быть выполнена одна из трех операций:

- если щелчок выполнен на пиктограмме файла, то должна начаться перегрузка выбранного файла;
- если щелчок выполнен на пиктограмме папки, содержимое которой еще не включено в список, то нужно загрузить в список перечень компонентов из этой папки;
- если щелчок выполнен на пиктограмме папки, содержимое которой уже включено в список, то соответствующая ветвь списка должна быть развернута.

Третий вариант процедуры выполняется в объекте класса `TTreeView` по умолчанию, и для ее реализации в программу не нужно добавлять специальный код. Два других варианта выполняются обработчиком события `OnDblClick` объекта `MyTree`, программный код которого приведен в листинге 12.7.

Листинг 12.7. Обработчик события `OnDblClick` объекта `MyTree`

```
void __fastcall TForm1::MyTreeDblClick(TObject *Sender)
{
    if(MyTree->Selected->ImageIndex == 0)
    {
        if(MyTree->Selected->Count == 0)
        {
            MyFtp->ChangeDir(GetPath());
            DoList();
        }
    }
    else
    {
        AnsiString RemoteFile;

        RemoteFile = GetPath();
        SaveDialog1->FileName = MyTree->Selected->Text;
```

```
        if(SaveDialog1->Execute())
            MyFtp->Download(RemoteFile, SaveDialog1->FileName);
    }
}
```

В этой программе сначала определяется тип узла в списке, на котором щелкнул пользователь. Если оказывается, что этот узел представляет папку, то возможны два варианта: если соответствующий узел имеет дочерние узлы в списке, значит, информация о содержимом данной папки уже загружена в список. В противном случае нужно извлечь эту информацию из сервера. Для анализа ситуации используется значение свойства `Count` выбранного элемента `Selected`. Если это значение равно `0`, то нужно загрузить информацию о содержимом папки, что выполняется методом `ChangeDir()` объекта `MyFtp`, которому в качестве аргумента передается результат выполнения функции `GetPath()`. Затем вызывается функция `DoList()`, которая добавляет новую информацию в список.

Если же выбранный элемент является файлом, то сначала в локальной переменной `RemoteFile` фиксируется результат выполнения метода `GetPath()`. Затем свойству `FileName` объекта `SaveDialog1` присваивается текст ярлыка выбранного файла (это — имя файла) и вызывается метод `Execute()` объекта `SaveDialog1`. Если работа с этим диалоговым окном завершается щелчком на кнопке Cancel, никакие действия не выполняются; в противном случае вызывается метод `Download()` объекта `MyFtp`. В качестве аргументов этому методу передаются компоненты полного имени файла — значение переменной `RemoteFile` и свойство `FileName` объекта диалогового окна `SaveDialog1`.

Метод `GetPath()` играет важную роль во всех операциях с каталогами и файлами на сервере. Поскольку дерево каталогов наследуется элементами объекта списка `MyList`, этот метод нужно использовать для определения пути к выбранным папкам или файлам. Добавьте в секцию общедоступных программных объектов файла `Unit1.h` следующий оператор объявления:

```
AnsiString __fastcall GetPath();
```

Далее в конец файла реализации `Unit1.cpp` добавьте тест метода `GetPath()`, приведенный в листинге 12.8.

Листинг 12.8. Метод `GetPath()`

```
AnsiString __fastcall TForm1::GetPath()
{
    TTreeNode *Base, *Temp;
    TStringList *TempList = new TStringList();
    int i;
    AnsiString ToReturn;

    Base = MyTree->Selected;
    TempList->Add(Base->Text);
    Temp = Base->Parent;
    while(Temp)
    {
        TempList->Add(Temp->Text);
        Temp = Temp->Parent;
```

```
    }
    for(i=TempList->Count-1;i>-1;i-)
    {
        ToReturn += "/" + TempList->Strings[i];
    }
    delete TempList;
    return ToReturn;
}
```

В этой функции сначала создается пара локальных переменных типа `TTreeNode` и новый список типа `TStringList`. В переменную `Base` дублируется указатель на выбранный узел типа `TTreeNode` объекта `MyList`. Затем текст ярлыка этого узла добавляется в список `TempList`, а в переменную `Temp` записывается указатель на родительский узел того узла, который зафиксирован в `Base`. Далее организуется цикл перехода по цепочке родительских узлов, который выполняется до тех пор, пока мы не выйдем на корневой узел дерева, т.е. узел, не имеющий родительского узла. При каждом переходе "вверх" по дереву в список `TempList` добавляется очередной ярлык. После завершения цикла просмотра дерева запомненные ярлыки считываются из списка `TempList` в обратном порядке, и формируется строка полного пути, в которой разделителем является косая черта. Перед завершением программы список `TempList` удаляется, а в вызывающую программу возвращается сформированная строка.

Завершение текущего сеанса и пересылка файлов

Для завершения текущего сеанса работы с FTP-сервером добавьте в обработчик события `OnClick` кнопки `StopButton` следующий программный код:

```
void __fastcall TForm1::StopButtonClick(TObject *Sender)
{
    MyFtp->Disconnect();
    StartButton->Enabled = true;
    StopButton->Enabled = false;
}
```

В этом фрагменте сначала вызывается метод `Disconnect()` объекта `MyFtp`, а затем разблокируется кнопка `StartButton` и блокируется кнопка `StopButton`. Таким образом, пользователь получает возможность подсоединиться к другому FTP-серверу и начать работу с ним.

Для того чтобы программа могла пересылать выбранный файл, добавьте в обработчик события `OnClick` кнопки `UploadButton` следующий программный код:

```
void __fastcall TForm1::UploadButtonClick(TObject *Sender)
{
    if(OpenDialog1->Execute())
    {
        MyFtp->Upload(OpenDialog1->FileName,
                 ExtractFileName(OpenDialog1->FileName));
    }
}
```

В этой простенькой функции вызывается метод `Execute` объекта диалогового окна `OpenDialog1`. Если работа с диалоговым окном завершена успешно (пользователь не щелкал на кнопке Cancel), вызывается метод `Upload()` объекта `MyFtp`. В качестве аргументов этому методу передаются полное имя файла (включая и путь) и имя файла без пути.

Теперь работа над проектом завершена, и можно сохранить все файлы приложения.

На этом мы завершаем рассмотрение протоколов обмена информацией и методику их использования в приложениях.

Резюме

В этой главе вы научились расширять возможности создаваемых приложений и обеспечивать обмен информацией по сети. Вы имели возможность на реальных примерах изучить методику программирования различных сетевых протоколов. Теперь в вашем арсенале появились инструменты, которые позволят создавать эффективные приложения.

Глава

13 Программирование Web-сервера

Боб Сворт
Вильям Моррисон

В этой главе будут рассмотрены вопросы программирования Web-серверов с помощью C++Builder. Эта среда поддерживает технологию ***WebBroker***, в частности методы CGI/WinCGI и ISAPI/NSAPI, а также функциональные возможности ***InternetExpress***.

Для создания приложений серверов Web можно использовать средства технологии ***WebBroker***, входящие в состав среды разработки C++Builder 5. В состав этих средств входят специальный модуль данных, названный ***Web Module***, два мастера и множество отдельных компонентов. Наибольшую популярность среди разработчиков завоевал мастер ***Web Server Application Wizard***. Другой мастер из состава C++Builder — ***Database Web Application Wizard*** — позволяет помимо прочего помещать в ***Web Module*** еще и таблицу.

Компоненты, которые используются при создании приложений Web-сервера, можно разделить на две группы — поддерживающие компоненты и классы, такие как `TWebModule`, `TWebDispatcher`, `TWebRequest` и `TWebResponse`, и производящие компоненты — `TPageProducer`, `TDataSetPageProducer`, `TDataSetTableProducer` и `TQueryTableProducer`.

Мастера ***WebBroker*** и перечисленные компоненты входят в комплект редакции C++Builder 5 Professional, а в комплект редакции C++Builder 5 Enterprise дополнительно включены компоненты `TMidasPageProducer` и `TReconcilePageProducer`, известные также как средства поддержки функций ***InternetExpress***.

На заметку

Файлы приложений, описанных в этой главе, находятся на прилагаемом к книге компакт-диске. Кроме них нужно подготовить самостоятельно несколько дополнительных файлов. Соответствующая инструкция находится в файле `README.TXT` в папке `WebServer`.

Web Module

Термины *WebBroker* и *Web Module* часто обозначают одно и то же. ***WebBroker*** можно рассматривать как часть ***Web Module*** (диспетчер операций — ***Action Dispatcher***), которая переносит модули данных в ***Web Module***. С другой стороны, ***WebBroker*** можно считать программным ядром, которое позволяет создавать Web-серверные приложения на базе протоколов ISAPI/NSAPI, CGI или WinCGI несмотря на различия, имеющиеся в этих протоколах. Кроме того, ***Web Bridge*** позволяет разработчику использовать один и тот же пакет API как для работы с Microsoft ISAPI (всех версий), так и для работы с Netscape NSAPI (вплоть до версии 3.6). При этом разработчик может игнорировать различия этих пакетов API. Более того, поскольку Web-серверные приложения являются невизуальными (т.е. они работают на Web-сервере, а пользовательский интерфейс обеспечивается клиентским приложением, которое использует Web-броузер), то мастера и компоненты ***Web Module*** обеспечивают разработчику необходимые средства поддержки в процессе разработки, что значительно предпочтительнее разработки невизуального программного кода с помощью C++Builder.

Мастер *Web Server Application Wizard*

Мастер ***Web Server Application Wizard*** можно отыскать на первой вкладке New хранилища ***Repository***. Для вызова диалогового окна хранилища на экран выберите команду File⇨New. После запуска мастера ***Web Server Application Wizard*** нужно выбрать тип приложения, которое планируется создать: ISAPI/NSAPI (предлагается по умолчанию), CGI или WinCGI (рис. 13.1).

*Рис. 13.1. Окно мастера **Web Server Application Wizard** позволяет выбрать тип приложения*

Приложение типа CGI

Web-серверное приложение, использующее *стандартный интерфейс шлюза* (CGI — Common Gateway Interface), — это консольное приложение, которое загружается Web-сервером в ответ на каждый запрос и выгружается после выполнения запроса. Исходная информация от клиента поступает через стандартный канал ввода, а результат— обычно в формате HTML — отсылается на стандартный выходной канал.

Приложение типа WinCGI

WinCGI — это специальный вариант протокола CGI, который используется для работы с операционной системой Windows. Этим протоколом, вместо стандартных входного и выходного каналов, для обмена информацией предусматривается использование назначений в INI-файле. С точки зрения программиста, единственное существенное отличие WinCGI-приложения типа от стандартного (консольного) CGI-приложения состоит в том, что первое является *GUI*-приложением, хотя оно также является невизуальным. В этой главе для демонстрации возможностей работы CGI/WinCGI-приложений я буду использовать только стандартный протокол CGI.

Приложение типа ISAPI/NSAPI

Модули DLL расширения Web-серверов ISAPI (Microsoft IIS) или NSAPI (Netscape) во многом похожи на WinCGI/CGI-приложения, но между ними есть и существенное различие, которое состоит в том, что использованный DLL-модуль динамической библиотеки остается загруженным и после того, как запрос к серверу выполнен. Поэтому последующие запросы, которые требуют использования того же модуля, будут выполняться быстрее.

В будущем Netscape обещает поддерживать и протокол ISAPI, и поскольку вы можете воспользоваться "транслятором" модулей DLL, который преобразует вызовы функций протокола ISAPI в обращение к соответствующим функциям протокола NSAPI (если придется использовать более старые Netscape Web-серверы), то в дальнейшем при освещении вопросов, касающихся как ISAPI, так и NSAPI, я буду использовать только термин *ISAPI*.

Сравнение CGI с ISAPI

Программный код, который мастер включает в приложения типов CGI и WinCGI, практически идентичен. Хотя DLL для работы с ISAPI/NSAPI имеют несколько отличные главные секции, сам по себе ***Web Module*** одинаков для всех трех вариантов. Таким образом, если вы собираетесь разработать приложение типа CGI, но желаете сначала протестировать его как модуль DLL, работающий с ISAPI, поскольку такие DLL можно проверять не покидая среды C++Builder, то нужно создавать два проекта (один — типа CGI, а другой — типа ISAPI), причем оба проекта будут использовать один и тот же модуль ***WebBroker***. Для этого создаются два проекта (в составе одной и той же группы проектов), удаляется ***Web Module*** из выполняемого CGI и добавляется (как разделяемый) ***Web Module*** из ISAPI DLL. Для самого ***Web Module*** не имеет никакого значения, будет ли он компилироваться как выполняемый или как динамически подгружаемый, но разработчику значительно удобней отлаживать второй вариант.

Основной недостаток приложений типа CGI — задержка реакции на запрос, поскольку это приложение приходится повторно загружать в ответ на каждый очередной запрос. Но динамически подгружаемые модули для работы с ISAPI труднее обновлять — для этого требуется выключать Web-сервер. Они менее устойчивы, причем "норовистый" модуль DLL потенциально способен вывести из строя и весь Web-сервер. С чисто программистской точки зрения, использование постоянно загруженного в память модуля DLL не гарантирует, что все

нужные переменные будут очищены после выполнения запроса и подготовлены к новому. Поэтому в процессе разработки придется тщательно проверить установку в исходное состояние всех переменных перед началом выполнения очередного запроса. Прежде чем устанавливать на сервере такой DLL-модуль, нужно удостовериться в его правильной работе на все 100 %.

Поддерживающие компоненты *WebBroker*

После того как в диалоговом окне New Web Server Application выбран тип создаваемого приложения (выберите вариант CGI или "закажите" сразу два варианта — и CGI, и ISAPI, о чем было сказано выше), C++Builder сформирует новый ***WebBroker***-проект и пустой модуль ***Web Module***. Сохраните этот проект под именем `WebShow`, а "новорожденному" модулю дайте имя `WebMod` — мы будем использовать его во всех дальнейших упражнениях в этой главе.

Прежде чем мы продолжим работу с этим модулем, откройте диалоговое окно Project Options, выберите в нем вкладку Linker и отключите опцию Use Dynamic RTL. После этого перейдите на вкладку Packages и отключите на ней опцию Build with Runtime Packages. Если не отключить эти опции, то придется распространять пакет вместе с созданным приложением ***WebBroker***. Все Web-серверные приложения, которые мы будем рассматривать в этой главе, строятся аналогично — без динамических библиотек RTL и пакетов времени выполнения.

В ***Web Module*** в процессе разработки приложения помещаются специальные компоненты ***WebBroker***, такие как `PageProducers` и `TableProducers`. Эти компоненты находятся на вкладке Internet палитры компонентов C++Builder 5, как показано на рис. 13.2. Компоненты, специфичные для ***InternetExpress***, находятся на вкладке InternetExpress, но о них мы расскажем ниже, в разделе *InternetExpress*.

На вкладке Internet вы увидите пиктограммы следующих компонентов (в порядке слева направо): `TClientSocket` и `TServerSocket` (эти два компонента не относятся к компонентам ***WebBroker***, но размещены на этой же вкладке), `TCppWebBroker` (он используется для отображения сформированных Web-страниц), `TWebDispatcher`, `TPageProducer`, `TQueryTableProducer`, `TDataSetTableProducer` и `TDataSetPageProducer`. ActiveX-компонент `TCppWebBroker` используется для реализации приложения хоста ***IntraBob***, с помощью которого можно отлаживать приложения ***Web Module*** не покидая среды разработки C++Builder (более подробно об этом будет рассказано позже в этой главе).

Помимо компонентов, представленных на палитре компонентов C++Builder 5, в этой главе мы рассмотрим и компоненты поддержки ***Web Module***, в частности `TWebModule`, `TWebRequest` и `TWebResponse`.

Компонент `TWebDispatcher`

Компонент `TWebDispatcher` встроен в ***Web Module*** и его можно использовать для преобразования существующих модулей данных в ***Web Module*** (можно считать, что `TDataModule` + `TWebDispatcher` = `TWebModule`). В каждом приложении может существовать только один объект класса `TWebDispatcher`, а это означает, что приложение может включать в свой состав не более одного экземпляра ***Web Module*** или модуля данных с объектом компонента `TWebDispatcher`. Обращаю ваше внимание на то, что ***Web Module*** можно сформировать, только обратившись к "услугам" мастера ***Web Server Application Wizard***, который выполняет все операции, необходимые для создания нового проекта. В свете этого замечания весьма прискорбно, что C++Builder 5 не предупреждает разработчика о выполнении недопустимой операции, если тот случайно "сбросит" в модуль данных больше одного объекта `TWebDispatcher` (если пользователь попытается установить его принудительно в ***Web Module***, такое предупреждение последует).

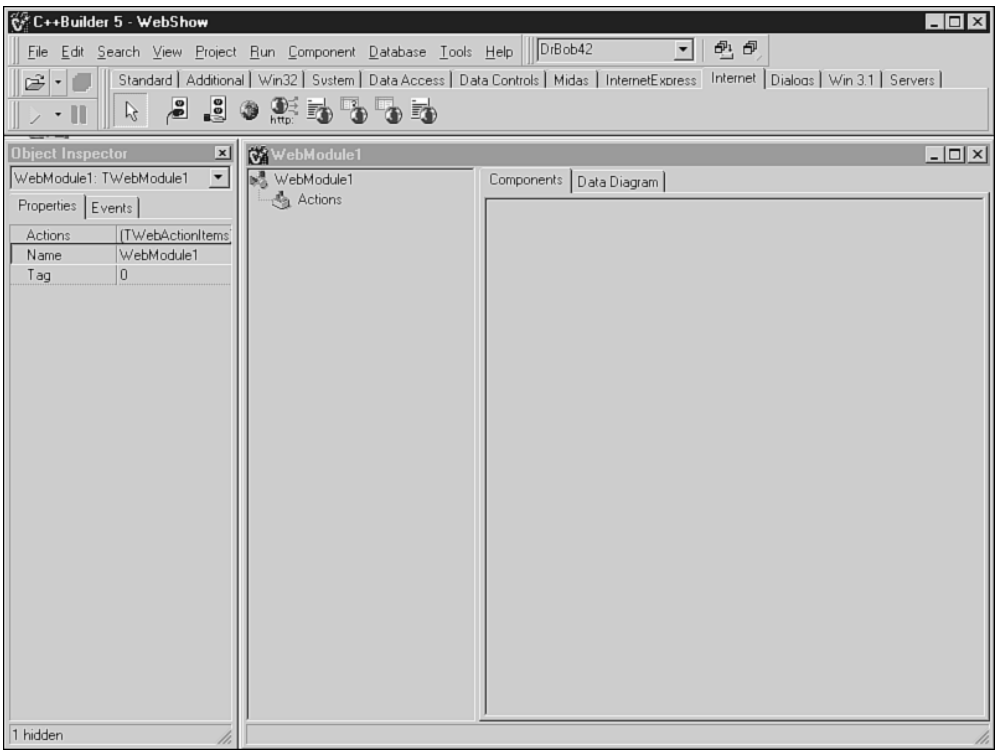


Рис. 13.2. Вкладка Internet палитры компонентов C++Builder 5

Компонент `TWebDispatcher` берет на себя заботы о распределении поступающих запросов между элементами Web, которые выполняют обработку этих запросов.

Компонент `TWebModule`

Web Module можно рассматривать как модуль данных, ориентированный на работу в сети Web. Наиболее важным свойством компонента `TWebModule` является свойство `Actions`, которое имеет тип `TWebActionItems`. Запустить Action Editor для работы с объектами класса `TWebActionItems` можно разными способами. Первый способ предполагает использование диалогового окна Object Inspector: перейдите в это окно и щелкните на многоточии рядом с (`TWebActionItems`). Другой способ: щелкните правой кнопкой мыши на Web Module и выберите в контекстном меню пункт Action Editor, а затем укажите различные типы запросов, на которые должен реагировать ***Web Module***. Есть и еще один способ, который появился только в версии C++Builder 5. Этот способ предполагает использование программы ***Visual Data Module Designer***, которая отображает все элементы `Action` в виде дочерних элементов ***Web Module***. Пользователь может щелкнуть правой кнопкой мыши на Actions и добавить в этот список новый элемент.

Элементы свойства `Actions` эквивалентны запросам клиентов к серверу, и разработчик приложения может специфицировать их свойства и события. Web-серверное приложение может реагировать на каждый элемент `Actions` по-разному — в зависимости от того, как реализован соответствующий обработчик запроса. Отличить одни объекты `TWebActionItems` от других можно по значению свойства `PathInfo` — это строка, которая следует сразу вслед за URL (но перед значением `querystring`).

Используя программу ***Actions Editor*** (ее диалоговое окно показано на рис. 13.3), разработчик может сформировать множество экземпляров `TWebActionItem`, причем каждый из них отличается от прочих значением свойства `PathInfo`. Свойство `PathInfo` содержит дополнительную информацию, которая присоединяется к запросу непосредственно перед полем `Query`. Это означает, что одно и то же Web-серверное приложение может реагировать на различные HTTP-запросы, причем каждый из них обрабатывается соответствующим объектом `TWebActionItem`.

В тех примерах, которые мы будем рассматривать в этой главе, вам придется определить девять объектов `TWebActionItem`, которые помогут продемонстрировать методы использования и возможности компонентов ***Web Module***.

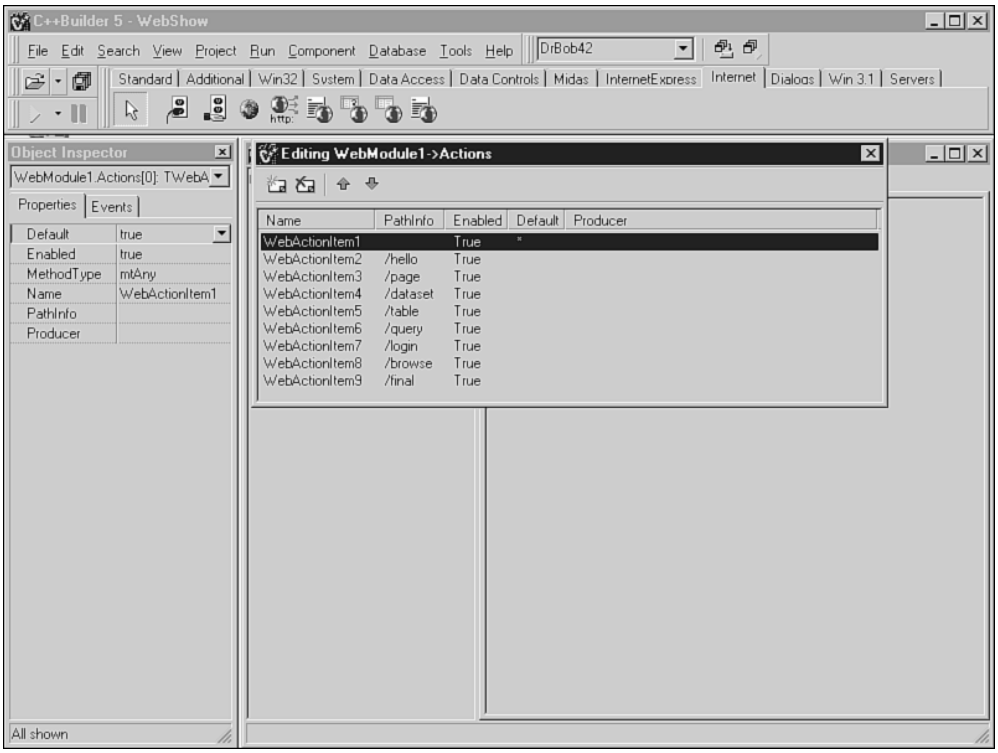


Рис. 13.3. Диалоговое окно программы **Actions Editor**

Обратите внимание на то, что в первом объекте значение свойства `PathInfo` не задано — это объект, выполняющий обработку по умолчанию. Этот объект `TWebActionItem` будет выбран в том случае, если свойство `PathInfo` в запросе не задано или если значение этого свойства не совпадает ни с одним из значений в других объектах `TWebActionItem`. В восьми других объектах свойствам `PathInfo` присвоены следующие значения: **`/hello`**, **`/page`**, **`/dataset`**, **`/table`**, **`/query`**, **`/login`**, **`/browse`** и **`/final`**. Эти объекты будут использоваться во множестве примеров в этой главе.

Для того чтобы приступить к разработке обработчика события для первого объекта `TWebActionItem` (объекта для выполнения обработки по умолчанию), выберите строку с `WebActionItem1` в окне программы ***Actions Editor*** (см. рис. 13.3), и активизируйте вкладку Events диалогового окна Object Inspector. Затем дважды щелкните на строке события `OnAction`. В результате откроется окно редактора программного кода, в котором вы увидите следующий текст, подготовленный для обработчика события средой разработки:

```
void __fastcall TWebModule1::WebModule1WebActionItem1Action(
      TObject *Sender, TWebRequest *Request,
      TWebResponse *Response, bool &Handled)
{

}
```

Прежде чем внести в него операторы, выполняющие собственно обработку события, давайте рассмотрим внутреннюю структуру классов `TWebRequest` и `TWebResponse`, поскольку объекты этих классов передаются обработчику в качестве аргументов при вызове.

Класс TWebResponse

Параметр `Response`, тип которого — `TWebResponse` — имеет множество свойств, наиболее важным из них является `Content`. Это строка, в которую можно поместить любой HTML-код, предназначенный для возвращения клиенту в ответ на запрос. Приведенный ниже программный код сформирует ответное сообщение **`Hello, world!`**:

```
void __fastcall TWebModule1::WebModule1WebActionItem1Action(
      TObject *Sender, TWebRequest *Request,
      TWebResponse *Response, bool &Handled)
{
  Response->Content = "<H1>Hello, world!</H1>";
}
```

Конечно, разработчик может поместить в свойство `Response->Content` все, что посчитает нужным в контексте разрабатываемого приложения. Как правило, содержимое этого свойства имеет тип `text/html` (таково по умолчанию значение свойства `Response->ContentType`). Если же вы пожелаете вернуть клиенту данные другого типа, нужно соответственно скорректировать и значение свойства `Response->ContentType`. Двоичные данные (например, изображение) нельзя возвращать непосредственно через `Response->Content`. Для этого следует использовать другое свойство — `Response->ContentStream`.

Среди прочих свойств объекта класса, представляющих интерес для разработчика, выделим `ContentEncoding`, `Cookies`, `StatusCode`, `ReasonString`, `WWWAuthenticate`, `Realm`, `Date`, `Expires` и `LogMessage`. Свойство `Realm` имеет отношение к безопасности, и о нем более подробно будет рассказано в разделе *Безопасность в сети Web* далее в этой же главе. Свойство `Expires` позволяет манипулировать "временем жизни" динамически формируемых документов. С помощью свойства `LogMessage` можно поместить определенное сообщение в файл протокола работы Web-сервера.

Класс TWebRequest

Параметр `Request` представляет собой объект класса `TWebRequest` и содержит свойства и методы, связанные с вводом запроса. В зависимости от того, какой метод использовался для отсылки запроса (`GET` или `POST`), поступившую информацию можно отыскать в свойствах `Query` и `QueryFields` или в `Content` и `ContentFields`. Программно это можно определить следующим образом:

```
void __fastcall TWebModule1::WebModule1WebActionItem1Action(
      TObject *Sender, TWebRequest *Request,
```

```
      TWebResponse *Response, bool &Handled)
{
  Response->Content = "<H1>Hello, world!</H1>";
  if (Request->Method == "GET")
    Response->Content = Response->Content + "<B>GET</B>" +
      "<BR>Query: " + Request->Query;
  else
    if (Request->Method == "POST")
      Response->Content = Response->Content + "<B>POST</B>" +
        "<BR>Content: " + Request->Content;
}
```

Среди прочих свойств класса `TWebRequest` обращаю ваше внимание на свойства `CookieFields`, `Authorization`, `Referrer`, `UserAgent` и `Method`. Свойство `Authorization` служит для определения авторства поступившего запроса, и о нем подробно будет рассказано разделе *Безопасность в сети Web* далее в этой же главе. Свойство `Method` позволяет определить, какой метод протокола был использован для передачи запроса — `GET` или `POST`.

Между методами протокола `GET` и `POST` существует довольно важное отличие. Когда используется протокол `GET`, поля запроса передаются на URL. Это выполняется довольно быстро, но метод ограничивает объем передаваемых данных — не более нескольких килобайт, хотя для большинства запросов этого вполне достаточно. Протокол `POST` предусматривает передачу данных с использованием стандартных технологий ввода/вывода (или INI-файлов Windows в случае WinCGI). Это медленнее, но объем передаваемых данных ограничивается только наличием свободного пространства на диске. Кроме того, при использовании протокола `POST` вы не сможете увидеть переданные данные непосредственно на URL, а потому они не могут быть случайно искажены.

Лично я предпочитаю пользоваться протоколом `POST`, поскольку он практически не ограничивает объем передаваемых данных. Протокол `GET` я использую только в том случае, когда на то есть достаточно веские причины.

Web-серверы

Для тестирования приложений ***Web Module*** нам потребуется Web-сервер, который может быть либо персональным (если вы работаете в операционной среде Windows 95/98), либо ***Microsoft Internet Information Server*** (IIS), если вы работаете под управлением Windows NT или Windows 2000. В главе 30 руководства *Borland C++Builder 5 Developer's Guide* содержатся достаточно четкие инструкции о том, как организовать персональный Web-сервер для тестирования и отладки ISAPI DLL или CGI-приложений. Это не так просто, если вы пользуетесь IIS версии 4 или более поздней (этот продукт входит в комплект NT 4 Option Pack и Windows 2000).

Обращаю ваше внимание также на то, что реальный Web-сервер не выгружает модули ISAPI DLL, после того как они загружены в первый раз. Это означает, что, прежде чем повторно скомпилировать разрабатываемый модуль ISAPI DLL, потребуется вручную остановить работу Web-сервера, а также сервис администрирования IIS. Затем придется повторно запустить Web-сервер, и только после этого будет загружена отредактированная версия отлаживаемого модуля ISAPI DLL. Кроме того, неотлаженный модуль DLL может стать причиной полного краха сервера, и его придется повторно запускать. Последние вер-

сии Internet Information Server располагают опцией Run in Separate Memory Space, которая помогает справиться с этой проблемой. Можно так настроить IIS, что модули ISAPI DLL будут выгружаться после выполнения запроса. Для этого в диалоговом окне Internet Service Manager нужно сбросить флажок Cache ISAPI Applications для соответствующего Web-сайта. Это очень удобно, поскольку избавляет на этапе отладки от необходимости останавливать и повторно запускать работу Web-сервера. Но если оставить такую настройку и после завершения отладки, преимущество в скорости, которое дает применение ISAPI DLL, будет утеряно.

В качестве альтернативы "реальному" Web-серверу можно воспользоваться свободно распространяемой программой ***IntraBob*** версии 5.0, диалоговое окно настройки которой показано на рис. 13.4. Лично я использую ее в качестве главного средства отладки ISAPI-приложений, создаваемых в средах Delphi и C++Builder. Эта программа позволяет на одном компьютере, без всякого подключения к сети разрабатывать и отлаживать модули ISAPI DLL. Установка этой программы выполняется очень просто. Перепишите файл `IntraBob.exe` в тот же каталог, в котором находится исходный код ISAPI-проекта и укажите расположение ***IntraBob*** в диалоговом окне Run⇨Run Parameters как `Host Application`. Выполняемый файл программы вы найдете на компакт-диске, прилагаемом к этой книге или на моем Web-сервере по адресу `http://www.drbob42.com`.

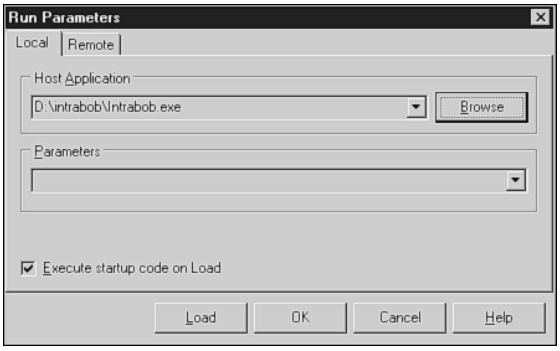


Рис. 13.4. Диалоговое окно настройки **IntraBob** *для использования в качестве средства отладки приложений* **Web Module**

При запуске модуля ISAPI DLL автоматически запускается и приложение ***IntraBob***, отображающее Web-страницу, с которой можно управлять отлаживаемым Web-серверным приложением. При этом вы сможете использовать и встроенные средства отладки среды C++Builder для установки точек останова в исходном тексте программы. Например, задайте точку останова на первом операторе, который анализирует значение `Request->Method`, и вы увидите, что C++Builder остановит здесь выполнение программы, как только вы запустите в режиме отладки приложение типа ***Web Module***.

Для запуска модуля ISAPI DLL нужно подготовить специальную Web-страницу, в которой должна присутствовать HTML-форма, загружающая приложение ***Web Module***. Если текущий файл проекта называется `WebShow.bpr`, то свойству `ACTION` нужно присвоить значение `http://localhost/scripts/WebShow.dll`. Ниже приведен соответствующий HTML-код.

```
<HTML>
<BODY>
```

```
<H1>WebBroker HTML Form</H1>
<HR>
<FORM ACTION="http://localhost/scripts/WebShow.dll" METHOD=POST>
Name: <INPUT TYPE=EDIT NAME=Name>
<P>
<INPUT TYPE=SUBMIT VALUE=Submit>
</FORM>
</BODY>
</HTML>
```

После загрузки Web-страницы в ***IntraBob*** на экране появится ее HTML-форма — ***WebBroker***. Нужно заполнить реквизиты в этой форме, щелкнуть на кнопке Submit — в ответ будет загружено и начнет выполняться приложение `WebShow`.

Программа ***IntraBob*** раскрывает HTML-форму и автоматически заполняет поля вкладки Options значениями, соответствующими настройке на удаленное приложение типа CGI (а в данном случае ISAPI), как показано на рис. 13.5. Обращаю ваше внимание на то, что на любой стадии отладки приложения можно переключиться на эту вкладку и вручную настроить опции (это самый постой способ изменить значение `PathInfo` запроса и таким образом активизировать другой объект `WebActionItem`).

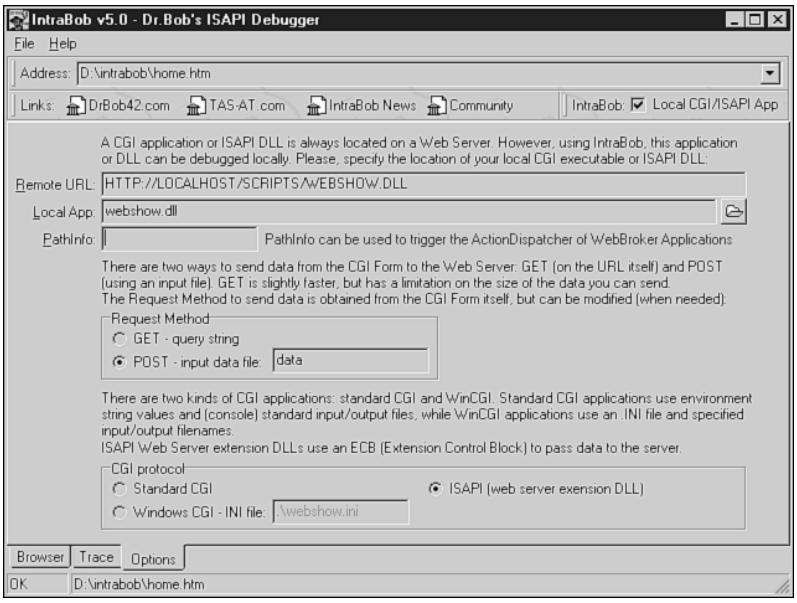


Рис. 13.5. Вкладка Options программы **IntraBob**

Помните, чуть выше мы с помощью отладчика C++Builder установили точку останова на первом операторе метода `WebModule1WebActionItem1Action`, в котором проверялось значение `Request->Method`? Теперь, как только вы щелкните на кнопке Submit, активизируется объект `WebActionItem` по умолчанию и программа остановится на заданной точке останова (рис. 13.6). Пользуясь опциями отладчика, например ToolTip Expression Evaluation (Вычисление выражения в окне контекстной подсказки), можно прочесть на экране значение `Request->Method` или `Request->Content`.

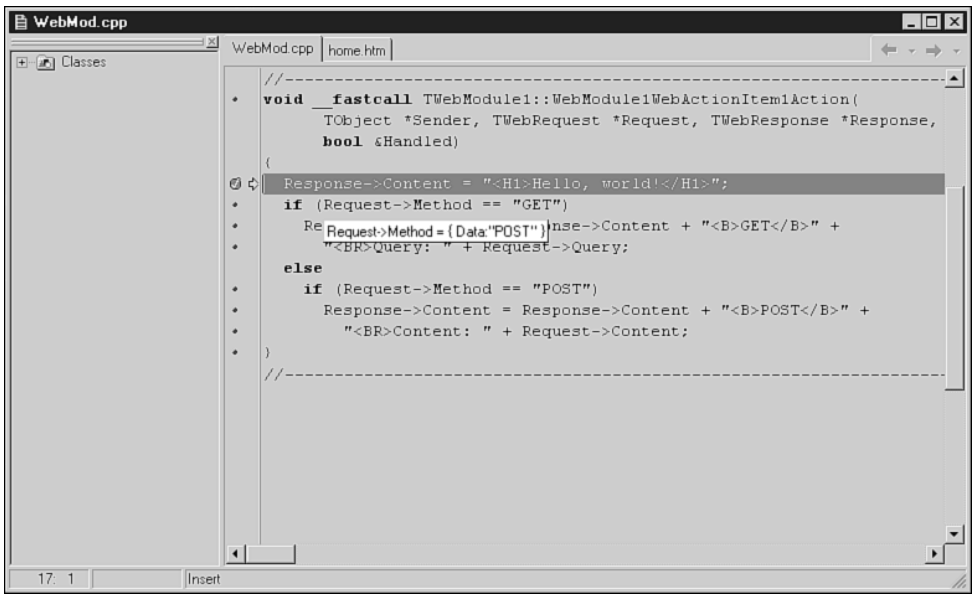


Рис. 13.6. Фрагмент программного кода **Web Module** *в окне отладчика C++Builder*

При запуске программы будет сформировано исключение "Delphi" (причем, неоднократно), и на экране появится следующее сообщение:

```
Project Intrabob.exe raise  exception class Exception with message
↳"Only one data module per application". Process Stopped.
↳Use Step or Run to continue.

Проект Intrabob.exe сгенерировал исключение класса Exception со
следующим сообщением
↳"Только один модуль данных на приложение". Процесс остановлен.
↳Воспользуйтесь командами Step или Run для продолжения работы.
```

Это исключение возникает вследствие невыявленной пока что ошибки в способе загрузки и выгрузки модулей ISAPI DLL, который используется в C++Builder. Нужно вручную внести исправление в исходный текст программы WebShow, который содержит некорректный вызов `CreateForm` при запуске `DllEntryPoint` (эта процедура запускается, если `reason` имеет одно из следующих значений: `Process Attach`, `Thread Attach`, `Process Detach` и `Thread Detach`). Вызывать `CreateForm` нужно только в том случае, когда значение `reason` — **`PROCESS ATTACH`**. Я рекомендую внести для этого в программу единственный условный оператор:

```
if (reason == DLL_PROCESS_ATTACH)
        Application->CreateForm(__classid(TWebModule1),
                                          &WebModule1);
```

Это исправление помогает легко справиться с проблемой.

После того как вы нажмете клавишу <F9>, в окне программы ***IntraBob*** появится результат работы приложения (рис. 13.7).

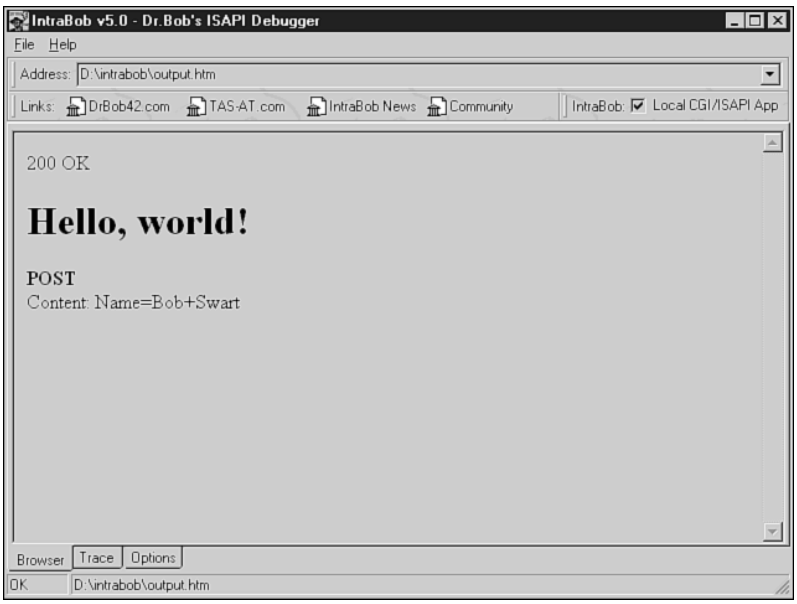


Рис. 13.7. Результат работы **Web Module** *в окне программы* **IntraBob**

Прекрасно! Только что вы разработали HTML-форму, которая возвращает имена и значения всех полей ввода. Эту форму очень удобно использовать в тех случаях, когда создается впечатление, что объект `WebActionItem` работает не так, как задумано, и нужно проверить, получил ли он исходные данные в нужном порядке. Обратите внимание на то, что пробелы в тексте сообщения заменены на знаки "плюс" (+). Кроме того, все специальные символы заменяются символом процента (%), за которым следует шестнадцатеричный код замененного символа.

Производящие компоненты *WebBroker*

Четыре производящих Internet-компонента ***WebBroker*** — `TPageProducer`, `TDataSetPageProducer`, `TDataSetTableProducer` и `TQueryTableProducer` — размещены на вкладке Internet палитры компонентов C++Builder.

Компонент `TPageProducer`

В строковую переменную `Response->Content` можно поместить любое значение, вплоть до полного текста Web-страницы. Однако иногда возникает необходимость вернуть HTML-строки, созданные по определенному шаблону, в котором нужно заполнить только некоторые реквизиты (например, имена или даты, или какие-то поля таблицы). В этом случае имеет смысл воспользоваться компонентом `TPageProducer`.

Компонент `TPageProducer` располагает двумя свойствами с помощью которых можно специфицировать заданные элементы формируемой HTML-страницы. Свойство `HTMLFile` хранит указатель на внешний HTML-файл. Пользоваться этим свойством рекомендуется в тех случаях, когда нужно изменять шаблон Web-страницы, не компилируя повторно все приложение (при этом шаблоном может заниматься кто-либо другой, а у вас будет возможность сосредоточиться на разработке приложения). Другое свойство, `HTMLDoc`, имеет тип `TStrings` и предназначено для

хранения HTML-текста, жестко зафиксированного в DFM-файле. Это свойство удобно использовать при подготовке демонстраций или статей по технологии ***WebBroker***, хотя такой подход не дает той гибкости, которую можно получить, пользуясь свойством `HTMLFile`.

Заданные элементы содержимого компонента `TPageProducer` могут содержать произвольный HTML-код, а также специальные теги, помеченные значком `#`. Эти теги считаются "неправильными" и Web-броузеры их игнорируют, но они генерируют событие `OnHTMLTag` компонента `TPageProducer` точно так же, как и "правильные". При обработке события можно заменять `TagString` на `ReplaceText`. Специальные теги могут также содержать параметры, которые следуют сразу за именем тега (например, параметр `Format=YY/MM/DD` задает формат вывода даты). Более подробную информацию вы можете отыскать в системе электронной справки по ключевому слову `TagParams`.

Давайте в качестве примера использования специальных тегов введем в свойство `HTMLDoc` следующий текст:

```
<H1>TPageProducer</H1>
<HR>
<#Greeting> <#Name>,
<P>
It's now <#Time> and we're working with a PageProducer.
```

Здесь использовано три специальных тега (их имена начинаются с символа #). Каждый из них вызовет генерацию события `OnHTMLTag` компонента `TPageProducer`. В листинге 13.1 приведен текст обработчика этого события, в котором выполняется подстановка осмысленного текста вместо таких тегов.

Листинг 13.1. Обработчик события `OnHTMLTag` компонента `TPageProducer`

```
void __fastcall TWebModule1::PageProducer1HTMLTag(
      TObject *Sender, TTag Tag, const AnsiString TagString,
      TStrings *TagParams, AnsiString &ReplaceText)
{
  if (TagString == "Name")
    ReplaceText = "Bob"; // имя, жестко вписанное в программу
  else
  if (TagString == "Time")
    ReplaceText = DateTimeToStr(Now());
  else // TagString == "Greeting"
    if ((double)Time() < 0.5)
      ReplaceText = "Good Morning";
    else
      if ((double)Time() > 0.7)
        ReplaceText = "Good Evening";
      else
        ReplaceText = "Good Afternoon";
}
```

Обратите внимание на то, что в этой программе использовано явное приведение типа (`double`) при обращении к функции `Time()`. В противном случае появится сообщение компилятора о подозрительном несоответствии между типами `double` и `int` в операции сравнения (функция `TDateTime()` возвращает результат типа `int`).

Использование ReplaceText в сочетании с фиксированным именем Bob выглядит несколько странным, поскольку в HTML-форме пользователю специально предлагается ввести свое имя. Не можем ли мы вместо жестко установленного имени использовать это значение (воспользовавшись, например, QueryFields или ContentFields)? Да, конечно же, можем, поскольку имеется непосредственный доступ к свойству Request компонента TWebModule. То же самое справедливо и в отношении свойства Response.

В листинге 13.2 представлен модифицированный вариант обработчика события OnHTMLTag, в котором учтены эти соображения.

Листинг 13.2. Второй вариант обработчика события OnHTMLTag компонента TPageProducer

```
void __fastcall TWebModule1::PageProducer1HTMLTag(
      TObject *Sender, TTag Tag, const AnsiString TagString,
      TStrings *TagParams, AnsiString &ReplaceText)
{
  if (TagString == "Name")
  {
    if (Request->Method == "POST")
      ReplaceText = Request->ContentFields->Values["Name"];
    else
      ReplaceText = Request->QueryFields->Values["Name"];
  }
  else
  if (TagString == "Time")
    ReplaceText = DateTimeToStr(Now());
  else // TagString == "Greeting"
    if ((double)Time() < 0.5)
      ReplaceText = "Good Morning";
    else
      if ((double)Time() > 0.7)
        ReplaceText = "Good Evening";
      else
        ReplaceText = "Good Afternoon";
}
```

Эта программа будет последней, в которой явно проверяется Request->Method для того, чтобы определить, какое свойство — QueryFields (GET) или ContentFields (POST) — использовать для считывания введенного значения. В дальнейшем мы всегда будем ориентироваться на метод POST (но не забывайте, что в своих программах вы можете при желании использовать и GET, и POST).

Прежде чем приступить к окончательной проверке программы, нам придется подключить объект Action (а именно /hello) к генератору страницы. Это можно сделать двумя способами. Новинкой версии C++Builder 5 является свойство Producer в элементах Action, которым можно воспользоваться для выбора PageProducer1 в качестве генератора для WebActionItem2. Преимущество такого способа состоит в том, что он фактически не требует дополнительного программного кода, а недостаток — нельзя задать точку останова в процессе отладки, поскольку отсутствует соответствующий оператор в тексте программе. Второй способ — разработать обработчик события OnAction для элемента /hello:

```
void __fastcall TWebModule1::WebModule1WebActionItem2Action(
      TObject *Sender, TWebRequest *Request,
```

```
      TWebResponse *Response, bool &Handled)
{
  Response->Content = PageProducer1->Content();
}
```

Можно использовать оба способа в одной и той же программе, но нужно помнить, что сначала выполняется обработка свойства `Producer`, а затем возбуждается событие `OnAction` и, соответственно, вызывается обработчик этого события.

Для того чтобы активизировать определенный объект `WebActionItem`, нужно обязательно передать в `PathInfo` вид действия `/hello`. Это можно сделать либо с помощью поля PathInfo в диалоговом окне программы ***IntraBob*** (см. рис. 13.5), либо, включив строку `PathInfo` в значение `ACTION` на начальной Web-странице `home.htm`:

```
<FORM ACTION="http://localhost/scripts/WebShow.dll/hello"
↳METHOD=POST>
```

Если HTML-форма заполняется с помощью ***IntraBob***, введите в поле имени значение **Bob Swart** и щелкните на кнопке Submit. Результат должен выглядеть так, как показано на рис. 13.8.

Рис. 13.8. Результат работы компонента `TPageProducer`

И еще одно замечание касательно компонента `PageProducer`, которое я делаю специально для пользователей FrontPage. Некоторые редакторы Web-страниц допускают включение в текст параметров тегов в двойных кавычках. При работе с таким редактором тег `<#Date Format>`, который используется в наших примерах, превратится в `<#Date "Format">`. Свойство `StripParamQuotes` (по умолчанию ему присвоено значение **`true`**) позволяет убрать эти нежелательные кавычки из специальных #-тегов. (Если ваш редактор не создает вам проблем с двойными кавычками, присвойте свойству `StripParamQuotes` значение **`false`**, но не забывайте о внешних шаблонах, на которые указывает свойство `HTMLFile`. Если окажется, что для их подготовки использовался другой редактор, то внесенные им двойные кавычки могут послужить причиной ненадежной работы разработанного приложения ***WebBroker***.)

Компонент TDataSetPageProducer

Компонент TDataSetPageProducer является производным от TPageProducer. В нем имеется новое свойство DataSet, и компонент сопоставляет имя специального тега, начинающееся с символа #, с именем поля в свойстве DataSet. Если будет обнаружено соответствие, компонент TDataSetPageProducer подставит значение из этого поля вместо специального тега.

Чтобы посмотреть, как можно использовать этот компонент, включите TDataSetPageProducer и компонент TTable в состав ***Web Module***. Присвойте объекту TTable наименование **TableBiolife**, свойству DatabaseName — значение **BCDEMOS**, свойству TableName — значение **biolife.db**, а свойству Active — значение **true** (таким образом, вам не придется самостоятельно открывать эту таблицу). Далее подключите свойство DataSet объекта TDataSetPageProducer к объекту TableBiolife, а свойству HTMLDoc присвойте следующее значение:

```
<H1>BIOLIFE Info</H1>
<HR>
<BR><B>Category:</B> <#Category>
<BR><B>Common_Name:</B> <#Common_Name>
<BR><B>Species Name:</B> <#Species Name>
<BR><B>Notes:</B> <#Notes>
```

Коды специальных тегов в этом тексте означают, что желательно увидеть значения из четырех полей таблицы BIOLIFE. Объект TDataSetPageProducer автоматически заменит специальные теги значениями этих полей и в программу понадобится добавить только текст обработчика события для TWebActionItem (и то, только в том случае, если вы откажетесь от использования свойства Producer, а в противном случае в программу вообще не придется включать ни одной дополнительной строки кода). Снова воспользуемся объектом TWebActionItem по умолчанию, в котором не специфицировано свойство PathInfo. Запустите ***Actions Editor***, щелкните на четвертом элементе action, перейдите на вкладку Events в окне Object Inspector и дважды щелкните на поле события OnAction. В обработчик этого события включите представленный ниже программный код.

```
void __fastcall TWebModule1::WebModule1WebActionItem4Action(
      TObject *Sender, TWebRequest *Request,
      TWebResponse *Response, bool &Handled)
{
  Response->Content = DataSetPageProducer1->Content();
}
```

Понадобится также изменить значение и в операторе ACTION= HTML-формы:

```
<FORM ACTION="http://localhost/scripts/WebShow.dll/dataset"
↳METHOD=POST>
```

Реакция ***Web Module*** на этот запрос представлена на рис. 13.9.

Хочу обратить ваше внимание на две погрешности в показанном результате. Во-первых, вы видите текст (MEMO) там, где должно быть выведено содержимое поля Notes. Во-вторых, не выведено значение поля Species Name.

Первую проблему несложно решить, приняв во внимание, что класс TDataSetPageProducer является наследником класса TPageProducer, а потому при обнаружении каждого специального #-тега возбуждается событие OnHTMLTag. В обработчике этого события нужно проверить значение аргумента ReplaceText. Если окажется, что это значение равно (MEMO), его нужно заменить реальным содержимым поля. Эта процедура выполняется с помощью свойства AsString этого поля, как показано в представленном ниже программном коде обработчика события.

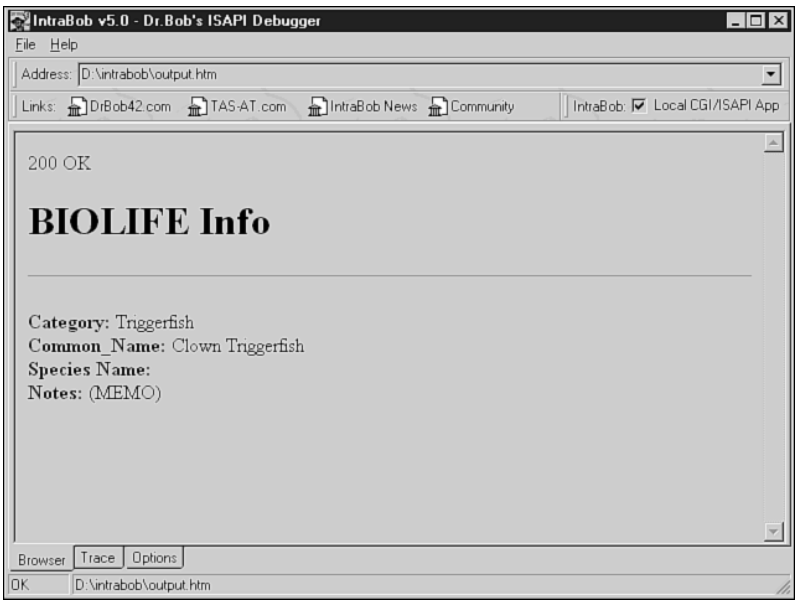


Рис. 13.9. Результат работы компонента `TDataSetPageProducer`

```
void __fastcall TWebModule1::DataSetPageProducer1HTMLTag(
      TObject *Sender, TTag Tag, const AnsiString TagString,
      TStrings *TagParams, AnsiString &ReplaceText)
{
  if (ReplaceText == "(MEMO)")
    ReplaceText = TableBiolife->FieldByName(TagString)->AsString;
}
```

Иногда может оказаться, что в объекте `TMemoFields` содержится значение `(Memo)`, а не `(MEMO)`. Отличие состоит в том, что значение `(Memo)` является индикатором пустого поля типа `memo`, а значение `(MEMO)` — индикатором заполненного поля. Именно этот второй вариант и используется в обработчике события `OnHTMLTag`.

Вторая погрешность в представленной экранной форме скорее всего объясняется тем, что в имени поля `Species Name` имеются символы пробела, а они воспринимаются как ограничители имени специального тега. В результате объект `TDataSetPageProducer` пытается отыскать поле с именем `Species`, а не поле `Species Name`.

Лично я всегда считал порочной практику использования пробелов в наименованиях полей таблиц баз данных. Но как бы мы не относились к этому, часто встречаются базы данных, организованные именно таким образом. Я в таких случая добавляю в таблицу вычисляемое поле с "корректным" именем (например, `SpeciesName`), в которое дублируется содержимое поля с "некорректным" именем.

Другой вариант решения этой проблемы опять же связан с использованием обработки события `OnHTMLTag`. На сей раз нужно заменить `TagString` со значением `#Species` содержимым поля `Species Name`. Как это делается, представлено в приведенном ниже варианте обработчика события `OnHTMLTag`.

```
void __fastcall TWebModule1::DataSetPageProducer1HTMLTag(
      TObject *Sender, TTag Tag, const AnsiString TagString,
```

```
      TStrings *TagParams, AnsiString &ReplaceText)
{
  if (TagString == "Species") // поле Species Name
    ReplaceText = TableBiolife->FieldByName(
             "Species Name")->AsString;
  else
    if (ReplaceText == "(MEMO)") // поле Notes
      ReplaceText =
        TableBiolife->FieldByName(TagString)->AsString;
}
```

После того как в программу были внесены описанные выше изменения, она отреагировала на запрос (рис. 13.10).

Рис. 13.10. Результат работы компонента `TDataSetPageProducer` после модификации программы

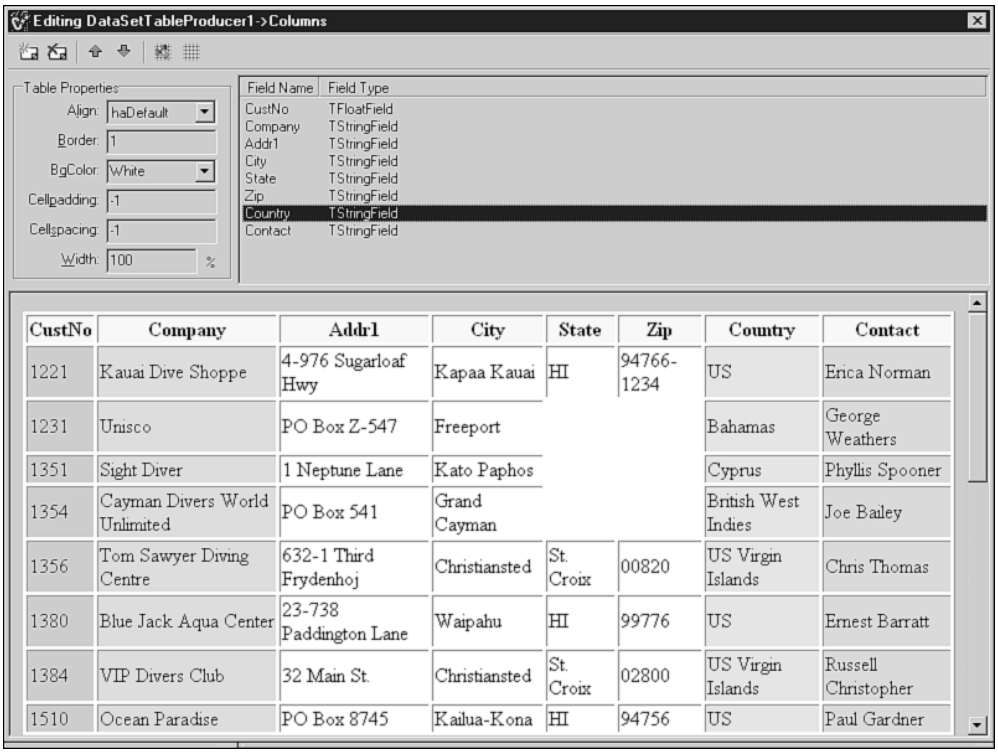
Если появится необходимость просматривать на экране более одной записи, это можно сделать двумя способами: пролистывать записи или показывать одновременно несколько записей. При переходе от одной записи к другой (следующей или предыдущей) потребуется помнить номер текущей записи, т.е. фиксировать и анализировать текущее состояние. Об этом мы поговорим в разделе *Обработка состояния* далее в этой главе. Если же вы более склонны к тому, чтобы представлять на экране сразу несколько записей, то это можно организовать с помощью другого WebBroker-компонента —`TDataSetTableProducer`.

Компонент `TDataSetTableProducer`

Как и `TDataSetPageProducer`, компонент `TDataSetTableProducer` использует свойство `DataSet`. Но в данном случае с помощью этого свойства можно извлечь содержимое нескольких записей, а результат будет сформатирован в виде таблицы.

"Сбросьте" вторую таблицу на поле ***Web Module***. Присвойте объекту `TTable` наименование **`TableCustomer`**, свойству `DatabaseName` — значение **`BCDEMOS`**, свойству `TableName` — значение **`customer.db`**, а свойству `Active` — значение **`true`**. Далее включите в ***Web Module*** объект компонента `TDataSetTableProducer` и установите значение **`TableCustomer`** его свойства `DataSet`. В структуре компонента `TDataSetTableProducer` имеется множество свойств, с помощью которых можно контролировать процесс формирования HTML-кода. Свойства `Header` и `Footer` содержат текст, который будет выводиться на экран до и после таблицы. Свойства `TableAttributes` и `RowAttributes` можно использовать для настройки внешнего вида всей таблицы и отдельных строк (выравнивание, цвет и т.д.).

Для настройки некоторых свойств, например `Column`, используются визуальные средства, в частности редактор свойств колонок ***Column property editor***. Запустить ***Columns property editor*** можно из диалогового окна Object Inspector, щелкнув на поле с многоточием рядом со свойством `Columns` (это свойство имеет тип `THTMLTableColumns`). На экран будет выведено диалоговое окно для настройки свойства `DataSetTableProducer1->Columns` (рис. **13.11.**).



CustNo	Company	Addr1	City	State	Zip	Country	Contact
1221	Kauai Dive Shoppe	4-976 Sugarloaf Hwy	Kapaa Kauai	HI	94766-1234	US	Erica Norman
1231	Unisco	PO Box Z-547	Freeport			Bahamas	George Weathers
1351	Sight Diver	1 Neptune Lane	Kato Paphos			Cyprus	Phyllis Spooner
1354	Cayman Divers World Unlimited	PO Box 541	Grand Cayman			British West Indies	Joe Bailey
1356	Tom Sawyer Diving Centre	632-1 Third Frydenhoj	Christiansted	St. Croix	00820	US Virgin Islands	Chris Thomas
1380	Blue Jack Aqua Center	23-738 Paddington Lane	Waipahu	HI	99776	US	Ernest Barratt
1384	VIP Divers Club	32 Main St.	Christiansted	St. Croix	02800	US Virgin Islands	Russell Christopher
1510	Ocean Paradise	PO Box 8745	Kailua-Kona	HI	94756	US	Paul Gardner

Рис. 13.11. Окно редактора столбцов компонента `TDataSetTableProducer`

Поскольку открыта таблица `TableCustomer`, атрибуты всех ее полей будут представлены в окне редактора колонок. Сейчас невозможно удалить какое-либо из полей или изменить их порядок. Это объясняется тем, что ранее не были отобраны поля, которые планируется выводить в приложении. Таково поведение всех компонентов, производных от `TDataSet`: если пользователь не указал, с какими полями он будет работать, ему предлагается работать со *всеми* полями. Однако у пользователя есть возможность, добавив

“физический” список полей, удалять поля или менять их очередность. Щелкните на списке полей правой кнопкой мыши и выберите в контекстном меню команду Add All Fields (добавить все поля). Видимых изменений после этого вы не заметите. Но теперь список полей по умолчанию станет действительным списком, и в нем можно будет удалять поля или менять порядок их следования.

С помощью специальных опций пользователь может настраивать внешний вид таблицы на экране. Например, установка `Border=1` приведет к выводу рамки, задание значения свойства `BgColor` определит цвет фона и т.д. Обратите внимание на то, что настройка отдельных полей (т.е. соответствующих колонок) производится после выбора определенного поля. Далее в окне Object Inspector можно установить значения свойств этой колонки — `BgColor`, `Align` (**`left`**, **`center`**, **`right`**) и `VAlign` (**`top`**, **`middle`**, **`bottom`**, **`baseline`**). Текст заголовка колонки настраивается свойством `Title` (опять же в окне Object Inspector). Свойство `Title`, в свою очередь, имеет параметры настройки (свойства) — `Align` (задает выравнивание только заголовка, а не полей данных) и `Caption`. Сам текст заголовка задается в окне Object Inspector в свойстве `Caption` элемента `Title`. Внесенные изменения автоматически отображаются в ***Columns editor***. После некоторых манипуляций предварительное изображение таблицы может выглядеть, как на рис. **13.11.**

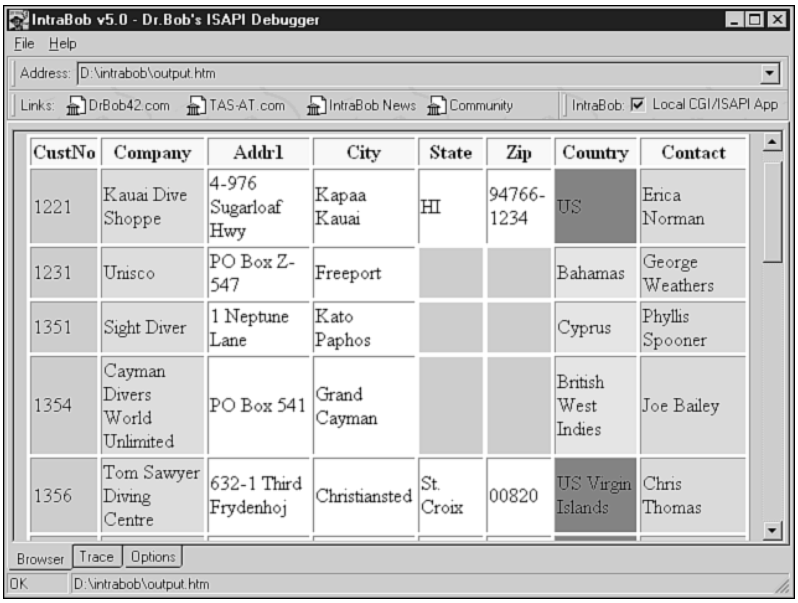
На этом настройка внешнего вида таблицы, формируемой с помощью `TDataSetTableProducer`, завершается. Обратите внимание, что для настройки нам не понадобилось написать ни единой строки программного кода. Заняться программным кодом нам придется только с одной целью — для обработки события `OnAction` объекта `WebActionItem`:

```
void __fastcall TWebModule1::WebModule1WebActionItem5Action(
      TObject *Sender, TWebRequest *Request,
      TWebResponse *Response, bool &Handled)
{
  Response->Content = DataSetTableProducer1->Content();
}
```

При желании можно еще немного поработать над внешним видом таблицы. Например, иногда желательно выделить в ней цветом определенные записи (например, записи о клиентах, принадлежащих к определенной категории). Если в ячейке таблицы отсутствуют данные, то такую ячейку можно также выделить отдельным цветом. Все эти настройки можно выполнить при помощи обработчика события `OnFormatCell` объекта `TDataSetTableProducer`, как показано в представленном ниже фрагменте кода.

```
void __fastcall TWebModule1::DataSetTableProducer1FormatCell(
      TObject *Sender, int CellRow, int CellColumn,
      THTMLBgColor &BgColor, THTMLAlign &Align,
      THTMLVAlign &VAlign, AnsiString &CustomAttrs,
      AnsiString &CellData)
{
  if (CellData == "") BgColor = "Silver";
  else
    if ((CellColumn == 6) && (CellData.Pos("US") > 0))
      BgColor = "Red";
}
```

При выполнении созданной таким образом программы таблица в окне броузера будет выглядеть примерно так, как на рис. 13.12. Хотя рисунок и черно-белый, но на нем видно отличие в цвете одних полей таблицы от других.



CustNo	Company	Addr1	City	State	Zip	Country	Contact
1221	Kauai Dive Shoppe	4-976 Sugarloaf Hwy	Kapaa Kauai	HI	94766-1234	US	Erica Norman
1231	Unisco	PO Box Z-547	Freeport			Bahamas	George Weathers
1351	Sight Diver	1 Neptune Lane	Kato Paphos			Cyprus	Phyllis Spooner
1354	Cayman Divers World Unlimited	PO Box 541	Grand Cayman			British West Indies	Joe Bailey
1356	Tom Sawyer Diving Centre	632-1 Third Frydenhoj	Christiansted	St. Croix	00820	US Virgin Islands	Chris Thomas

Рис. 13.12. Результат работы компонента `TDataSetTableProducer` *в окне программы* **IntraBob**

Внесем в приложение еще одно усовершенствование. Пользователю иногда необходимо просмотреть в отдельном окне информацию, касающуюся определенных клиентов (например, о сделанных ими заказах), причем эта информация, как правило, содержится в другой таблице. Заказы связаны с полем `CustNo`, поэтому можно заменить его содержимое связью, которая сформирует другой запрос к ***Web Module***, что приведет к динамической генерации другой HTML-страницы. В этой новой странице будет представлена информация о заказах, сделанных данным клиентом. Для реализации такой функции нам потребуется компонент, который мы еще не рассматривали, — `TQueryTableProducer`.

Компонент TQueryTableProducer

Компонент `TQueryTableProducer` формирует почти такой же результат на выходе, как и компонент `TDataSetTableProducer`. Отличие состоит лишь в том, что к этому компоненту подсоединяется объект класса `TQuery`, а не `TTable`. Следует отметить, что к `TDataSetTableProducer` можно подключить любой объект класса `TDataSet` или производного от него, в том числе и `TTable` или `TQuery`. Но компонент `TQueryTableProducer` предоставляет в распоряжение разработчика специальные средства настройки значений параметров параметризованных запросов `TQuery`.

Включите в состав приложения ***Web Module*** компоненты `TQueryTableProducer` и `TQuery`. Свойству `DatabaseName` (псевдоним) объекта `TQuery` присвойте значение **`BCDEMOS`**, дайте объекту новое имя **`QueryOrders`** и введите следующий код в свойство `SQL`:

```
SELECT * FROM ORDERS.DB AS O
WHERE (O.CustNo = :CustNo)
```

Это SQL-запрос с одним параметром. Теперь нужно указать тип этого параметра в свойстве `Parameter` с помощью редактора параметров ***Property Editor*** компонента `TQuery`. Щелкните на

многоточии рядом со свойством Params в окне Object Inspector. Выберите в списке параметр CustNo и перейдите в окно Object Inspector. Установите в строке свойства DataType значение **ftInteger**, в строке свойства ParamTyp значение **ptInput**, а в строке свойства Value значение по умолчанию **0** (это позволит активизировать запрос на стадии разработки).

Можно открыть объект TQuery (для этого нужно присвоить свойству Active значение **true**) и посмотреть, не вкралась ли ошибка при вводе значений с клавиатуры. Теперь щелкните на строке TQueryTableProducer и присвойте свойству Query значение **QueryOrders**. Обратите внимание на то, что в структуре компонента TQueryTableProducer имеются те же свойства для настройки внешнего вида таблицы, что и в структуре компонента TDataSetTableProducer. Фактически оба компонента — и TQueryTableProducer, и TDataSetTableProducer — являются производными от TDSTableProducer, причем в TQueryTableProducer добавлены только специфические средства работы с запросами. Прежде чем перейти к следующему этапу, отредактируйте свойство Columns и формат выходной таблицы по своему вкусу.

Компонент TQueryTableProducer отыскивает имя параметра запроса (в данном случае CustNo) среди ContentFields (или QueryFields, если используется метод GET) и присваивает параметру значение этого поля. В данном случае нам потребуется простенький HTML-файл запуска, текст которого приведен ниже:

```
<HTML>
<BODY>
<H1>WebBroker HTML Form</H1>
<HR>
<FORM ACTION="http://localhost/scripts/WebShow.dll/query" METHOD=POST>
CustNo: <INPUT TYPE=EDIT NAME=CustNo>
<P>
<INPUT TYPE=SUBMIT>
</FORM>
</BODY>
</HTML>
```

Обратите внимание на то, что имя входного поля — CustNo — совпадает с именем параметра запроса Query. Если ввести в него определенное значение, например 1221, то можно извлечь все заказы этого клиента. Если свойству MaxRows присвоено достаточно большое значение (лучше всего 99999999), то можно быть уверенным, что будут считаны все записи заказов. Учтите, что установка в MaxRows большого значения приведет к тому, что будет показано больше записей, но при этом увеличится время появления результата и объем выходных данных (особенно при работе с компонентом TDataSetTableProducer). Снижение скорости объясняется не только тем, что через сеть придется передать больший объем информации, но и тем, что HTML-таблица не выводится на экран, пока не обнаружен завершающий ее тег </TABLE>. Все это может привести к тому, что при работе с действительно большой таблицей (например, содержащей 99999999 строк) пользователь будет долго созерцать пустой экран и только потом на нем появится желанная таблица.

Для завершения этого примера нам понадобится только небольшой фрагмент кода обработчика события OnAction для запроса /query в объекте WebActionItem:

```
void __fastcall TWebModule1::WebModule1WebActionItem6Action(
      TObject *Sender, TWebRequest *Request,
      TWebResponse *Response, bool &Handled)
{
  Response->Content = QueryTableProducer1->Content();
}
```

Если запустить на выполнение объект `WebActionItem`, в котором свойство `PathInfo` имеет значение **`/query`**, и ввести значение `1221` в поле CustNo, то получим результат, показанный на рис. 13.13.

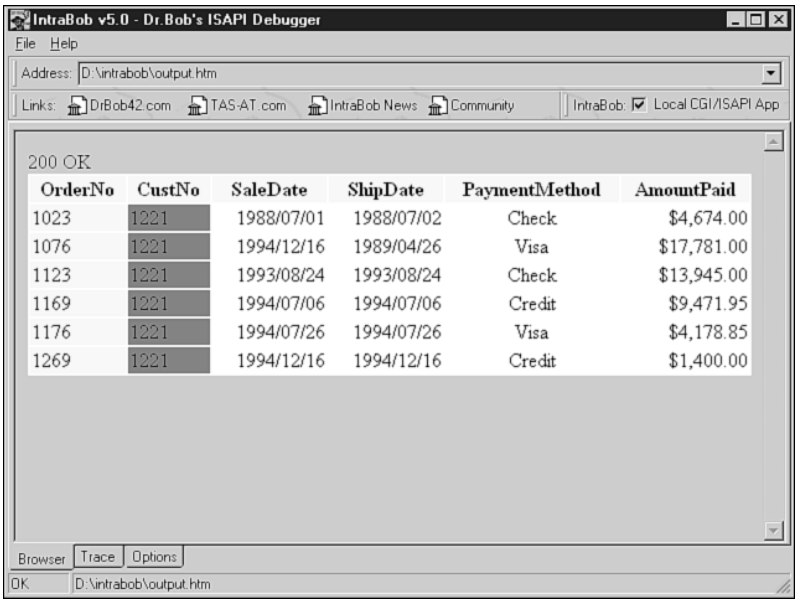


Рис. 13.13. Результат работы компонента `TQueryTableProducer` в окне программы **IntraBob**

Теперь нужно подсоединить программу обработки запроса к таблице из предыдущего примера, в котором использовался компонент `TDataSetTableProducer`. Это делается с помощью обработчика события `OnFormatCell` объекта `TDataSetTableProducer`. В нем нужно заменить `CustNo` на запрос гиперсвязи с тем же Web-сервеным приложением, но свойству `PathInfo` должно быть присвоено значение **`/query`**, вслед за которым нужно с помощью протокола `GET` передать пару поле/значение для `CustNo`. Таким образом, в первой колонке (в ней `CellColumn` имеет значение `0`) нужно заменить реальное значение `CellData` на гиперсвязь к `/query` `WebActionItem` с текущим `CellData` (другими словами, `CustNo`) в качестве значения поля, названного `CustNo`, и все это передать с помощью протокола `GET`.

Существует и другой вариант: можно заменить каждое значение `CustNo` новой формой с операцией `/query` и скрытым полем, в котором будет имя `CustNo` и определенное значение `CustNo`. Обе опции реализованы в модифицированном обработчике события `OnFormatCell`, который представлен в листинге 13.3.

Листинг 13.3. Обработчик события `OnFormatCell` компонента `TDataSetTableProducer`

```
void __fastcall TWebModule1::DataSetTableProducer1FormatCell(
      TObject *Sender, int CellRow, int CellColumn,
      THTMLBgColor &BgColor, THTMLAlign &Align,
      THTMLVAlign &VAlign, AnsiString &CustomAttrs,
      AnsiString &CellData)
{
```

```
  // Первая колонка - CustNo
  if ((CellColumn == 0) && (CellRow > 0))
    CellData =
#ifdef LINK
      "<A HREF=\
↳"http://localhost/scripts/WebShow.dll/query?CustNo=" +
        CellData + ">" + CellData + "</A>";
#else
      (AnsiString)"<FORM ACTION=\ "WebShow.dll/query\"
↳METHOD=POST>" +
      "<INPUT TYPE=HIDDEN NAME=CustNo VALUE=" + CellData + ">" +
      "<INPUT TYPE=SUBMIT VALUE=" + CellData + ">" +
      "</FORM>";
#endif
  else
  if (CellData == "") BgColor = "Silver";
  else
    if ((CellColumn == 6) && (CellData.Pos("US") > 0))
      BgColor = "Red";
} ;
```

Для тестирования опции гиперсвязи нужно использовать реальный Web-сервер. Программу ***IntraBob*** можно использовать для тестирования и отладки запросов ISAPI, которые запускаются из HTML-формы. Можно протестировать опцию Form, которая должна давать результат, подобный тому, что представлен на рис. 13.14 (здесь использован компонент `TDataSetTableProducer` и таблица `Customer`).

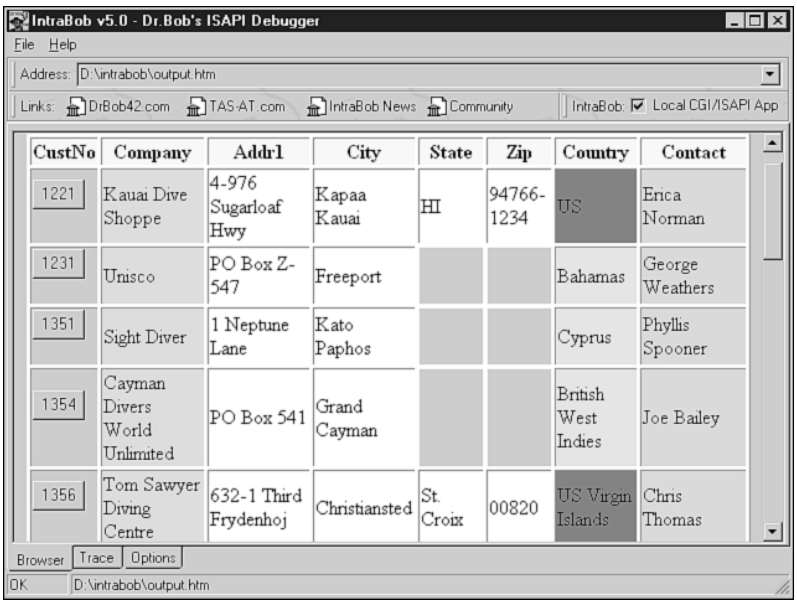


Рис. 13.14. Основная форма, подготовленная компонентом `TDataSetTableProducer`, в окне программы **IntraBob**

Если щелкнуть на одной из кнопок, соответствующих CustNo (например, на кнопке с номером 1231), то на экране появится новая форма, в которой представлены все заказы, сделанные этой компанией (рис. 13.15).

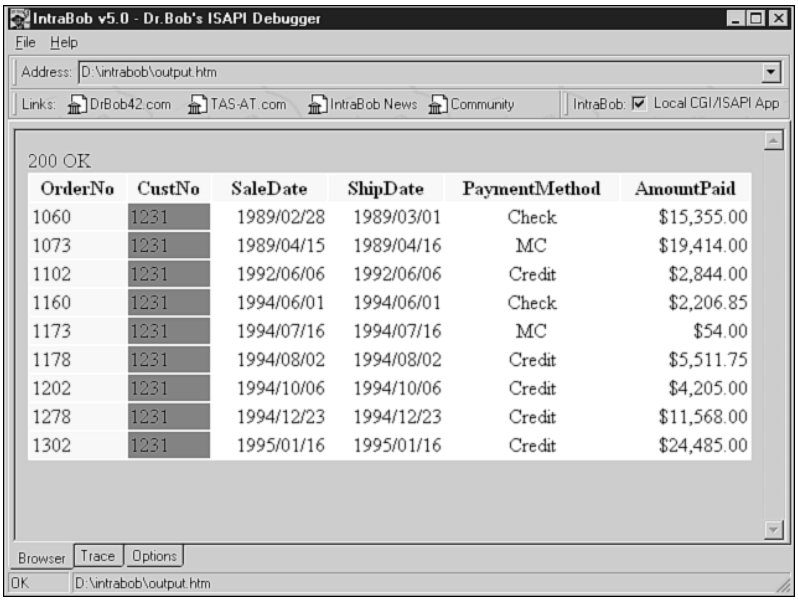


Рис. 13.15. Дочерняя форма, подготовленная компонентом `TQueryTableProducer`*, в окне программы* **IntraBob**

Мастера Web-приложений

Перед тем как продолжить описание приемов работы с компонентами ***WebBroker***, рассмотрим кратно возможности мастера, имеющие отношение к компоненту `TDataSetTableProducer`. Мастер ***Database Web Application Wizard***, который находится на вкладке Business

Object Repository, позволяет задать псевдонимы баз данных, имена таблиц, некоторые имена полей и ряд свойств компонента `TDataSetTableProducer`. Затем мастер формирует новое приложение ***Web Module***, в которое включаются объект `TWebModule`, объект `TSession` (`AutoSessionName = true`), объект `TTable` и объект `TDataSetTableProducer`.

Для того чтобы предотвратить возможный конфликт сеансов BDE, свойство `AutoSessionName` объекта `TSession` должно иметь значение `true`. Все остальное, что делает мастер ***Database Web Application Wizard***, вполне можно сделать и без него. Лично я никогда им не пользуюсь.

Вы, наверное, уже подметили, что технология ***WebBroker*** особенно подходит для формирования HTML на базе шаблонов или заданных свойств. Но при этом у пользователя практически отсутствует возможность активно взаимодействовать с созданной таким способом формой. Для создания современных динамических Web-страниц возможности ***WebBroker*** нужно дополнить возможностями ***InternetExpress*** (об этом мы поговорим в конце этой главы). А сейчас перейдем к теме управления состояниями и безопасностью.

Обработка состояния

В одном из предыдущих примеров для просмотра таблицы TableBiolife мы использовали объект TDataSetPageProducer. В этом примере на экран Web-броузера выводилась одна запись таблицы и шла речь о том, что желательно было бы организовать работу приложения так, чтобы пользователь мог "перелистывать" страницы с записями в любом направлении, т.е. Web-страница обладала бы всеми привычными функциями просмотра записей таблиц, включая переход к первой или последней записи. Все это можно реализовать в Web-приложении с помощью компонента TDataSetPageProducer и небольшого дополнения к разработанному ранее программному коду.

Основная проблема, с которой нам предстоит столкнуться при этом, — обработка информации о состоянии. Какая запись (под каким номером) просматривается в текущий момент? Протокол HTTP сам по себе не сохраняет информации о состоянии, поэтому нам самим придется позаботиться об этом.

Сохранение информации о состоянии можно реализовать тремя способами: использовать расширение адреса URL, сообщений *cookies* и скрытых полей.

Расширенные URL

Весьма распространенный способ сохранить информацию о состоянии — добавив переменную Form, присвоить ей значение URL. Мы уже использовали этот прием, когда добавляли к URL ключевое значение поля CustNo. Нечто подобное можно сделать и в HTML-форме — для этого нужно модифицировать операцию ACTION. Если, например, в текущем состоянии просматривается первая запись таблицы, фрагмент HTML-кода будет выглядеть так:

```
<FORM ACTION="http://localhost/scripts/WebShow.dll/dataset?RecNo=1"
↳METHOD=POST>
```

Обратите внимание на то, что общим методом пересылки переменной Form по-прежнему остается POST, хотя переменная состояния (RecNo) передается с помощью протокола GET. Это значит, что мы увидим RecNo и ее значение в составе URL, который можно проанализировать с помощью какого-либо механизма отслеживания Web.

Лично я придерживаюсь того мнения. что любая информация, посылаемая с URL, является потенциальным источником ошибок, потому, как правило, я стараюсь не применять такой способ (хотя использование метода POST для передачи обычных полей формы и метода GET для передачи полей состояния и позволяет разделить эти два вида информации).

Сообщения *cookies*

Сообщения *cookies* посылаются сервером броузеру. Если вы используете такие сообщения. то инициатива их пересылки принадлежит серверу, а клиент имеет возможность отказаться от них или заблокировать. Иногда сервер посылает cookies, хотя вы его об этом и не просили. Именно по этой причине многие не любят такого рода сообщения (к их числу принадлежу и я).

Сообщения *cookies* можно включить в качестве составного элемента в Response с помощью метода SetCookieField. Сообщения *cookie* имеют вид NAME=VALUE, а потому можно вставить RecNo=value в Response следующим образом:

```
TStringList* Cookies = new TStringList();
Cookies->Add("RecNo="+IntToStr(Master->RecNo));
Response->SetCookieField(Cookies,NULL,NULL,Now()+1,false);
Cookies->Free();
```

Обратите внимание на то, что в этом фрагменте используется объект класса TStringList для создания списка сообщений *cookie*. Каждый такой список может иметь ассоциированный домен и путь, которые служат индикатором URL, по которым должны рассылаться эти сообщения. Вы можете не заполнять их. Четвертый параметр определяет дату истечения срока действия *cookie*, и ему можно присвоить значение Now+1. Последний аргумент указывает, используется ли *cookie* при безопасном подключении.

Установить значение cookie — это только полдела. Далее при обработке события OnAction нужно будет считать переданное значение и определить, куда перейти относительно текущей записи в таблице Master. В этом случае cookie становятся частью класса Request, точно так же, как и ContentFields, и это сообщение можно получить с помощью свойства CookieFields.

```
int RecNo = StrToInt(Request->CookieFields->Values["RecNo"]);
```

Во всем остальном сообщения *cookie* работают так же, как и любое поле CGI.

Скрытые поля

Третий способ сохранения информации о текущем состоянии основан на использовании скрытых полей. По моему мнению, этот способ наиболее гибкий и предоставляет разработчику самые широкие возможности. Для реализации скрытых полей сначала нужно переписать HTML-форму (и поместить ее в свойство HTMLDoc), задав объект WebActionItem по умолчанию и включив в нее четыре кнопки отсылки с разными надписями. Для хранения номера текущей записи в HTML-форму включается специальный #-тег с именем RecNo. Обработчик события OnHTMLTag заменит этот тег номером текущей записи. Текст HTML-формы приведен ниже.

```
<FORM ACTION="http://localhost/scripts/WebShow.dll/dataset" METHOD=POST>
<H1>BIOLIFE Info</H1><HR>
<INPUT TYPE=SUBMIT NAME=SUBMIT VALUE="First">
<INPUT TYPE=SUBMIT NAME=SUBMIT VALUE="Prior">
<INPUT TYPE=SUBMIT NAME=SUBMIT VALUE="Next">
<INPUT TYPE=SUBMIT NAME=SUBMIT VALUE="Last">
<#RecNo>
<BR><B>Category:</B> <#Category>
<BR><B>Common_Name:</B> <#Common_Name>
<BR><B>Species Name:</B> <#Species Name>
<BR><B>Notes:</B> <#Notes>
</FORM>
```

Для того чтобы заменить тег #RecNo номером текущей записи, мы воспользовались синтаксисом скрытых полей:

```
<INPUT TYPE=HIDDEN NAME=RecNo VALUE=1>
```

Этот оператор означает, что скрытое поле, названное RecNo, имеет значение 1. Скрытые поля невидимы для конечного пользователя, а их значения отсылаются обратно Web-серверу и приложению ***Web Module***, как только пользователь щелкнет на одной их четырех кнопок отсылки. В обработчике события OnHTMLTag объекта DataSetPageProducer можно организовать вывод на экран некоторой информации, в частности номера текущей записи и общего количества записей в таблице (листинг 13.4).

Листинг 13.4. Обработчик события `OnHTMLTag` объекта `DataSetPageProducer`

```
void __fastcall TWebModule1::DataSetPageProducer1HTMLTag(
      TObject *Sender, TTag Tag, const AnsiString TagString,
      TStrings *TagParams, AnsiString &ReplaceText)
{
  if (TagString == "RecNo")
    ReplaceText =
      "<INPUT TYPE=HIDDEN NAME=RecNo VALUE=" +
        IntToStr(TableBiolife->RecNo) + // номер текущей записи
      "> " + IntToStr(TableBiolife->RecNo) +"/"+
             IntToStr(TableBiolife->RecordCount) + "<P>";
  else
    if (TagString == "Species") // поле Species Name
      ReplaceText =
         TableBiolife->FieldByName(
                "Species Name")->AsString;
    else
      if (ReplaceText == "(MEMO)") // поле Notes
        ReplaceText =
           TableBiolife->FieldByName(TagString)->AsString;
}
```

После этого остается только задать операцию, которую должен выполнить объект `WebActionItem` после щелчка на каждой из четырех кнопок. Можно было бы, конечно, распределить эти операции между четырьмя разными объектами, но в этом случае придется использовать четыре формы и четыре копии каждого скрытого поля и всей другой необходимой информации. Программный код обработчика события по умолчанию объекта `WebActionItem` представлен в листинге 13.5. Этот обработчик должен получить значение скрытого поля `RecNo` и информацию о том, какая из четырех кнопок отсылки была активизирована.

Листинг 13.5. Обработчик события `WebActionItem`

```
void __fastcall TWebModule1::WebModule1WebActionItem4Action(
      TObject *Sender, TWebRequest *Request,
      TWebResponse *Response, bool &Handled)
{
  // Если набор данных еще не открыт!
  DataSetPageProducer1->DataSet->Open();
  int RecNr = 0;
  AnsiString Str = Request->ContentFields->Values["RecNo"];
  if (Str != "") RecNr = StrToInt(Str);
  Str = Request->ContentFields->Values["SUBMIT"];
  if (Str == "First") RecNr = 1;
  else
    if (Str == "Prior") RecNr-;
    else
      if (Str == "Last") RecNr =
        DataSetPageProducer1->DataSet->RecordCount;
      else // if Str = 'Next' then {  default }
        RecNr++;
```

```
  if (RecNr > DataSetPageProducer1->DataSet->RecordCount)
    RecNr = DataSetPageProducer1->DataSet->RecordCount;
  if (RecNr < 1) RecNr = 1;
  if (RecNr != DataSetPageProducer1->DataSet->RecNo)
    DataSetPageProducer1->DataSet->MoveBy(
      RecNr - DataSetPageProducer1->DataSet->RecNo);
  Response->Content = DataSetPageProducer1->Content();
}
```

Результат работы программы очень похож на тот, который был получен в одном из предыдущих примеров (см. рис. 13.10), но на сей раз в экранной форме появились четыре кнопки, манипулируя которыми пользователь может переходить к первой, последней, следующей или предыдущей записи просматриваемой таблицы BIOLIFE. На рис. 13.16 показано, как будет выглядеть результат работы приложения после того, как пользователь трижды щелкнет на кнопке Next.

Рис. 13.16. Результат работы компонента `TDataSetPageProducer` *после модификации программы*

Использованный в этом примере метод сохранения информации о текущем состоянии можно применять и в других аналогичных приложения. Но хочу обратить ваше внимание на одну тонкость. В этом примере для хранения номера текущей записи используется поле RecNo, но кроме него нужно также передавать и уникальный ключ записи, поскольку для поиска записи в динамической таблице, к которой имеет доступ множество пользователей (причем никто не гарантирует, что кто-то из них не добавляет при этом новые записи в таблицу), нужен именно уникальный ключ этой записи.

Если вы пожелаете наделить свое приложение более широкими возможностями работы с таблицами, то понадобится использовать ***InternetExpress*** — расширение ***WebBroker***. Но прежде чем перейти к описанию ***InternetExpress*** , остановимся еще на одном вопросе, которому в последнее время уделяется много внимания, — обеспечении безопасности Web-приложений.

Безопасность в сети Web

Обеспечение безопасной работы в сети Web является на сегодняшний день одной из наиболее острых проблем, особенно учитывая все более широкое распространение "электронной" коммерции и непрерывно поступающие сообщения об атаках хакеров и "крэкеров". При создании даже простейших Web-серверных приложений нельзя упускать из поля зрения вопросы безопасности. Ниже мы познакомим вас с основными средствами, которые используются для защиты Web-приложений от нежелательного вмешательства.

Прежде всего никогда не размещайте саму базу данных на Web-сервере. Между Web-сервером и сетью Internet необходимо установить брандмауэр, а другой брандмауэр — между Web сервером и локальной сетью, в которой и размещается база данных. Таким образом, ваш Web-сервер оказывается в так называемой "демилитаризованной" зоне и для взаимодействия как с окружающим миром, так и с локальной сетью должен преодолевать определенный защитные барьеры.

Для поддержки работы с прокси-сервером в состав C++Builder 5 включен новый компонент `TWebConnection`. Этот компонент используется в клиентской части многоуровневого MIDAS-приложения, которое наделено возможностью подключения к удаленной базе данных через прокси-сервер HTTP. В архитектуре многоуровневых приложений удаленная база данных представляет собой отдельный уровень, размещенный в пределах локальной сети. Для подключения используется специальный DLL-модуль `httpsrvr.dll`, работающий с протоколом ISAPI. Этот модуль нужно установить на компьютер и обеспечить доступ к нему со стороны Web-серверного приложения. В структуре компонента `TWebConnection` имеется свойство `Proxy`, в котором нужно задать IP-адрес (или имя) прокси-сервера. Для доступа через прокси-сервер к уровеню базы данных в объекте `TWebConnection` нужно установить значения свойств `UserName` (имя пользователя) и `Password` (пароль). Учтите, что в самом объекте `TWebConnection` имя пользователя и пароль не проверяются — он просто пересылает их прокси-серверу, который специфицирован в свойстве `Proxy`.

Если прокси-сервер в приложении не используется, то свойства `Proxy`, `Username` и `Password` нужно очистить. Но в такой архитектуре для подключения можно использовать и другие компоненты — `TDCOMConnection` или `TCORBAConnection` – и нет необходимости подключаться с помощью протокола HTTP.

Протокол Secure Sockets Layer

Помимо защиты Web-сервера и серверов баз данных с помощью брандмауэров, необходимо позаботиться и о защите данных в процессе пересылки. Это особенно важно в тех случаях, когда в процессе работы приложения через сеть Web передается конфиденциальная информация, например реквизиты клиентов и данные о финансовых документах (номера кредитных карточек). Функции защиты информации в процессе передачи (защищенного сеанса работы) реализуются специальным протоколом шифрования SSL (Secure Socket Layer). Индикатором использования этого протокола является изображение закрытого замка в окне броузера. Протокол SSL использует специальную разновидность протокола HTTP — Secure HTTP (в документации часто встречается сокращение S-HTTP). При использовании этого протокола префикс `http://` заменяется префиксом `https://`. Для того чтобы воспользоваться средствами поддержки протокола SSL, вам прежде всего нужно получить и установить на своем компьютере общедоступный и личный ключи шифрования. Их можно получить в специальной службе сертификации (Certificate Authority). Одной из самых известных служб такого рода является ***VeriSign***. При первом подключении `https://` общедоступный ключ пере-

дается от сервера клиенту, который затем использует его для шифрования всех данных. Личный ключ, полученный сервером, используется для расшифровки данных и работы с ними. Для разработчика Web-серверных приложений, использующего ***WebBroker***, протокол SSL может быть достаточно прозрачным.

Авторизация

Помимо использования протокола SSL для защиты поступающей информации, иногда возникает необходимость обеспечить доступ к определенным Web-сайтам только отдельным пользователям. Это означает, что нужно включить в приложение средства авторизации доступа. Как правило, для этого вполне достаточно использовать те средства авторизации, которые предусмотрены в HTTP. Именно так мы и поступим в примере, который будет рассмотрен ниже.

Эти средства авторизации предусматривают проверку имени пользователя и пароля и формирование адекватного ответа. Хотя разрешение доступа к определенным каталогам можно организовать и с помощью IIS, мы покажем, как для решения аналогичной задачи использовать средства, имеющиеся в составе C++Builder.

Заголовки HTTP

Результат выполнения CGI-приложения или ISAPI DLL включает заголовок HTTP, за которым следует пустая строка, а уже за ней передается содержательная информация. В заголовке HTTP специфицируется MIME-тип содержимого (обычно `text/html`, но может быть и `image/gif` или что-нибудь подобное, если передается изображение). Получив такой заголовок, броузер может подготовиться к приему соответствующей информации. Сам Web-сервер добавляет в заголовок специальную первую строку, которая обычно выглядит так: `HTTP/1.0 200 OK`. Это строка, которая содержит код результата динамической Web-страницы: “Все прекрасно, можно выводить на экран то, что последует за заголовком”.

Предположим, что вместо этого текста мы поместим в первую строку заголовка специальный код ошибки HTTP, а именно `401`. Этот код означает “отсутствует авторизация”, и от пользователя требуется сначала зарегистрироваться на сервере. Подобный HTTP-заголовок представлен ниже.

```
HTTP/1.0 401 Unauthorized
content-type: text/html
WWW-Authenticate: Basic realm="/DrBob"
```

В обработчике события `OnAction` объекта `WebActionItems` можно использовать аргумент `Request` (его тип — `TWebRequest`). Этот аргумент располагает свойством `Authorization`, в котором содержится информация, касающаяся HTTP-авторизации, закодированная по основанию 64.

Для формирования HTTP-заголовка используется аргумент `Response`, в частности его свойства `StatusCode` и `ReasonString`. Тип авторизации нужно указать в свойстве `WWWAuthenticate`. Я обычно пользуюсь только типом `Basic`. Текст программы, формирующей такой HTTP-заголовок, приведен в листинге 13.6.

Листинг 13.6. Формирование запроса авторизации

```
void __fastcall TWebModule1::WebModule1WebActionItem7Action(
      TObject *Sender, TWebRequest *Request,
      TWebResponse *Response, bool &Handled)
{
```

```
  AnsiString Auth = Request->Authorization;
  if (Auth.Pos("Basic ") == 1) Auth.Delete(1,6);
  if (Auth.Pos("bswart") == 0)
  {
    Response->StatusCode = 401;
    Response->ReasonString = "Unauthorized";
    Response->WWWAuthenticate = "Basic";
    Response->Realm = "/DrBob";
    Response->SendResponse();
  }
  else
    Response->Content =
      "Welcome:  ["+Request->Authorization+"]=["+Auth+"])";
}
```

После завершения формирования этого заголовка нужно сразу же вызвать метод `Response.SendResponse()` и отослать заголовок броузеру клиента.

Учтите, что приведенный в листинге 13.6 программный код — это только пример того, как можно извлечь информацию из `Response.Authorization` в приложении ***WebBroker*** или ***InternetExpress***.

Если мы разработаем простое консольное приложение ***WebBroker***, которое отошлет эти несколько строк, то в броузере будет выведена пустая страница. Это произойдет по той простой причине, что Web-сервер п вставит в заголовок строку `HTTP/1.0 200 OK`, а текст, который сформирован в приведенной программе, пойдет после этой строки и будет проигнорирован броузером. Если нужно заставить Web-сервер не вставлять первую строку, придется переименовать приложение — добавить к имени префикс `NPH-` (NPH означает Non-Protocol Header), т.е. назвать модуль приложения `NPH-WebShow.dll`.

Ошибки в библиотеке VCL Delphi

В методах `TISAPIResponse.SendResponse` и `TCGIResponse.SendResponse`, определенных в файлах `IsapiApp.pas` и `CgiApp.pas`, я обнаружил ошибку, которую можно исправить следующим образом. Вместо оператора вызова функции

```
AddHeaderItem(WWWAuthenticate, 'WWW-Authenticate: %s'#13#10);
```

вставьте два оператора (не забудьте и об объявлении переменной `Tmp` типа `String` перед началом процедуры):

```
Tmp := Format('WWW-Authenticate: %s realm="%s"'#13#10, ['%s',
       Realm]);
AddHeaderItem(WWWAuthenticate, Tmp);
```

В первом операторе заменяется строка `realm`, а функция `AddHeaderItem` вставляет строку `WWWAuthenticate` на предназначенное для нее место.

Эти изменения в файлах `CGIApp.pas` и `ISAPIApp.pas` позволяют справиться с проблемой, которую не заметили разработчики Delphi. Теперь перенесите модифицированные Pascal-модули в каталог, где находится ваш ***WebBroker***-проект, и включите этот модуль в состав проекта. После полной компиляции файлов проекта в него войдут и изменения, внесенные в Pascal-модули. Таким образом вы исправите ошибку в исходных файлах библиотеки VCL Delphi.

Создание защищенных Web-приложений

Надеюсь, что примеры, рассмотренные в предыдущих разделах этой главы, убедили вас в том, что с помощью C++Builder можно создавать Web-приложения с широкими функциональными возможностями. Именно в таких приложениях нуждается современный рынок электронной коммерции.

Для разработки современных Web-приложений можно использовать не только C++Builder. На рынке программных продуктов имеются и другие средства с аналогичными возможностями, например ***ASP*** или ***Perl***. Каждый продукт имеет достоинства и недостатки, и какой из них выбрать для разработки конкретного приложения — одна из наиболее трудных проблем среди тех, что приходится решать разработчику. Не менее важное значение имеет и метод, который будет выбран для обеспечения надежного хранения информации, накопленной Web-приложением, и защиты ее от несанкционированного доступа.

Выше уже шла речь о том, что протокол Secure Socket Layer (SSL) помогает обеспечить защиту информации в процессе ее пересылки по сетям от броузера к Web-приложению. Но ответственность за сохранение полученной информации возлагается уже на Web-приложение. К сожалению, многие разработчики не уделяют этому вопросу должного внимания, а это приводит к тому, что компьютерным взломщикам иногда удается добраться до весьма важной и секретной коммерческой и иного рода информации. Похищение свыше 300000 номеров кредитных карточек с Web-сайта CD Universe в 1999 году показывает, насколько необходимо обеспечить защиту хранящейся на сервере информации.

О чем следует задуматься

> То, что ты сам не параноик, отнюдь не означает, что их нет вокруг.
>
> *Неизвестный*

Web-приложения и базы данных, к которым эти приложения обращаются, функционируют в довольно сложных условиях. Как правило, пользователям должна быть обеспечена возможность свободного доступа к Web-серверу, на котором выполняется такое приложение. Этот означает, что Web-сервер открыт для атак со стороны любого, кто имеет доступ к Internet. Хотя администраторы и стараются обеспечить безопасность своих Web-серверов, хакеры умудряются получить неавторизованный доступ и повредить программы или похитить данные. Частично причины некоторых проблем безопасности лежат внутри службы администрирования — например, выбор недостаточно надежных паролей и средств их защиты (особенно FTP-паролей) значительно повышает вероятность несанкционированного доступа к информации, хранящейся в системе.

Когда речь идет о мерах обеспечения безопасности в Web-приложении, то разработчик не должен льстить себя надеждой, что хакеру не удастся добраться до содержимого сервера — выполняемого файла приложения, базы данных, файлов исходного кода, паролей и т.п.

А добравшись до этих файлов хакер с помощью доступных практически каждому программисту утилит извлечет всю необходимую ему информацию На рис. 13.17 показано, как выглядит в окне программы ***HackMan*** содержимое модуля DLL приложения ***WebBroker***. Как видите, при желании можно без труда извлечь из этого файла и имена пользователей и их пароли (если, конечно, они в этом файле присутствуют). Не спасает положение и использование технологии динамического формирования паролей — метод генерации паролей “вскрывается” с помощью декомпилирования программного кода. Существуют и утилиты для раскрытия паролей и имен пользователей, которым разрешен доступ к базам данных.

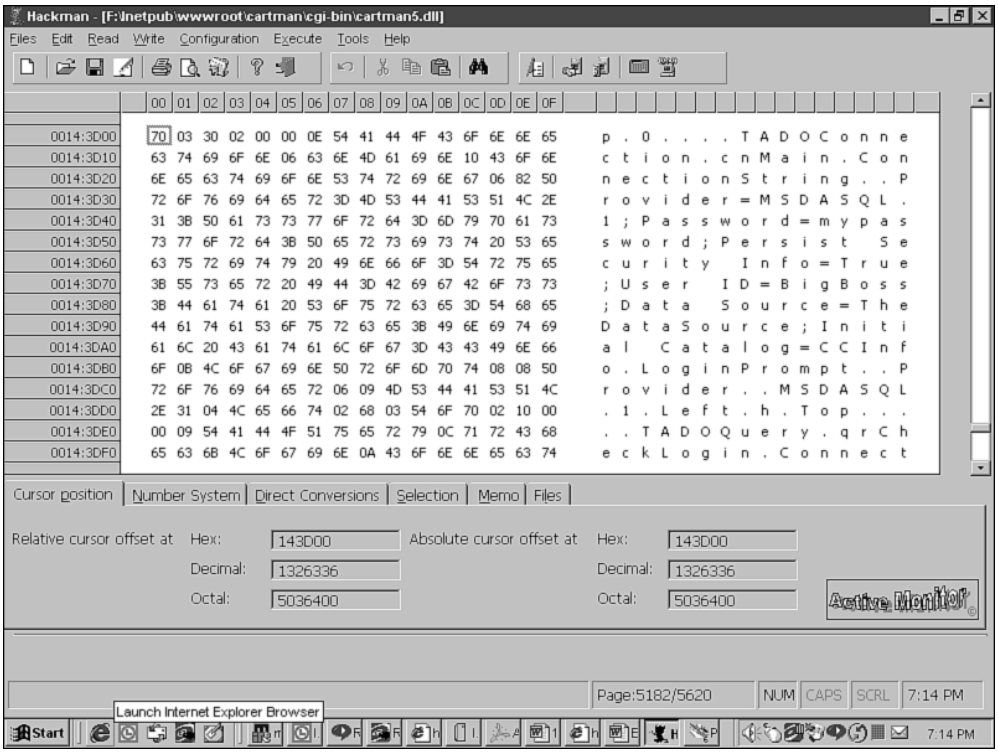


Рис. 13.17. Редактор шестнадцатеричного кода **HackMan** *(публикуется с разрешения TechnoLogismiki)*

Криптография

Мы не собираемся обсуждать в этом разделе проблемы криптографии как науки. Нас интересуют только вопросы эффективного использования криптографических методов в разрабатываемых Web-приложениях. Хотя математические основы криптографии вряд ли могут быть понятны рядовому программисту, но научиться использовать средства их реализации, уже разработанные другими, под силу практически всем.

В большинстве Web-приложений, предназначенных для обработки и сохранения конфиденциальной информации, в том или ином виде всегда использовались криптографические методы. Другое дело, что для их применения зачастую выбирали неверную методику. Если в цепочке криптографических средств имеется слабое звено, то и вся система защиты не может быть более надежной, чем это звено.

Разработчики зачастую выбирают какой-либо один, довольно мощный криптографический алгоритм и включают его в приложения, наивно полагая, что этот алгоритм как волшебный щит защитит информацию от “нескромного взгляда”. При этом забывают, что пароль, который используется для шифрования, хранится в самой программе или в ней используется довольно предсказуемая формула для динамической генерации паролей (например, слияние идентификатора заказа и дата отгрузки заказа). Стоит добраться до такого пароля и все усилия, вложенные в создание алгоритма, пойдут насмарку. В результате такое Web-приложение попадает на первые страницы газет, после того как обнаружится, что хакерам удалось “увести” несколько тысяч паролей или номеров кредитных карточек из этого приложения.

Эффективность криптографии определяется комбинированием алгоритмов и одностороннего хеширования, которые выполняются поочередно в соответствии с протоколом. Только протокол, реализованный надлежащим образом, может обеспечить неприкосновенность охраняемой информации. В следующем разделе мы кратко рассмотрим базовые компоненты криптографии, а потом обсудим, как использовать такой протокол для защиты информации в виртуальном магазине.

Протоколы, алгоритмы и однонаправленное хеширование

Протокол представляет собой перечень процедур, которые используются в Web-приложении для защиты и извлечения информации. Некоторые из предусмотренных протоколом процедур предусматривают использование в том или ином виде криптографических методов. Если в протоколе будет хотя бы одно слабое звено, это сведет на нет весь эффект использования в нем самых изощренных алгоритмов.

Под алгоритмом в данном случае мы понимаем некую математическую формулу, с помощью которой выполняется шифрование и расшифровка данных. По самой своей природе алгоритмы шифрования/расшифровки делятся на две больших категории — симметричные и асимметричные. Симметричный алгоритм использует для шифрования и расшифровки один и тот же ключ. Асимметричные алгоритмы используют два ключа — один, общедоступный для шифрования и другой, личный для расшифровки. Примерами симметричных алгоритмов могут служить ***DES***, ***Blowfish***, ***Twofish*** и ***RC2***. К асимметричным относятся ***RSA***, ***Rabin*** и ***Knapsack***. Скрупулезный анализ этих алгоритмов показал, что взломать зашифрованные с их помощью данные очень трудно. Атака на результаты работы этих алгоритмов может увенчаться успехом только в том случае, если при их реализации в приложении (а точнее, при организации каких-либо других операций, связанных с их использованием) была допущена ошибка или для взлома требуются огромные вычислительные ресурсы (например, совместная работа всех компьютеров в мире в течение нескольких лет). Ошибки, которыми могут воспользоваться хакеры, принадлежат к числу курьезов — например, программа не обеспечила очистку памяти, в которой хранился ключ шифрования, удаление временных файлов с промежуточными результатами, некорректная интерпретация алгоритма в процессе написания программного кода и т.п.

Функции хеширования принимают в качестве входной информации строки произвольной длины, а в качестве результата возвращают строку фиксированной длины. Для многих хеш-функций несложно предсказать, какой получится результат при определенных данных на входе или какие входные данные привели к определенному результату. Однако однонаправленные хеш-функции — это совершенно особый случай. Они и называются однонаправленными именно потому, что, зная результат, практически невозможно угадать, какой была входная последовательность данных, хотя, зная входную строку, можно предсказать, каков будет результат выполнения функции. Изменение хотя бы одного бита во входной последовательности приводит к радикальному изменению результата однонаправленного хеширования. В прикладной криптографии однонаправленное хеширование используется достаточно широко. В частности, с его помощью формируется уникальный ключ “сеанса” для симметричного алгоритма шифрования. Примером алгоритмов однонаправленного хеширования могут служить ***SHA***, ***MD4*** и ***MD5***.

Обеспечение безопасности виртуального магазина

Для демонстрации возможностей защиты информации рассмотрим, как это организовать в достаточно простом Web-приложении, обеспечивающем работу виртуального магазина. Защите будут подлежать сохраняемые данные о поступивших заказах. Будем полагать, что доступ к базе данных приложения, исходному коду и паролям открыт (или предположим, если вам нравятся детективные сюжеты, что хакерам удалось до них добраться).

Инструментальные средства

Для защиты информации будем использовать гибридную криптосистему, в которой сочетается скорость симметричного алгоритма шифрования с надежностью асимметричного. В качестве симметричного алгоритма выберем ***Blowfish***, в качестве асимметричного — ***RSA***, а в качестве функции однонаправленного хеширования — ***SHA***.

Некоторые алгоритмы шифрования запатентованы. При этом иногда обладатель патента не разрешает использовать алгоритм без оформления лицензионного соглашения. Нужно принимать во внимание и тот факт, что использование определенных алгоритмов в экспортируемом программном обеспечении ограничивается существующим законодательством.

Шаг 1. Формирование общедоступных и личных ключей

Выше в этой главе мы уже говорили о том, что при асимметричном шифровании обычно используется пара ключей — общедоступный и личный. Сам алгоритм формирует такую пару ключей, используя для этого случайные числа. После того как пара ключей сформирована, общедоступный ключ можно "положить на хранение" в базу данных. Личный ключ должен храниться иначе, предпочтительнее не на том компьютере, на котором выполняется приложение. В принципе набор ключей можно использовать бесконечно долго, но лучше все-таки их время от времени обновлять.

Шаг 2. Начальная фаза формирования заказа

Для шифрования данных, содержащихся в заказе клиента, мы будем использовать симметричный алгоритм ***Blowfish***, поскольку это сулит более высокую скорость обработки заказов. Но сначала нужно сформировать ключ шифрования текущего сеанса, которым будет пользоваться алгоритм ***Blowfish***. Для этого воспользуемся функцией хеширования ***SHA***. Надежность хеширования непосредственно зависит от степени случайности той информации, которая передается на вход функции хеширования. Если, например, хешировать показания системных часов, то для взлома хакер может воспользоваться данными, хешированными в прошлом месяце и попытаться расшифровать их используя ключи, полученные хешированием показаний часов. Более надежный результат может быть получен, если на вход функции хеширования подавать системное время, показания счетчика тактовых импульсов процессора, данные о текущем объеме занятой памяти в системе и другие переменные, значение которых уникально для того момента времени, когда формируется хеш.

Шаг 3. Шифрование заказа

После того как информация зашифрована, нужно подумать над способом ее хранения. В некоторых приложениях зашифрованное имя пользователя сохраняется в поле `Customer Name`, адрес — в поле `Address` и т.д. Ненадежность этого метода в том, что он дает хакеру сведения о характере информации, хранящейся в определенном поле, хотя и в зашифрованном виде. Информация определенной категории не является совершенно случайной. Например, номера всех кредитных карточек Discover начинаются с 6011 или 1800, а один из основных методов взлома кодов как раз и основан на анализе повторяющихся компонентов.

Более эффективным является использование режима ***Cipher Block Chaining*** (CBC). Этот режим предполагает формирование одной длинной строки, в которую включается вся зашифрованная информация. Для разделения результата расшифровки такой длинной строки на отдельные поля можно использовать компонент, производный от `TDataSetPageProducer`.

Проще всего для формирования длинной строки воспользоваться перед шифрованием объектом класса `TStringList`, который позволит включить в строку имена полей и их значения.

Шаг 4. Шифрование ключа

После того как информация о заказе зашифрована, нужно позаботиться о сохранении уникального ключа текущего сеанса шифрования этого заказа, без которого ее не удастся расшифровать. Для шифрования ключа сеанса будем использовать асимметричный алгоритм ***RSA***. Для шифрования этот алгоритм использует псевдослучайные числа и общедоступный ключ, который был сформирован на шаге 1. Расшифровка ключа текущего сеанса производится с помощью личного ключа, также сформированного на шаге 1 и сохраненного на другом компьютере.

Чтобы не оставить хакеру ни малейшего шанса, нужно тщательно "убрать за собой" после выполнения всех процедур, предусмотренных протоколом. В процессе их выполнения создается множество временных переменных в памяти, в которых хранится разнообразная информация, представляющая интерес для хакеров — ключи, фрагменты незашифрованных данных, промежуточные результаты и т.п. Рекомендую поверх всей этой информации записать всякий "мусор", а потом его стереть.

Извлечение информации о заказе

Поскольку Web-приложение располагает подходящим личным ключом, то не составляет труда расшифровать с его помощью ключ того сеанса работы, когда была сохранена информация о заказе, а затем с помощью этого ключа расшифровать саму запись. Полученный видимый текст можно дальше обрабатывать любым из предусмотренных в приложении способом.

Атака!

Давно стало аксиомой утверждение, что любой шифр в конце концов можно взломать. Достоинство описанного выше крипто-протокола не в том, что его нельзя в принципе взломать, а в том, что на это потребуется очень много сил (вычислительных ресурсов) и времени. Если строго следовать предписаниям протокола, то хакеру остается только одно — использовать "грубую силу", т.е. пытаться подобрать ключи перебором вариантов. Такой перебор для взлома шифра одной записи в базе данных может занять несколько месяцев напряженной работы. Не меньше времени уйдет и на подбор значений пары общедоступный/личный ключ, а ведь нужно еще вскрыть и ключ сеанса шифрования учетной записи. Если к тому же первая пара ключей время от времени меняется, то у хакера уйдет не менее года на то, чтобы добраться до вожделенной информации.

Вернемся вновь к аналогии с цепочкой и проанализируем надежность отдельных процедур, предусмотренных крипто-протоколом.[1] Первое звено в этой цепи — формирование пары общедоступный/личный ключ, для чего я рекомендовал использовать алгоритм ***RSA***. Личный ключ должен быть после этого сохранен в надежном месте, куда хакер добраться не сможет, и конечно же, его ни в коем случае нельзя хранить на том же компьютере, который используется в качестве Web-сервера. Второе звено в нашей цепи — формирование ключа сеанса работы, который затем будет использован алгоритмом ***Blowfish***. Если для этого используется несколько уникальных значений переменных, которые затем обрабатываются формулой однонаправленного хеширования SHA, то подобрать значение ключа хакеру будет очень трудно.

Информация, которую мы хотим скрыть от нескромного взгляда, — данные о заказе клиента — шифруется алгоритмом ***Blowfish*** с помощью созданного перед этим ключа текущего сеанса, и эта процедура — третье звено цепи. Этот алгоритм на сегодняшний день считается одним из самых надежных, и, как показывает детальный анализ, созданный им шифр очень

---

[1] *В данном случае аналогия с цепью, состоящей из нескольких звеньев, не совсем уместна. Скорее напрашивается аналогия с последовательно установленными препятствиями. Только преодолев все препятствия хакер сможет добраться до "охраняемого объекта".* — Прим. ред.

трудно или вообще невозможно взломать. Четвертым звеном цепи является блочный метод шифрования ***Cipher Block Chaining*** (CBC). Объединяя блоки шифрованной информации в единую длинную строку, CBC не даст хакеру ни малейшего шанса извлечь какую-либо пользу из имен полей и структуры базы данных, содержимое которой шифруется. Пятое, последнее звено цепи — шифрование с помощью алгоритма ***RSA*** ключа текущего сеанса, который был использован алгоритмом ***Blowfish*** при шифровании конфиденциальной информации о заказе. Алгоритм ***RSA*** также считается одним из самых надежных из существующих на сегодняшний день. Итак, из всего сказанного можно сделать вывод, что реализация этого протокола в Web-приложении практически исключает возможность завладения охраняемой информацией для хакера, какими бы мощными ресурсами он не обладал.

Заключение

В этом разделе я обратил ваше внимание на необходимость включения в Web-приложение специальных средств защиты конфиденциальной информации. Вы узнали, как это реализовать на практике и какие криптографические средства можно использовать для защиты от хакеров. Конечно, протокол защиты, описанный в этом разделе, несовершенен — в нем не предусмотрена защита некоторых видов атак, например от похищения ключей или их подмены.

HTML и XML

Одна из новинок C++Builder 5 Enterprise — поддержка языка Extensible Markup Language (XML), для чего в состав среды разработки включена инструментальная программа ***InternetExpress***. В этом разделе будет описана методика работы с ***InternetExpress*** и использование с ее помощью XML в качестве языка общения между приложениями.

Язык XML

Язык XML представляет собой подмножество Standard Generalized Markup Language (SGML), а HTML — это одно из приложений SGML. Отличие между XML и HTML в синтаксических правилах, которые существуют в HTML. XML-документ должен быть сформирован в соответствии с обобщенными синтаксическими правилами XML, но назначение ключевых слов (тегов) в нем отдано на откуп автору документа. Использованные ключевые слова должны быть зафиксированы в специальном разделе документа, названном *Document Type Definition* (DTD), или отдельном DTD-файле, прилагаемом к XML-документу. Говорят, что XML-документ, включающий DTD, является “сам себя описывающим”, а это открывает широкие возможности для включения в него самой разнообразной информации. Итак, XML-документ является просто ASCII-файлом, который может читать кто угодно, но смысл этого документа станет понятен только после использования описаний тегов, включенных в DTD-файл.

В чем преимущество языка XML? Во-первых в том, что XML-файл имеет простой формат, читать который можно с помощью любого текстового редактора. Учитывая, что синтаксис конкретного XML-документа описан в специальном DTD-файле, в документ можно включить любую информацию — от книги, представляемой на Web-странице, до базы данных и вложения целиком операционной или файловой системы. И если уж можно вложить в XML-документ все, что угодно, то его можно использовать в качестве средства пересылки информации при общении двух приложений (или отдельных компонентов одного и того же приложения).

Я думаю, что есть определенная аналогия между ролью XML/DTD как универсального языка обмена информацией в мире компьютеров и ролью языка эсперанто в человеческом обществе. Поддержка XML в Web-приложении весьма желательна, особенно с учетом потребностей открытого сообщества в Internet. Но, с другой стороны, я очень сомневаюсь в том, что со временем XML вытеснит все прочие языки. Через пару лет большинство инструментальных и прикладных программ будет в той или иной мере поддерживать работу с XML, точно так же, как сегодня поддерживается работа HTML. Но это же не значит, что все в Internet свелось к HTML, так же как не свелось все программирование к языку Java, как предрекали отчаянные головы несколько лет назад.

С помощью XML можно структурировать данные и информацию, которая передается от одного приложения другому, в том числе и между уровнями. Именно такая технология поддерживается инструментальной программой ***InternetExpress***, которая входит в комплект C++Builder 5. При этом предлагается использовать стандартный формат, не зависящий от используемого транспортного протокола. По этой причине ожидается, что XML будет играть важную роль в будущих приложениях типа Electronic Commerce EDI (Electronic Data Interchange). Тот факт, что для определения синтаксиса (а значит, и для синтаксического анализа) самого XML-документа можно использовать DTD, означает — по крайней мере теоретически, — что, подобрав подходящий синтаксис в DTD, каждый сможет общаться с каждым и "говорить" о чем угодно. Не располагая DTD, ни одно приложение не сможет правильно интерпретировать содержимое XML-документа. Это доступно только приложению, которое его создало и имеет в своем распоряжении соответствующий файл DTD.

Лично я рассматриваю XML как переносимый формат файла, весьма удобный для организации обмена информацией между приложениями. ***InternetExpress*** позволяет использовать XML для работы серверного MIDAS-приложения (`DataSetProvider`) с клиентом, в данном случае — Web-сервером (`XMLProvider`, который может обмениваться информацией только с ***WebBroker-***компонентом `MidasPageProducer`). (MIDAS — это аббревиатура от *Multi-tier Distributed Application Services Suite — набор сервисных программ для создания многоуровневых приложений*.) XML-сообщения, которые передаются в обе стороны при общении серверного приложения и Web-сервера, фактически содержат пакеты данных MIDAS. В ранних версиях MIDAS такие пакеты данных имели соответствующий файловый формат. Теперь ими стали XML-документы, а это означает, что структура пакета стала более открытой. Правда, пока что все это только теоретически, поскольку пакет данных MIDAS поступает к получателю без DTD. Сейчас разобраться в содержимом такого пакета смогут только приложения, написанные в среде Delphi 5 или C++Builder 5.

Другими словами, хотя MIDAS-клиенты и серверы, созданные в среде C++Builder 5, и могут теперь передавать пакеты в виде XML-документов, не это является главным достоинством XML. Я по-прежнему не могу подключить клиента PowerBuilder к `DataSetProvider` или заставить Oracle8i формировать XML-документ, с которым сможет работать компонент `XMLBroker`. Даже в приложении, созданном полностью в среде Delphi 5, XML-документ, сформированный объектом `DataSetProvider`, скорее всего удастся использовать только в объекте `XMLBroker`, который, в свою очередь, сможет общаться только с объектом `MidasPageProducer`. Между `DataSetProvider` и "обычным" объектом `ClientDataSet` данные, как и прежде, нужно передавать в особом (к тому же и недокументированном) формате.

Те средства поддержки XML, которые имеются в ***InternetExpress***, — это только начало. С их помощью вы сможете создавать распределенные приложения, в которых обмен информацией между уровнями выполняется с помощью XML-документов. XML-формат можно будет также использовать при загрузке и сохранении MIDAS-пакетов данных. Все эти возможности сейчас можно реализовать только в MIDAS-приложениях, но ведь это только начало!

InternetExpress

Инструментальная программа ***InternetExpress*** базируется на технологиях MIDAS и ***WebBroker***. (Подробно о MIDAS-приложениях будет рассказано в главе 19.) Эта технология, разработанная фирмой Borland, обеспечивает совместную работу серверных и клиентских приложений, созданных с помощью Delphi, C++Builder и JBuilder. Помимо интеграции MIDAS с ***WebBroker***, ***InternetExpress*** поддерживает и работу с XML — новым многообещающим стандартом в Internet.

Обработка заказов

InternetExpress-клиент получает данные от MIDAS-сервера — в данном случае приложения `MidasServer`, — который экспортирует информацию о клиентах и сделанных ими заказах. Процесс создания этого приложения детально рассмотрен в главе 19, где показано, как подключить его к обычному MIDAS-клиенту. ***InternetExpress*** можно рассматривать как особый вариант MIDAS-клиента, а потому мы не будем в этом разделе останавливаться на детальном анализе работы MIDAS-сервера, а сосредоточим внимание на клиентской части приложения. При желании вы можете сначала прочесть главу 19, списать исходные файлы проекта `MidasServer` с прилагаемого компакт-диска и скомпилировать его, а уже потом приступить к изучению материала этого раздела.

Компонент `TMidasPageProducer`

Будем считать, что MidasServer готов к работе и к нему можно подключаться. Модифицируем существующее ***Web Module***-приложение `WebShow` и превратим его в MIDAS-клиент с помощью ***InternetExpress***. У нас остались неиспользованными два элемента Web-операций: `/browse` и `/final`. Их-то мы и используем в этом разделе.

Однако сначала нужно подключиться к приложению MIDAS-сервера и получить от него данные, которые планируются публиковать в сети Web. Поскольку используется удаленный модуль данных, для выполнения этих функций подходит компонент `TDCOMConnection` (он находится на вкладке MIDAS). Перенесите один из них на поле ***Web Module*** и откройте список имен удаленных объектов. Где-то в этом списке должно быть имя сервера `MidasServer.CustomerOrders`. Первый компонент имени — наименование проекта MIDAS-сервера (о нем мы поговорим в главе 19), а второй компонент имени — наименование *сопряженного* класса (*coclass*) удаленного модуля данных, который определен в проекте. Использование составного имени позволяет работать с MIDAS-сервером, в котором имеется несколько удаленных модулей данных. После того как задано значение свойства `ServerName`, можно активизировать и свойство `Connected`. Когда ему присваивается значение `true`, MIDAS-сервер начинает работать, и вы увидите его окно на своем рабочем столе. Это может служить подтверждением, что MIDAS-сервер найден и с ним организована связь. Остается извлечь данные из этого сервера и представить их на Web-странице. Эта операция выполняется двумя компонентами, размещенными на вкладке InternetExpress: `XMLBroker` и `MidasPageProducer`. Нужно включить в ***Web Module*** по одному объекту каждого их этих компонентов.

В структуре компонента `XMLBroker` имеется свойство `RemoteServer`, которому нужно присвоить имя объекта компонента `DCOMConnection`. Этим организуется подключение и предоставляется возможность развернуть список имен провайдеров и выбрать в нем объект

DataSetProvider удаленного сервера. В данном случае в этом списке есть только один объект — DataSetProviderCustomerOrders, – но в принципе выбранный удаленный модуль данных может иметь несколько объектов DataSetProvider. После установки значения свойства ProviderName можно переходить к следующему этапу — собственно разработке Web-страницы с использованием компонента MidasPageProducer.

Редактор Web-страниц

После щелчка на поле объекта MidasPageProducer правой кнопкой мыши откроется окно редактора Web-страниц. Этот редактор имеет два режима просмотра — режим броузера (он организуется элементом управления типа Internet Explorer) и режим HTML (в этом режиме можно просматривать сформированный HTML-код). В верхней части окна редактора имеются две панели. На поле левой панели представлены компоненты, которые включены в состав объекта MidasPageProducer, а на поле правой — показана структура (дочерние элементы) выбранного компонента.

Работая в окне этого редактора, можно создавать компоненты, хотя на экране палитра компонентов и не присутствует. После следующего щелчка правой кнопкой мыши открывается диалоговое окно New Component, в котором представлены компоненты, которые могут быть включены в выбранный объект, т.е. в эго иерархию, представленную в левой панели. Например, когда мы начинаем работу в окне редактора Web-страниц, то выбирать предлагается из множества DataForm, QueryForm и LayoutGroup. После того как выбран объект DataForm и на его поле выполнен щелчок правой кнопкой мыши, в окне New Component будут представлены DataGrid, DataNavigator, FieldGroup и LayoutGroup. Выбор LayoutGroup не внесет никакого нового объекта в страницу, а позволит изменить компоновку существующих (например, установить FieldGroup слева от DataGrid). На практике я никогда не пользуюсь LayoutGroup, поскольку имеются другие средства размещения элементов на поле Web-страницы, информацией о которых я с вами поделюсь. А сейчас выберите компоненты DataNavigator и FieldGroup.

После того как вы вставите в формируемую Web-страницу эти два объекта, на вкладке Browser редактора чуть выше изображений объектов появятся два предупреждающих сообщения. То, которое выведено около объекта DataNavigator, предупреждает, что свойство XMLComponent имеет значение nil. Второе сообщение появляется возле объекта FieldGroup, и в нем содержится предупреждение о том, что свойство XMLBroker имеет значение nil. Первую ошибку устранить довольно просто — выберите объект DataNavigator на поле правой панели и свяжите его свойство XMLComponent с объектом FieldGroup (т.е. этот навигатор должен работать с объектом FieldGroup). После этого выберите объект FieldGroup и свяжите его свойство XMLBroker с объектом XMLBroker1 (который входит в состав модуля ***Web Module***). Это приведет к тому, что редактор снимет второе предупреждающее сообщение и выведет список всех полей таблицы Customer на поле правой панели, причем в списке будет присутствовать и специальное поле FieldStatus1. Это поле содержит односимвольное значение статуса текущей записи, которое является индикатором того, что запись вставлена, модифицирована или удалена. Это поле несет полезную информацию для разработчика, но, я думаю, представлять ее конечному пользователю Web-страницы вряд ли целесообразно. По умолчанию в список включаются все поля выбранной главной таблицы (в данном случае, Customer), но разработчик может и явно выбрать необходимые поля — для этого следует щелкнуть на FieldGroup правой кнопкой мыши и выбирать только нужные поля (рис. 13.18).

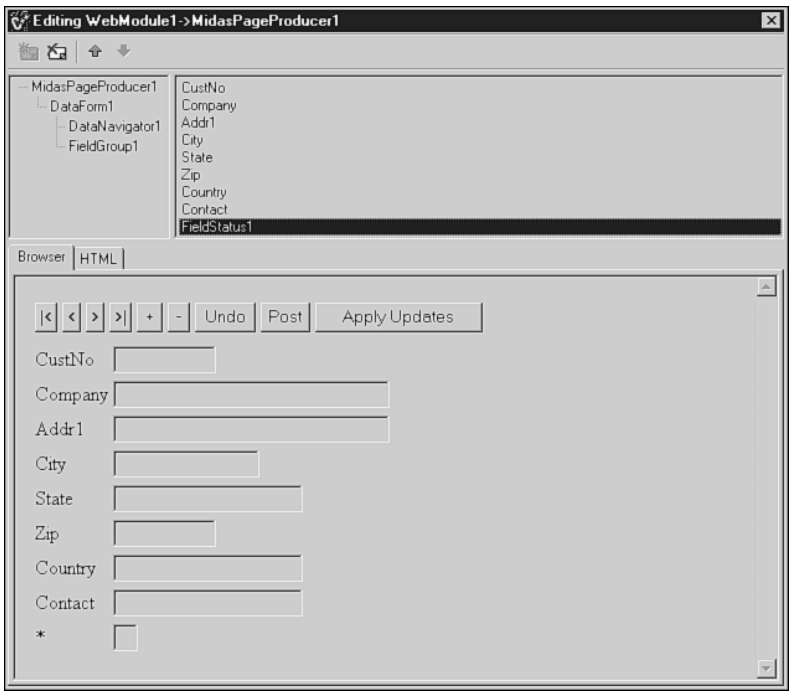


Рис. 13.18. Окно редактора Web-страницы в процессе разработки

Запуск на выполнение

Прежде чем запустить клиентское приложение ***InternetExpress***, нужно проверить, выполнены ли необходимые подготовительные операции. Во-первых, создаваемое Web-серверное приложение (в данном случае — `WebShow.dll`) должно быть помещено в каталог сценариев на том компьютере, на котором работает Web-сервер. Проще всего для этого настроить параметры проекта таким образом, чтобы в качестве выходного каталога проекта фигурировал каталог со сценариями на локальном диске вашего компьютера. Тогда в этом каталоге всегда будет находиться самая последняя версия приложения.

IIS кэширует выполняющиеся модули ISAPI DLL, но это не всегда срабатывает, поскольку DLL на диске может быть заблокирован. Это еще один довод в пользу того, чтобы для отладки модулей ISAPI DLL использовать программу ***IntraBob***.

Вторая подготовительная операция — настройка свойства `Producer` того объекта `WebActionItem`, который "отвечает" за операцию `/browse`. Этому свойству должна быть присвоена ссылка на объект `MidasPageProducer` нашего ***Web Module***-приложения.

Нужно также проверить, может ли сформированный приложением выход получить доступ к набору файлов JavaScript, которые необходимы для грамматического разбора формируемых пакетов XML (эти пакеты будут внедрены в сформированный HTML-код Web-страницы). Исходные файлы JavaScript поставляются в комплекте с C++Builder 5 Enterprise и находятся в каталоге `CBuilder5\Source\WebMidas`. Там вы увидите пять файлов общим объемом около 60 Кбайт. Проще всего обеспечить доступ к набору файлов JavaScript следующим образом: скопируйте эти файлы в определенный каталог на

Web-сервере и укажите этот каталог в свойстве `IncludePathURL` объекта `MidasPageProducer`. У меня на компьютере это `http://localhost/scripts` – тот же каталог, в котором я держу выполняемые файлы CGI-приложений и модули ISAPI DLL. Если вы будете пользоваться программой ***IntraBob***, то можете просто указать `./` в качестве значения `IncludePathURL` и скопировать файлы JavaScript в тот же каталог, в котором находятся модули ISAPI DLL и файл `IntraBob.exe`.

Удостоверившись, что все подготовительные операции выполнены, можно еще раз повторить полную компиляцию проекта `WebShow` и запустить на выполнение созданные модули ISAPI DLL с помощью программ ***IntraBob*** v 5.0, Internet Explorer версии 4+ или Netscape Communicator версии 4+. Результат работы программы представлен на рис. 13.19.

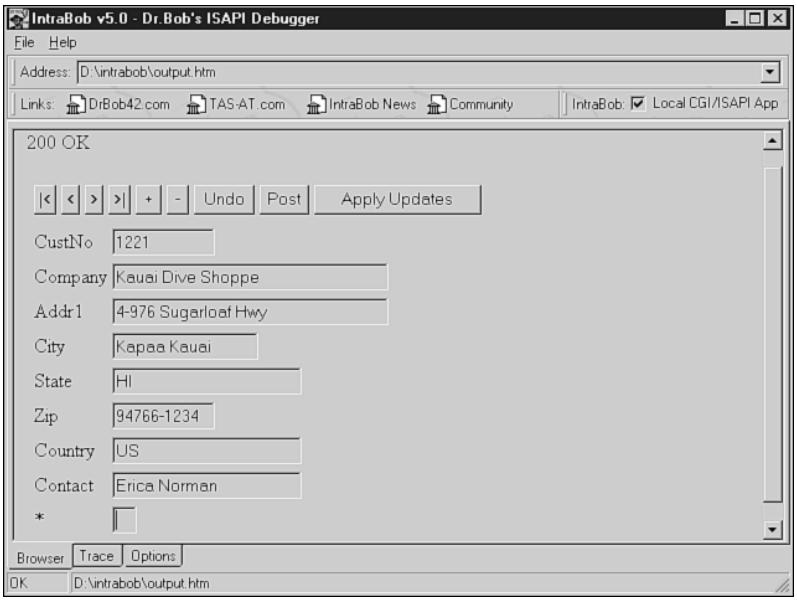


Рис. 13.19. Результат работы приложения, созданного с помощью **InternetExpress***, в окне программы* **IntraBob**

Новая версия просмотра данных в режиме „главный–подчиненный“

Последний пример, который будет рассмотрен в этой главе, включает возможность просмотра подробной информации о клиентах в режиме “главный/подчиненный”, причем главное окно приложения располагает средствами навигации по записям таблицы. Откройте опять окно редактора Web-страниц (щелкните на `MidasPageProducer` правой кнопкой мыши) и добавьте в `DataForm` объект `DataGrid` и еще один объект `DataNavigator`. Расположите `DataNavigator` выше `DataGrid` и проверьте, чтобы `DataGrid` был объектом `XMLComponent` по отношению к `DataNavigator`. На сей раз опять появится предупреждение о том, что в `DataGrid` значение свойства `XMLBroker` не определено (равно `nil`). Свяжите свойство `XMLBroker` объекта `DataGrid` с объектом

XMLBroker1 (это единственный доступный объект компонента XMLBroker). После этого предупреждение исчезнет с экрана, но результат все же будет не таким, как задумано: в сетке DataGrid будут выведены заголовки (имена полей) главной таблицы Customer, а не подчиненной таблицы Orders. Чтобы устранить эту несуразицу, откройте свойство XMLDataSetField этого объекта и установите в нем ссылку на таблицу TableOrders. Только после этого в сетке DataGrid появятся нужные нам заголовки — имена полей подчиненной таблицы. Естественно, не стоит выводить в экранной форме все поля этой таблицы, отберите только те, которые представляют интерес для конечного пользователя. Оставьте в числе отобранных полей и StatusColumn1 — это поле несет информацию о том, что данная запись изменена в процессе работы с броузером.

По умолчанию ширина колонок сетки DataGrid устанавливается соответственно длине имен полей таблицы, и, как видно на рис. 13.20, колонки могут оказаться довольно широкими. Настройка ширины колонок выполняется с помощью свойства DisplayWidth каждого отдельного объекта колонки. Скомпонуйте сетку таким образом, чтобы все колонки имели одинаковую ширину в 12 символов. Для этого по очереди выбирайте колонки и присваивайте свойству DisplayWidth значение 12. Можно изменить и заголовки некоторых колонок (свойство Caption), например вместо PaymentMethod (способ оплаты) установите более короткое Payment (оплата).

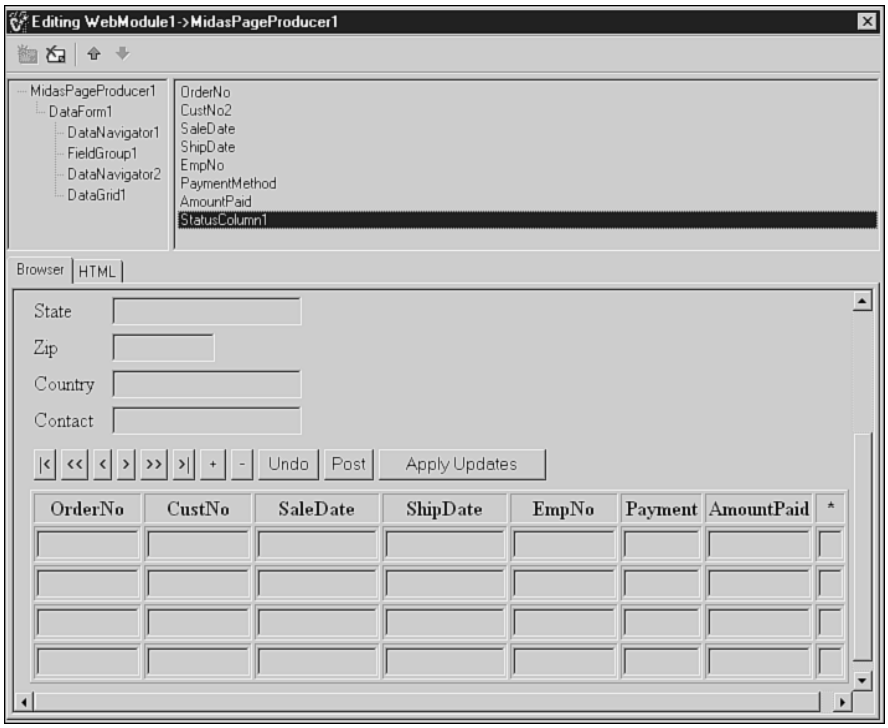


Рис. 13.20. Компоновка представления главной и подчиненной таблиц в Web-странице

Хотя работа над этой Web-страницей и не завершена — мы скомпоновали только функциональные элементы и не занимались оформлением (цвет, графика, шрифты

и т.п.), — она вполне пригодна для тестирования. Думаю, вы со мной согласитесь: сначала нужно обеспечить, чтобы продукт работал, как следует, а уж потом заботиться об украшательстве.

На этот раз мы можем воспользоваться операцией `/final` и вызывать соответствующий объект `WebActionItem`. Результат работы `MidasPageProducer` показан на рис. 13.21.

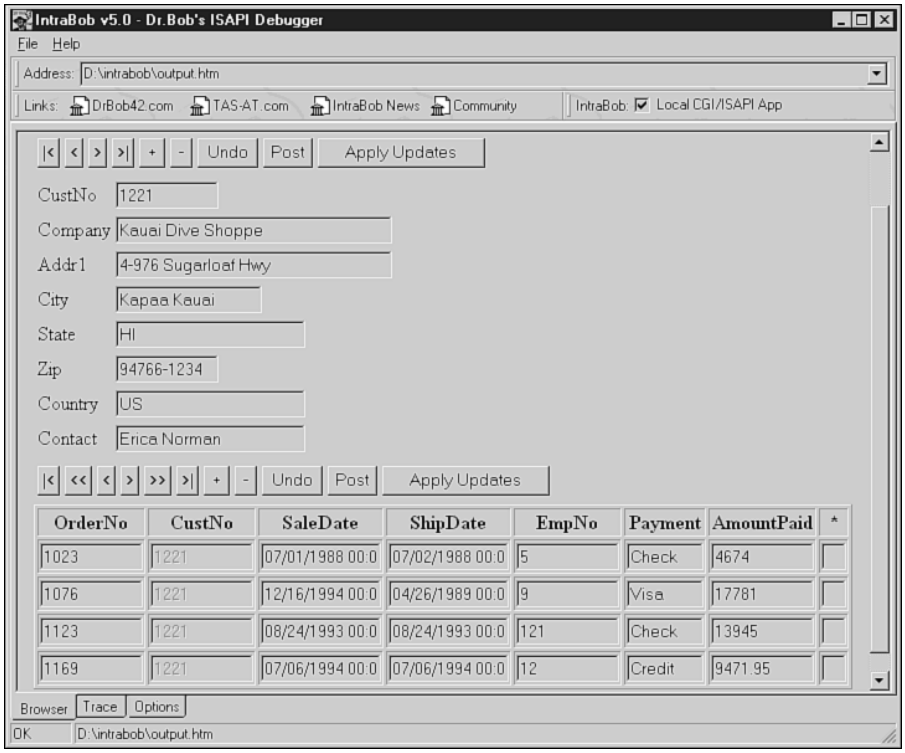


Рис. 13.21. Web-страница с данными из связанных таблиц в окне программы **IntraBob**

На этой странице представлено все, что мы собирались показать конечному пользователю. В ее состав входит HTML-код вывода таблицы, XML-код, который содержит определения полей главной и подчиненной таблиц (вместе с соответствующими данными из этих таблиц) и JavaScript для грамматического разбора XML-данных и связывания элементов данных с элементами управления, которые представлены в Web-странице. Итак, должен вас порадовать (а некоторых, возможно, и огорчить) — это все, больше ничего делать не нужно!

Что нас радует в этой странице? Когда вы будете поочередно просматривать записи таблицы, то никакого мелькания на экране видно не будет. HTML-код остается все время загруженным, а изменяются только данные в элементах управления. Это обеспечивается кодом JavaScript, который работает с XML-данными. Когда с таким эффектом встречаешься в первый раз, то он производит довольно сильное впечатление.

А что не радует? При переходе от одной записи к другой явно видна задержка обновления полей. Поскольку при каждой операции перехода набор данных из главной и подчиненной таблиц пересылается по каналу связи, задержка, особенно при работе с таблицей заказов `Order` большого объема, может быть весьма заметной. Уменьшить задержку

можно, ограничив количество записей подчиненной таблицы, пересылаемых на сервер. Для этого нужно установить соответствующее значение в свойстве MaxRecords объекта XMLBroker — если там установлено значение X, то будут пересланы только первые X записей подчиненной таблицы.

Возможно, вы заметили, что поле CustNo в сетке DataGrid выглядит не так, как остальные. Это объясняется тем, что это поле имеет статус "только для чтения" и данные в нем не могут быть изменены пользователем. Другие же поля можно редактировать, причем при внесении изменений соответствующая запись помечается маркером M (*modified* — модифицировано) в поле Status.

Обратите внимание на то, что на рис. 13.21 поле Status отсутствует, поскольку я удалил его из списка видимых полей.

Учтите также, что любое изменение, выполненное в записях таблиц при работе с броузером, остается до поры до времени только в броузере. Переслать изменения на MIDAS-сервер можно, щелкнув на кнопке Apply Updates. Только после этого изменения помещаются в XML-пакет и пересылаются на сервер, а уже сервер вносит эти изменения в базу данных и при необходимости передает на броузер сообщение об обнаруженной ошибке. (Тем, кто пожелает детально познакомиться с видами ошибок, которые характерны для многопользовательского режима работы с базой данных, я рекомендую познакомиться с материалами в каталоге DEMOS\MIDAS\INTERNETEXPRESS\INETXCUSTOM. Там вы найдете два пакета, среди прочего содержащих и компонент ReconcilePageProducer, который можно подключить к компоненту XMLBroker для обработки таких ошибок. И конечно же, нужно внимательно изучить материал о MIDAS-клиентах и серверах в главе 19).

Оформление Web-страниц

Та Web-страница, которую мы спроектировали в предыдущем разделе, выглядит довольно невзрачно. Вряд ли ее дизайн привлечет внимание пользователей — некоторые называют подобный стиль оформления "казарменным". Существует множество способов придать Web-странице более привлекательный вид. Прежде всего измените значение свойства HTMLDoc объекта MidasPageProducer (но не трогайте специальных тегов, имена которых начинаются с #). Если желательно обеспечить гибкость компоновки Web-страницы, воспользуйтесь свойством HTMLFile, которое позволяет связать объект с внешним HTML-файлом, над которым может потрудиться кто-либо другой.

Кроме того, у вас всегда есть возможность "поиграть" с настройками внешнего вида отдельных полей в составе объекта FieldGroup или отдельных колонок сетки объекта DataGrid — задать шрифт, цвет и т.п. атрибуты. Все это можно сделать, используя встроенные стили, непосредственно в HTML-коде. В каталоге Examples/MIDAS/InternetExpress вы найдете множество примеров использования разнообразных дополнительных компонентов ***InternetExpress***.

Пакет Examples/MIDAS/InternetExpress/Inetxcustom/dclinetxcustom_bcb.bpk предназначен для создания и загрузки проекта Webshow.bpr (этот проект вы сможете отыскать на прилагаемом компакт-диске). Загрузите этот пакет, скомпилируйте его и установите на своем компьютере, прежде чем приступить к дальнейшему совершенствованию дизайна Web-страницы.

Этот пакет включает ImageNavigator и прочие компоненты, с которыми мы детально познакомимся в главе 19. Пустая экранная форма приложения ***InternetExpress*** на этапе проектирования должна после настройки внешнего вида выглядеть примерно так, как на рис. 13.22.

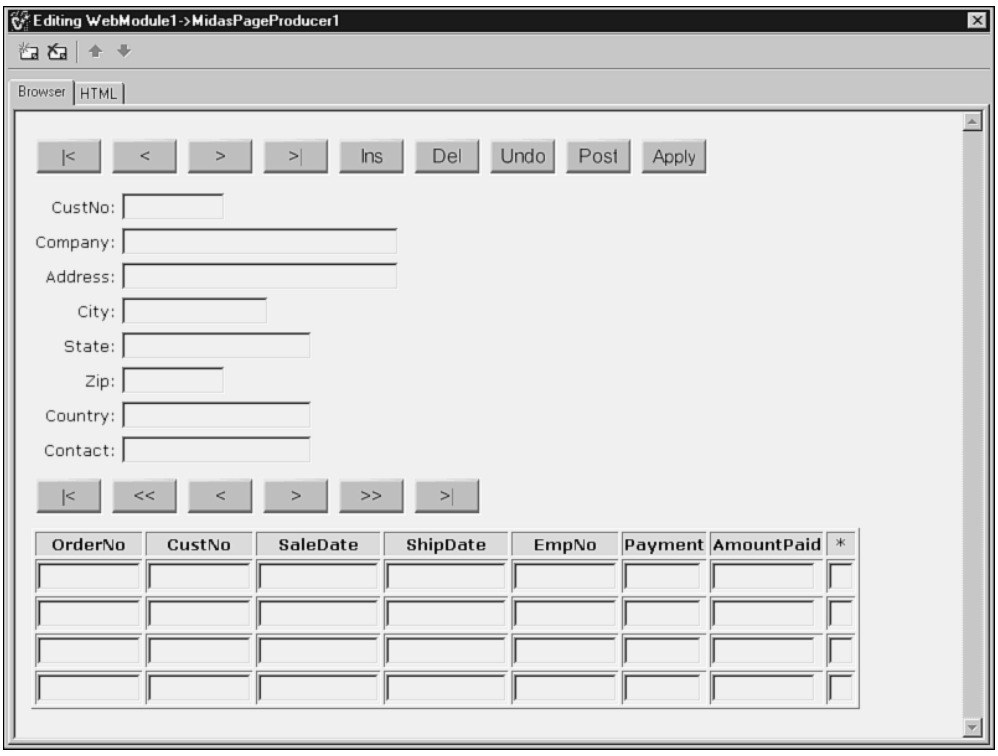


Рис. 13.22. Оформление Web-страницы для просмотра главной и подчиненной таблиц в окне программы **IntraBob**

После завершения настройки свойств (в том числе и свойства `HTMLDoc` объекта `MidasPageProducer`) можно сохранить полученный HTML-код в файле, имя которого установлено в свойстве `HTMLFile`. В принципе, в этом HTML-коде можно менять все, что угодно, за исключением специальных #-тегов.

Резюме

В этой главе вы познакомились с теми средствами, которые имеются в составе C++Builder для разработки Web-серверных приложений. Мы продемонстрировали вам, какие проблемы возникают в процессе создания таких приложений и предложили средства для их разрешения.

InternetExpress использует язык XML как для обмена информацией между приложениями (в частности, между приложениями MIDAS-сервера и Web-сервера), так и для формирования пакетов данных в Web-странице. Хотя для поддержки работы с большими наборами данных потребуются дополнительные средства, продемонстрированная методика позволяет быстро и без особых усилий публиковать данные в сети Web на основе технологии “тонкого” клиента.

Надеюсь, материал этой главы достаточно убедительно продемонстрировал достоинства технологий ***WebBroker*** и ***InternetExpress***, которые поддерживаются средой C++Builder 5.

Программирование БД-приложений

Глава

14

Марк Кэшмен
Вильям Моррисон
Стефан Махо

Среда C++Builder и библиотека VCL разрабатывались с таким расчетом, чтобы разработчик с их помощью мог быстро и без особых усилий создавать элегантные полнофункциональные приложения для работы с базами данных любого размера. В этой главе будут рассмотрены соответствующие инструментальные средства в составе C++Builder, в частности Borland Database Engine — ядро работы с базами данных фирмы Borland, модули данных и компоненты для работы с данными. Но начнем мы с изложения общих концепций организации приложений, работающих с базами данных (в дальнейшем будем именовать их *БД-приложениями*), и краткого описания основных элементов языка SQL (Structured Query Language), который используется для формирования запросов и обновления информации в базах данных.

Архитектура БД-приложений

Когда речь идет о создании БД-приложений, то вряд ли найдется другая среда программирования, которая могла бы соперничать с C++Builder. Поэтому нет ничего удивительного в том, что эта среда позволяет разработчику выбрать наиболее подходящую архитектуру создаваемого приложения из множества вариантов, которые поддерживаются в C++Builder. Даже в редакции C++Builder Professional имеются средства поддержки нескольких вариантов архитектур, а в редакции Enterprise их количество еще больше. В этом разделе мы представим читателю варианты архитектуры, которые наиболее часто используются в практике создания БД-приложений.

Хочу заранее предупредить, что в этом разделе мы будем затрагивать только технические характеристики различных вариантов, а не тонкости их реализации в рамках определенного проекта. Вне поля зрения останутся также требования к аппаратному обеспечению, материальные средства, необходимые для осуществления проекта, график выполнения работ, квалификация участников проекта и т.п.

Ядро Borland Database Engine

Ядро системы управления базами данных *Borland Database Engine* (BDE) является в архитектуре БД-приложения фундаментом, на который "опираются" компоненты библиотеки VCL. При создании этого ядра специалисты фирмы Inprise/Borland преследовали цель обеспечить использование самых разных форматов баз данных совместно с компонентами VCL. BDE поддерживает множество форматов, среди которых текстовый формат с разделителями (формат *ASCII-delimited text*), xBase, Paradox, формат реляционной модели данных (известный также как формат удаленных баз данных SQL) и формат ODBC (Open Database Connectivity). Обратите внимание на то, что использование ODBC фактически означает возможность использования и множества других форматов, (например, баз данных Access). Первые три формата обрабатываются непосредственно и являются, таким образом, "родными" для BDE. Для обработки данных в других форматах используются дополнительные слои в структуре приложения. Эти варианты мы также рассмотрим в последующих разделах этой главы.

BDE позволяет использовать множество различных форматов баз данных с одними и теми же компонентами VCL. Кроме того, BDE избавляет разработчика от необходимости организации транзакций, кэширования обновлений и поддержки XML. Таким образом, работая с BDE, программист может все внимание сосредоточить на том, *какие* данные нужно извлечь или записывать, а не на том, *как* это осуществить.

BDE также позволяет абстрагироваться от сложностей процесса подключения к базам, поскольку все компоненты VCL используют для ссылки на данные механизм псевдонимов.

Псевдоним (*alias*) представляет собой имя, выбранное разработчиком приложения или специалистом по установке приложения, которое с помощью административных средств BDE связывается с именем драйвера базы данных и набором всей сопутствующей информации, включая и информацию о реальном расположении базы данных. Правда, за такую услугу приходится платить определенную цену — программы поддержки BDE довольно "прожорливы" по отношению к памяти и пространству на диске. Но учтите, что все БД-приложения, созданные с помощью C++Builder и Delphi, совместно используют один и тот же набор программ BDE (если, конечно, они работают на одном компьютере).

Базовая BDE-архитектура (одноуровневая)

Этот вариант поддерживается в редакциях C++Builder Professional и Enterprise. За фирмой Borland уже давно закрепилась репутация лидера в разработке средств работы с локальными базами данных. В BDE фактически собрано все лучшее, что было создано специалистами Borland за последние годы в области инструментальных средств работы с базами данных самого разного формата. Базовыми для BDE являются формат текстового файла с разделителями (*ASCII-delimited text*), группа xBase-форматов (в нее входят форматы баз данных dBase, Clipper и FoxPro) и формат базы данных Paradox. Для небольших приложений, в которых решающими факторами является низкая цена и производительность, а защита от несанкционированного доступа не так существенна, этого набора вполне достаточно.

Преимущества

- Использование любого из перечисленных форматов позволяет добиться наивысшей скорости работы компонентов VCL.
- Все эти форматы доступны абсолютно свободно — за их использование не нужно никому платить.
- Форматы группы xBase очень распространены в приложениях, ориентированных на использование ПК, причем практически все такие приложения позволяют импортировать таблицы и базы данных в указанных форматах.
- Довольно легко переключаться в рамках такой архитектуры с одного формата на другой.
- Нет необходимости приобретать и изучать дополнительные компоненты (или библиотеки).

Недостатки

- Ни один из перечисленных форматов не поддерживает "серьезные" механизмы защиты от несанкционированного доступа и сохранения целостности многотабличной базы данных. Использование таких форматов в сетевой среде и приложении, рассчитанном на одновременную работу большинства пользователей, может породить множество проблем, связанных с надежностью, поскольку приложение каждого пользователя самостоятельно обновляет информацию непосредственно в базе данных.
- Программы BDE требуют больших ресурсов оперативной и дисковой памяти. В этом смысле, а также по части производительности BDE не сможет составить конкуренцию программам, специально настроенным на один из перечисленных форматов.

BDE/SQL Links (клиент/сервер)

Этот вариант поддерживается в редакции C++Builder Enterprise. При работе с реляционными базами данных (часто их называют SQL-базами данных) между BDE и собственно

базой данных размещается дополнительный уровень, который получил наименование *SQL Links* (соединения SQL). Его можно считать своего рода посредником между BDE API и API подключенной базы данных. В комплект редакции Enterprise входят средства поддержки SQL Links для баз данных Oracle, SQL Server, DB2 и InterBase. В комплект редакции Professional входит только поддержка SQL Links для InterBase. Такую архитектуру, согласно новейшей терминологии, принято называть двухуровневой (two-tier), и она является одной из наиболее широко распространенных в традиционных БД-приложениях. Эта архитектура обеспечивает достаточно широкие функциональные возможности приложения и обладает необходимой гибкостью, причем все это по весьма умеренной цене, особенно в сравнении с многоуровневой архитектурой.

Преимущества

- Все системы управления базами данных (СУБД), которые поддерживаются SQL Links, реализуют технологию клиент/сервер.
- Все СУБД, которые поддерживаются SQL Links, обладают полным спектром функциональных возможностей работы с реляционными базами данных.
- В такой архитектуре относительно просто переходить с одного формата базы данных на другой.
- Эта архитектура обеспечивает более высокую производительность, чем альтернативный вариант, предполагающий использование ODBC.

Недостатки

- Использование большинства СУБД, поддерживаемых в этой архитектуре, предполагает выплату определенных отчислений фирме-разработчику.
- Некоторые разработчики считают, что тот набор СУБД, который поддерживается SQL Links (напомню, в набор входят Oracle, Sybase, MS SQL Server, Informix, DB2, Access и InterBase), слишком узок.
- Для работы BDE требуется большой объем оперативной и дисковой памяти. Производительность приложения, построенного по этой архитектуре, значительно уступает приложению, специально сконфигурированному под определенный тип СУБД.

Распределенная (многоуровневая) архитектура

Этот вариант поддерживается в редакции C++Builder Enterprise. Распределенные БД-приложения в последнее время все чаще называются *многоуровневыми* (*multitier*). Смысл этого термина в том, что программа разделяется на две или несколько секций, которые размещаются на разных серверах. Взаимодействие этих секций программы организуется на основе разных методов (или протоколов), в частности CORBA, DCOM и HTTP. Аббревиатура CORBA означает Common Object Request Broker Architecture (Универсальная архитектура с использованием посредничества при запросе объектов). По самой свей сути спецификация CORBA является открытой и многоплатформенной, ее достаточно просто использовать, но средства поддержки этой архитектуры нужно приобретать отдельно, причем по весьма солидной цене. Архитектура DCOM разработана фирмой Microsoft и годится только для платформы Windows. Протокол HTTP так же, как и CORBA, является открытым и многоплатформенным, но значительно уступает по своим возможностям альтернативным решениям.

Реализация многоуровневой архитектуры требует привлечения очень сложных средств и применяется обычно только в очень больших проектах. В рамках многоуровневой архитектуры используется три модели приложений, которые кратко рассмотрены ниже.

Стандартная многоуровневая модель позволяет приложению использовать компоненты, размещенные на других серверах. Таким образом, несколько приложений могут совместно использовать программные компоненты. которые определяют логику бизнес-правил, а значит, во всех приложениях реализуется единая политика обработки информации.

Другой вариант ориентирован на применение в проектах, реализующих *модель тонкого клиента* (*thin-client*). Суть ее состоит в том, что на компьютере клиента размещается только программный компонент небольшого объема, обеспечивающий интерфейс с пользователем. Все остальные компоненты приложения, которые определяют логику обработки информации и бизнес-правила, размещаются на сервере. Такой вариант идеально подходит для сетей с низкой пропускной способностью (например, Internet), сетевых компьютеров, которые рассчитаны централизованную обработку данных, или на использование в системе множества операционных систем.

Примерно тот же набор компонентов требуется и для другой модели, получившей наименование *briefcase* (*портфель*). Эта модель позволяет клиентской части приложения работать даже в той ситуации, когда нет связи с сетью. Весь фокус в том, что такие компоненты могут использовать кэшированную копию данных в виде локального последовательного файла. В результате такая модель позволяет пользователям “запастись” данными, запросив их на сервере, а затем уехать в командировку (или еще куда-нибудь), прихватив с собой компьютер с этими данными. В этой модели компоненты, которые размещаются на компьютере клиента, должны включать средства поддержки логики приложения и некоторых бизнес-правил. Когда такой “мобильный” пользователь по возвращении подключается к сети, а через нее — к остальным компонентам приложения, основная и “мобильная” базы данных обмениваются информацией об изменениях, внесенных за время автономной работы клиента. Эти изменения автоматически фиксируются в обеих базах, восстанавливая их идентичность. Следует сказать, что такая процедура устранения разночтений в двух источниках данных (ее часто называют процедурой восстановления “согласия” — reconciliation — баз данных) может быть довольно сложной, поскольку нет никакой гарантии, что внесенные в разные базы изменения не противоречат друг другу.

Преимущества

- Гибкость на этапе разработки.
- Гибкость в использовании.
- Не требуется покупать новые VCL-компоненты.
- Для реализации такой архитектуры разработчику нужно научиться использовать всего несколько специфических VCL-компонентов.
- Приложение, созданное в рамках этой архитектуры, легко внедряется и обновляется.
- Каждая из описанный выше моделей имеет специфические преимущества.

Недостатки

- Сложная архитектура, которая предполагает включение средств, реализующих множество различных технологий.
- Довольно продолжительный начальный этап разработки.
- Производительность приложения зависит от множества факторов, а потому прогнозировать ее на этапе разработки довольно сложно.

Методы доступа к данным

Перечисленные варианты архитектуры поддерживаются множеством методов доступа к данным. Поскольку эти методы оказывают очень существенное влияние на работу приложения в целом, рассмотрим основные из них.

Специализированные компоненты

Этот вариант поддерживается сторонними производителями. На него имеет смысл ориентироваться в том случае, когда требуется обеспечить максимально возможную производительность БД-приложения. Специализированные компоненты разрабатываются под API определенных типов СУБД и фактически работают в обход BDE.

Преимущества

- Максимально возможная производительность приложения. Использование таких компонентов позволяет избежать непроизводительных затрат, обойти дополнительные процедуры обработки, в том числе и те, которые выполняются BDE, и работать напрямую с API выбранной СУБД.
- Использование специфических возможностей СУБД.
- В подавляющем большинстве случаев значительное уменьшение требований к объему памяти, по сравнению с использованием SQL Links.

Недостатки

- Программист должен освоить методику работы с новыми компонентами.
- Проект попадает в определенную зависимость от стороннего разработчика. Это особенно существенно для долговременных проектов, предполагающих широкое внедрение.
- Компоненты нужно приобретать за отдельную плату, кроме того, иногда необходимо выплачивать авторам отчисления от сумм, полученных в результате реализации проекта.
- Усложняется обновление приложения и его сопровождение, поскольку обновление покупных компонентов должно выполняться их разработчиком.

ODBC с использованием BDE

Этот вариант поддерживается в редакциях C++Builder Professional и Enterprise. ODBC (Open Database Connectivity — открытая архитектура подсоединения баз данных), разработанная в Microsoft, в настоящее время стала своего рода стандартом для многих БД-приложений. В основе ODBC лежит та же идея, что и в основе SQL Links, т.е. средства поддержки ODBC можно рассматривать как промежуточное звено для связи с API СУБД определенного типа. Разница в том, что поскольку ODBC стала своего рода стандартом, то практически все разработчики СУБД включают в комплект своего продукта и ODBC-драйвер. В этом случае любые приложения, настроенные на работу с ODBC, смогут работать и с этим типом СУБД. На сегодняшний день я не знаю ни одной СУБД, работающей на платформе Windows, которая не комплектовалась бы ODBC-драйвером.

Сам по себе программный продукт ODBC имеет двухуровневую архитектуру. Верхний уровень предназначен для интерфейса с приложением (оно в таком случае играет роль

ODBC-клиента) и поддерживает стандартные функции API с базами данных. Нижний уровень представляет собой специализированный драйвер преобразования информации в соответствии с форматом конкретной СУБД.

Преимущества

- ODBC является фактическим стандартом работы с базами данных разного формата.
- Ориентация на использование ODBC позволяет приложению с помощью одних и тех же средств подключаться к базам данных любого формата (если, конечно, разработчики такой СУБД оснастили ее соответствующим драйвером).
- Использование ODBC позволяет с минимальными усилиями преобразовать одноуровневое приложение в приложение типа "клиент/сервер".
- Не нужно приобретать новые компоненты и осваивать методику работы с ними.

Недостатки

- Совместное использование BDE и ODBC требует очень большого объема оперативной и дисковой памяти, что, кроме всего прочего, отнюдь не способствует высокой производительности работы приложения. Кроме того, использование ODBC означает, что на пути потока данных появляются два дополнительных слоя преобразования.
- Некоторые ODBC-драйверы недостаточно надежны. Прежде чем принимать решение об их использовании, нужно провести тщательное тестирование.
- Сопровождение и обновление БД-приложения, использующего ODBC, также представляет определенную проблему, поскольку VCL, BDE, базовая программа ODBC, ODBC-драйвер и СУБД обновляются разработчиками независимо.

Подключение к ODBC через специализированные компоненты

Этот вариант поддерживается сторонними производителями и является довольно привлекательным компромиссным решением, обеспечивающим, с одной стороны, неплохую производительность, а с другой, — гибкость в выборе СУБД и не очень большую зависимость от стороннего разработчика компонентов. Специализированные компоненты работы с базами данных позволяют отказаться от услуг BDE и напрямую связываться с API ODBC. В результате несколько повышается производительность, в сравнении с предыдущим вариантом, и требуется меньше ресурсов памяти.

Преимущества

- Потенциально достаточно высокая производительность, хотя это во многом определяется качеством используемых компонентов и ODBC-драйвера для выбранного типа СУБД. Учтите, что эти компоненты программы поступают из разных источников.
- ODBC является фактическим стандартом работы с базами данных разного формата.
- Ориентация на использование ODBC позволяет приложению с помощью одних и тех же средств подключаться к базам данных любого формата (если, конечно, разработчики такой СУБД оснастили ее соответствующим драйвером).
- Использование ODBC позволяет с минимальными усилиями преобразовать одноуровневое приложение в приложение типа "клиент/сервер".

Недостатки

- Программисту придется освоить методику работы с новыми компонентами.
- Проект попадает в определенную зависимость от стороннего разработчика. Это особенно существенно для долговременных проектов, предполагающих широкое внедрение.
- Компоненты нужно приобретать за отдельную плату и, кроме того, иногда выплачивать авторам отчисления от сумм, полученных в результате реализации проекта.
- Некоторые ODBC-драйверы недостаточно надежны. Прежде чем принимать решение об их использовании, нужно провести тщательное тестирование.
- Сопровождение и обновление БД-приложения, использующего ODBC, также представляет определенную проблему, поскольку приобретаемые компоненты, базовая программа ODBC, ODBC-драйвер и СУБД обновляются разработчиками независимо.

ADO (ActiveX)

Этот вариант поддерживается в редакции C++Builder Enterprise. ADO (ActiveX Database Objects) — новая технология работы с базами данных, разработанная в Microsoft. Средства поддержки ADO образуют промежуточный уровень, который упрощает доступ к базам данных OLE. Реализуется такой доступ при помощи технологии COM (это еще одно изобретение Microsoft). Организацию доступа с помощью ADO целесообразно использовать в Internet-приложениях, серверных приложениях или приложениях другого типа, в которых используется технология COM.

Преимущества

- Эта технология позволяет обойтись без BDE.
- Хорошо согласуется с работой в сети Internet.
- Легко внедряется.
- Поддерживает модель *briefcase*.
- Поддерживает работу с XML.
- Нет необходимости приобретать дополнительные компоненты. Фирма Inprise включила средства поддержки этой технологии в комплект поставки C++Builder 5. Более подробно о компонентах Borland, поддерживающих работу с ADO, вы сможете узнать в разделе *Компоненты для работы с ADO в составе C++Builder* далее в этой же главе.
- При подключении к СУБД, поддерживающим функционирование механизма OLE (например, MS-Access), обеспечивается более высокая производительность, чем при использовании BDE.

Недостатки

- Можно применять только в операционной среде Windows.
- Не все провайдеры OLE-СУБД отличаются высокой надежностью в работе. Прежде чем принимать окончательное решение об использовании этого метода доступа, рекомендуется тщательно протестировать провайдер, который планируется использовать в приложении.
- Обеспечивает несколько более узкий набор функций управления, чем BDE.

- Некоторые функции можно реализовать только с помощью средств фирмы Microsoft.
- Не годится для работы с MIDAS.
- Несмотря на то что все необходимые компоненты включены в состав C++Builder 5, они существенно отличаются от компонентов, поддерживающих другие технологии, а потому придется затратить определенное время на освоение методики работы с этими компонентами.

Внедренный SQL-код

Этот вариант поддерживается фирмой Borland/Inprise и сторонними производителями. Хотя C++Builder и считается одной из лучших платформ для быстрой разработки приложений, все же некоторые операции оказываются слишком узкоспециализированными или сложными для парадигмы, положенной в основу этой среды. Поэтому лучшие результаты можно получить, манипулируя данными напрямую. Приобретение предкомпилятора SQL позволяет программисту манипулировать данными с помощью выражений на псевдо-SQL, непосредственно внедренным в исходный программный код. Этот предкомпилятор затем производит лексический анализ внедренного кода и преобразует его в код на языке C.

Преимущества

- Непревзойденная гибкость. Например, манипулирование некоторыми типами данных, специфическими для определенной СУБД, можно выполнить только с помощью такой технологии.
- Максимально высокая производительность.
- Отсутствует необходимость пользоваться услугами BDE.

Недостатки

- Программисту придется изучить новую методику и новые программные продукты.
- Программист должен досконально знать язык SQL.
- Нельзя использовать визуальные компоненты работы с базами данных из библиотеки VCL. Все манипуляции с информацией из базы данных должны выполняться программно.
- Необходимы дополнительные расходы.

API СУБД

Этот вариант поддерживается разработчиком используемой СУБД. Обычно такие средства используются для работы с относительно простыми базами данных, теми, например, которые основаны на модели ISAM (в частности, семейство xBase). Выбирая такой вариант, вы тем самым отказываетесь от использования технологии визуальной разработки в пользу мощных средств доступа к данным, которые имеются в составе СУБД.

Преимущества

- Максимально возможная для данной модели СУБД производительность.
- Используемые средства включены в состав СУБД.

Недостатки

- Может увеличиться срок разработки.
- Программист должен освоить все тонкости работы с API определенной СУБД. Как правило, программирование ведется на языках достаточно низкого уровня.
- Нельзя использовать визуальные компоненты работы с базами данных из библиотеки VCL. Все манипуляции с информацией из базы данных должны выполняться программно.

Архитектура БД-приложений — выводы

Обзор вариантов архитектуры БД-приложений, с которым вы познакомились, конечно же, не претендует на всеобъемлющую полноту, но даже такой краткий материал дает достаточную пищу для размышлений. Рассмотренные варианты архитектуры схематически изображены на рис. 14.1. Выбор варианта для конкретного проекта должен быть взвешенным и осознанным, и, надеюсь, соображения, которые мы привели выше, помогут вам.

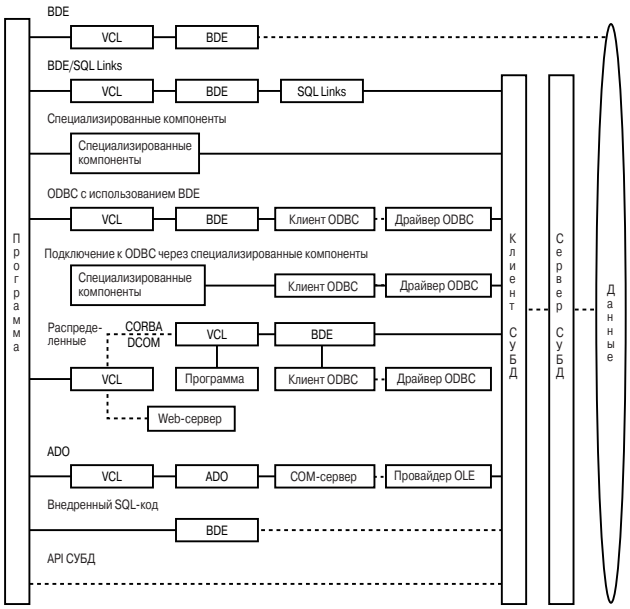


Рис. 14.1. Варианты архитектуры, представленные в этом разделе. Пунктирными линиями обозначены связи по сети

Источники дополнительной информации

Ниже перечислены информационные материалы, в которых вы сможете найти достаточно много интересных сведений об архитектуре БД-приложений:

- *Borland C++Builder 5 Developer's Guide* (Руководство, поставляемое в комплекте с C++Builder), Inprise Corporation: Главы 13–15, 23 и 25.
- Marco Cantú, *Data Access Dilemma*, Web-сайт сообщества пользователей продуктов Borland; адрес в Internet: `http://community.borland.com/article/0,1410,20191,00.html`

- Charlie Calvert, *Accessing Databases Using ADO and Delphi*, Web-сайт сообщества пользователей продуктов Borland; адрес в Internet: `http://community.borland.com/soapbox/techvoyage/article/1,1795,10270,00.html`

Язык SQL

В этом разделе мы познакомим вас с основными возможностями языка структурированных запросов SQL. Использование языка SQL (Structured Query Language) открывает перед программистом новые возможности общения с информацией, хранящейся в базах данных, посредством текстовых команд. Хотя в разных СУБД используются специфические диалекты SQL, базовые операции, такие как создание таблиц и индексов, заполнение таблиц данными, выборка информации, практически везде выполняются одинаково. Будет показано, как использовать средства формирования и модификации структуры таблицы, вставлять, редактировать и удалять отдельные записи, искать нужную информацию в базе данных. Еще раз обращаю ваше внимание на то, что синтаксис отдельных SQL-выражений в некоторых СУБД может отличаться от канонического, но основные концепции одни и те же.

На заметку В примерах этого раздела используется база данных, структура которой подробно анализируется ниже, в разделе *Набор компонентов InterBase Express*. Все приведенные здесь примеры можно опробовать в действии с помощью утилиты ***InterBase Interactive SQL***.

Таблицы и индексы

Начнем с того, что с помощью выражений SQL сформируем новую таблицу `program`:

```
create table program (
     pro_id           integer not null,
     pro_name         char (80) not null,
     pro_made         date not null);
```

С помощью этого выражения создается таблица с тремя полями: `pro_id`, `pro_name` и `pro_made`. Следом за наименованием поля задается тип данных в нем, причем для поля `pro_name` также указывается размер поля. После типа идет необязательная часть объявления поля, с помощью которой специфицируется, может ли данное поле иметь значение `null`. Все поля, формируемые данным выражением, не могут содержать значения `null`. Если же такое значение допускается в каком-либо из полей, то нужно удалить из соответствующей строки выражения объявление `not null`.

Однако в созданной таким выражением таблице будет отсутствовать первичный ключ. Объявить первичный ключ можно двумя способами. Первый способ — включить объявление непосредственно в выражение формирования таблицы, как показано ниже:

```
     pro_made date not null,
primary key (pro_id));
```

Альтернативный способ — создать индекс по первичному ключу в отдельном SQL-выражении, которое начинается командой `alter`. Команда `alter` в SQL-выражениях задает модификацию ранее созданных объектов — таблиц или индексов. Выражение, в котором создается индекс по первичному ключу в ранее сформированной таблице `program`, будет иметь вид:

```
alter table program
add primary key(pro_id);
```

Эта команда модифицирует таблицу program таким образом, что поле pro_id становится полем первичного ключа. С помощью команды alter table можно добавлять или удалять из таблицы разнообразные компоненты — поля, ключи, отношения и ограничения.

Если в таблице уже имеются какие-либо данные, то в нее можно добавлять только такие поля, которым "разрешено" хранить значение **null**. Объясняется такое ограничение очень просто: если полю запрещено хранить значение **null**, то каким образом СУБД определит, какое значение установить для этого поля в уже существующих записях таблицы? Существуют и другие нюансы, связанные с изменением структуры таблицы, в которой желательно сохранить имеющиеся данные. Поэтому во избежание подобных сложностей еще на этапе разработки приложения тщательно продумайте структуру создаваемых таблиц. На этом этапе легко скорректировать структуру данных, поскольку нет необходимости сохранять записанную информацию. Позже, когда созданные таблицы будут заполнены данными, изменить их структуру значительно сложнее.

Точно такой же синтаксис выражения используется и при программировании операции удаления какого-либо компонента таблицы. Единственное отличие — вместо SQL-команды add используется команда drop. Например, приведенное ниже выражение удалит из таблицы program поле pro_made:

```
alter table program
drop pro_made;
```

А вот для удаления из базы данных всей таблицы целиком используется выражение drop table program. Но будьте осторожны: если вы попытаетесь выполнить этот оператор в том виде, как он здесь приведен, это будет рассматриваться как ошибка. Причина в том, что две другие программы в той же базе данных имеют внешние (foreign) ключи, которые ссылаются на таблицу program; если таблица program будет удалена, то нарушится целостность ссылок.

Избежать появления подобных ошибок довольно просто: перед удалением таблицы удалить в других таблицах все внешние ключи, ссылающиеся на "обреченную" таблицу. Снова хочу обратить ваше внимание на то, что всех этих сложностей можно избежать, если на этапе проектирования тщательно продумать структуру базы данных.

Параметры

Для задания значений в определенных SQL-выражениях можно использовать параметры. После того как выражение с параметрами подготовлено, можно специфицировать значения параметров и выполнить операцию, заданную параметризованным выражением. Имя параметра начинается с двоеточия (:). Использовать параметры можно в самых разных SQL-выражениях

Рассмотрим следующее выражение:

```
insert into program(pro_id, pro_name, pro_made)
↳.values(:the_id, :the_name, :the_date)
```

Это выражение задает вставку в таблицу program и включает три параметра: the_id, the_name и the_date. Это выражение затем может выполняться в программе многократно, причем для каждой очередной операции вставки записи можно специфицировать значения параметров.

Такую методику очень удобно использовать в сочетании с объектом класса C++Builder TQuery или аналогичного ему. Можно создать объект TQuery, в который будет включено приведенное выше SQL-выражение, а затем вставлять записи в таблицу program, присваивая параметрам нужные значения.

Команды insert, update, delete и select

Создав таблицу, можно наполнять ее данными. Для этого в SQL имеется четыре команды и соответственно четыре формата выражений: insert, update, delete и select.

Команда insert

Команда insert Позволяет добавлять строки (записи) в таблицы базы данных. Например, выражение вставки в таблицу program новой записи, которая содержит определенные значения, имеет вид:

```
insert into program
      values(1, 'Program Name', 7/4/00);
```

Можно применять и такой формат выражения:

```
insert into program(pro_id,pro_name,pro_made)
       values(1, 'Program Name', 5/5/00);
```

Результат выполнения обоих выражений будет один и тот же. Но второй вариант позволяет явно указать поля новой записи в таблице, в которые заносятся заданные значения. Это особенно удобно в таблицах, которые оснащены специальным механизмом автоматического заполнения новых полей значениями по умолчанию (такой механизм получил наименование *триггера таблицы*). Кроме того, поля, которым при формировании было разрешено иметь значение **null**, будут автоматически заполняться этим значением, если они не перечислены в списке полей выражения insert. Значения во второй строке выражения могут быть либо статическими данными, либо параметрами, либо выражениями select, которые возвращают значение одного поля (колонки) (Такие выражения получили наименование *однотонных — singleton*.)

На место строки значений в выражении insert можно вставить выражение select, но это выражение должно возвращать данные того типа, который совпадает с типом полей, перечисленных в списке полей. Такая форма выражения insert используется для копирования данных из одной таблицы в другую.

Команда update

Выражение update позволяет изменять значение в одном или нескольких столбцах записей, ранее внесенных в таблицу. Рассмотрим выражение:

```
update bugs
set
   bug_resolved =0,
   r_id =null
where r_id =:r_id
```

Это выражение можно использовать в том случае, если необходимо удалить запись из таблицы revision, одновременно обеспечив отсутствие в таблице bugs записей, которые ссылаются на нее через внешний ключ. Это выражение изменяет таблицу bugs и записывает в bug_resolved значение **0**, а полю r_id присваивает значение **null** во всех записях, в которых значение поля r_id совпадает со значением параметра r_id .

Синтаксис этого выражения незатейлив. Все поля, которым мы собираемся присвоить новые значения, перечислены (через запятую) после update <*имя_таблицы*> set. После ключевого слова where задается критерий отбора модифицируемых записей. Формат критериев отбора в SQL-выражениях будет подробнее описан в разделе *Команда select*.

Команда delete

SQL-выражение delete позволяет удалить выбранные записи из заданной таблицы (вряд ли можно ожидать чего-нибудь другого от команды с таким именем). В выражении delete также можно специфицировать критерии отбора удаляемых записей. Рассмотрим следующее выражение:

```
delete from bugs
where bug_resolved = 0 and bug_id > 10
```

Это выражение удалит из таблицы bugs все записи, в которых выполняется условие, заданное после ключевого слова where. Если в выражении отсутствует where-компонент, то из таблицы bugs будут удалены все записи.

Команда select

До сих пор речь шла о тех средствах SQL, которые позволяют создавать таблицы определенной структуры, изменять эту структуру, заполнять таблицу данными и модифицировать данные (добавлять, заменять или удалять). Теперь же перейдем к средствам, обеспечивающим выборку информации из базы данных. Для выполнения подобных процедур в составе SQL имеется команда select. Она позволяет извлекать информацию из базы данных — формировать подмножество отобранных данных. Представьте себе, что в таблице program хранится информация, представленная в табл. 14.1.

Таблица 14.1. Данные в таблице program

Pro_id	Pro_Name	Pro_Made
1	First Try	9/9/1999
2	Second Try	9/9/1999
3	Last Try	6/9/2000

Предположим, нам нужны все сведения о программе, разработанной в определенный день. В этом случае нужно воспользоваться следующим SQL-выражением в командой select:

```
select * From Program where Pro_Made =9/9/1999
```

Символ * в выражении select означает, что в формируемое подмножество нужно извлекать значения всех полей отобранных записей. Можно поступить и по-другому: вместо * явно указать только те поля, которые нас интересуют. После ключевого слова From следует та часть выражения, в которой указывается, из какой именно таблицы извлекается информация. В рассматриваемом выражении речь идет о таблице program. И наконец, после ключевого слова where приводятся критерии отбора записей (фильтр) — в рассматриваемом выражении отбираются записи, которые в поле Pro_Made имеют значение 9/9/1999 (по замыслу разработчиков таблицы в этом поле хранится дата разработки программы).

Функции совместной обработки записей

Рассмотрим пять функций совместной обработки записей — SUM, MIN, MAX, COUNT и AVG. Ниже перечислены краткие сведения о каждой из них.

- SUM(). Вычисляет сумму значений заданного числового поля по всем записям, которые удовлетворяют определенному условию.

- `MIN()`. Возвращает наименьшее среди всех записей таблицы значение заданного числового поля.
- `MAX()`. Возвращает наибольшее среди всех записей таблицы значение заданного числового поля.
- `COUNT(*)`. Возвращает количество записей, удовлетворяющих заданному условию.
- `AVG()`. Вычисляет среднее значение заданного числового поля по всем записям таблицы.

Например, чтобы определить, сколько в таблице записей о программах, разработанных 9/9/1999, нужно использовать следующее выражение:

```
select count(*) from Program where Pro_Made = 9/9/1999
```

Результатом выполнения этого оператора будет число **2**.

Итак, мы кратко познакомили вас с основными командами языка SQL. Этих сведений вполне достаточно для того, чтобы понять последующий материал книги. Тем же, кто пожелает более глубоко изучить язык SQL, я советую обратиться к следующим источникам.

- Harrington. *SQL Clearly Explained*, Morgan Kaufmann Publishers; ISBN: 012326426X. Эта книга — вводный курс в язык SQL и будет полезна новичкам и программистам средней квалификации.
- Celko. *SQL For Smarties*, Morgan Kaufmann Publishers; ISBN: 1558605762. Это более серьезная книга и содержит исчерпывающие сведения о языке SQL и методике работы с ним.

Компоненты для работы с ADO в составе C++Builder

ADO (аббревиатура от *ActiveX Database Objects*) — это разработанная в Microsoft технология доступа к базами данных, которая должна прийти на смену ODBC (Open Database Connectivity) и более ранним технологиям, таким как DAO и RDO. Программный продукт Microsoft ADO представляет собой, по сути, оболочку механизма OLE DB, которая должна заменить ODBC API — комплекс программ, базирующихся на языке C. Microsoft ADO значительно облегает реализацию объектно-ориентированного подхода при работе с базами данных, причем позволяет программировать операции любой сложности, в том числе и операции нижнего уровня.

Компоненты ADO берут на себя большую часть работы, связанной с использованием технологии COM, и позволяют использовать практически ту же методику работы с базами данных, что и обычные компоненты этого назначения из библиотеки Borland VCL, такие как сетки (grid), поля редактирования и диаграммы. Это стало возможным благодаря тому, что компоненты ADO являются потомками класса `TDataSet`, а объекты класса `TDataSource` могут работать с любыми потомками `TDataSet`. Средства поддержки ADO позволяют использовать эту технологию при работе с любыми СУБД, оснащенными драйверами ODBC. Кроме того, ADO предлагает разработчику поддержку данных, хранящихся в нереляционном формате, например XML-данных или сообщений электронной почты, если только соответствующий провайдер удовлетворяет требованиям стандартов ADO. Теоретически к любым данным, полученным с помощью ADO, можно применять выражения на языке SQL, но для нереляционных данных требуется использовать специальные формы SQL-операторов.

Сравнение ADO и BDE

Среди программистов, работающих в средах Delphi и C++Builder, всегда были популярными VCL-компоненты, которые не требуют обращения к BDE. (BDE, по мнению многих разработчиков, замедляет работу приложения и требует слишком больших ресурсов памяти — оперативной и дисковой.)

До появления ADO большинство таких "независимых от BDE" компонентов были специализированными и могли работать только с определенным типом СУБД, например dBase или InterBase, либо со средствами поддержки определенной технологии, например ODBC. Существовали и такие компоненты, которые обладали собственными средствами выполнения функций BDE. Однако при работе с этими компонентами возникала проблема уникальности интерфейса, который разительно отличался от привычного интерфейса компонентов, ориентированных на BDE. Поэтому такие приложения было очень сложно модифицировать, особенно если возникала необходимость включить в них BDE. К сожалению, тот же недостаток присущ и ADO-компонентам.

У ADO, в сравнении с BDE, есть и другие недостатки. Во-первых, при использовании ADO нужно явно указывать в строке подключения имя провайдера сервера и базы данных, а это затрудняет переориентирование ADO-компонента на работу с другой базой данных или другим типом СУБД. Во-вторых, в отличие от BDE, для однородного объединения таблиц из разных баз данных нельзя использовать SQL в сочетании с ADO-компонентами (причем это не зависит от того, созданы ли эти базы данных с помощью одной и той же СУБД или разных СУБД).

Есть и еще одно предостережение, которое касается сопровождения приложений, использующих ADO-компоненты. Нужно постоянно отслеживать изменения, которые вносятся разработчиками в элементы операционной среды, связанные с работой ADO-компонентов и СУБД (например, ядро Jet, с которым работают базы данных Access).

Учитывая все сказанное, каковы же преимущества технологии ADO?

- Бо́льшая часть программных средств поддержки этой технологии поставляется в составе операционной системы, а потому разработчик БД-приложения избавлен от необходимости их внедрения.
- Использование ADO позволяет получить доступ к данным, созданным с помощью нетрадиционных технологий, таких как XML.
- Использование ADO-компонентов позволяет программисту, знакомому с инструментальными средствами Microsoft, например Visual C++, без особого труда переориентироваться на C++Builder.
- Компоненты ADO допускают асинхронное выполнение операторов SQL и позволяют отслеживать процесс выполнения команд с помощью обработчиков событий. Это дает пользователю наглядную информацию о том, насколько далеко продвинулось выполнение запроса.
- В отличие от BDE-компонентов, механизм ADO позволяет остановить работу программ базы данных без потери информации с помощью `Program Reset`.
- И наконец, знакомство с концепцией и методикой работы с ADO очень поможет вам в будущем найти работу в проектах, руководство которых ориентируется на применение ADO.

Прежде чем приступить непосредственно к работе с ADO, задумайтесь над такой идеей. Неплохо создать между БД-приложением и "фирменными" ADO-компонентами, которые предполагается использовать в этом приложении, тонкий слой компонентов, производных от фирменных. Тогда в приложении будут использоваться только такие промежуточные компоненты, в которые при желании всегда можно внести небольшие изменения, не трогая при этом фирменных. Вносимые изменения всегда можно "спрятать" при помощи условной трансляции, применяя директивы `#ifdef #else #endif` в сочетании с `#define`.

К сожалению, класс TBatchMove не может работать с ADO-компонентами. Вам придется разработать собственный родовой класс для выполнения операций копирования, который мог бы работать с ADO-компонентами и выполнять те же функции, что и TBatchMove.

Компоненты группы ADOExpress

Ниже представлен перечень компонентов группы ADOExpress, которые находятся на вкладке ADO палитры компонентов. При описании каждого компонента этой группы указано, функции какого BDE-компонента он выполняет по отношению к ADO.

- TADOConnection. Выполняет подключение к базе данных и управляет транзакциями. По своим функциям этот компонент равнозначен BDE-компоненту TDatabase. Обратите внимание на то, что в составе ADO-компонентов отсутствует "двойник" TSession — все функции этого компонента берет на себя TADOConnection.
- TRDSConnection. Выполняет подключение к базе данных в режиме, который позволяет организовать доступ к набору записей со стороны многоуровневого БД-приложения. Этот компонент можно использовать только в сочетании с TADODataSet. Поддерживает функции ***Microsoft Remote Data Space***, которые обеспечивают передачу набора данных ADO через уровни в многоуровневом приложении. Обращаю ваше внимание на то, что эти функции компонентом TRDSConnection выполняются не так, как поддержка удаленных модулей данных в MIDAS.
- TADOTable. Организует доступ к отдельной таблице базы данных по ее имени; фактически — "двойник" BDE-компонента TTable.
- TADOQuery. Организует доступ к одной или нескольким таблицам определенной базы данных в процессе выполнения команды SQL. Может выполнять любые команды SQL, в том числе SELECT, INSERT, DELETE, ALTER TABLE; фактически — "двойник" BDE-компонента TQuery.
- TADOStoredProc. Организует выполнение хранимых процедур из определенной базы данных; фактически — "двойник" BDE-компонента TStoredProc.
- TADODataSet. Выполняет практически те же функции, что и TADOQuery, за исключением того, что с помощью этого компонента нельзя выполнять SQL-выражения, которые не возвращают набор записей.
- TADOCommand. По существу, выполняет те же функции, что и TADOQuery. С помощью этого компонента можно выполнять любые команды SQL, но нельзя получить доступ в сформированному SQL-командой набору записей. Для этого нужно использовать отдельный объект TADODataset.

Иерархия классов компонентов группы ADOExpress показана на рис. 14.2. Как видно на этой схеме, большинство ADO-компонентов являются потомками TDataSet.

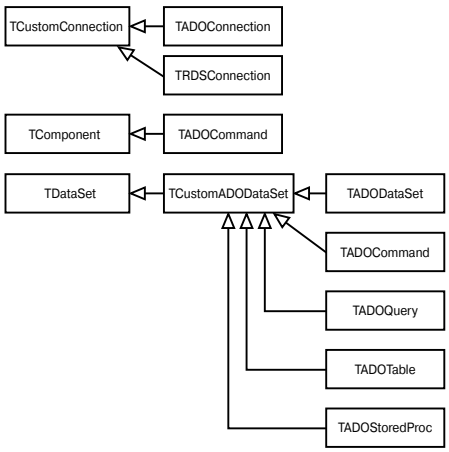


Рис. 14.2. Иерархия классов ADO-компонентов

Подключение к базе данных

В результате подключения к базе данных определенный компонент получает доступ к хранящимся в ней данным через посредство определенного драйвера. До завершения подключения доступ к информации, хранящейся в базе данных, закрыт.

Класс TADOConnection

Этот класс выполняет те же функции, что и BDE-классы — `TSession` и `TDatabase`. Чаще всего объект этого класса в БД-приложении выполняет подключение к базе данных. В принципе, использовать в приложении объект этого класса необязательно, поскольку каждый ADO-компонент располагает собственным методом для организации подключения и может содержать собственный вариант строки подключения.

Провайдер

Собственно подключение к базе данных выполняется в ADO с помощью провайдера — именованного объекта OLE, который реализует функции интерфейса OLE DB (Object Linking and Embedding Database). Провайдер предоставляет в распоряжение других программных компонентов набор интерфейсов, с помощью которых можно единообразно обращаться к базам данных разных типов.

Строка подключения

Строка подключения играет ту же роль, что псевдоним в BDE и более сложные объекты данных, стоящие "за спиной" псевдонима. Строка подключения в ADO идентична строке подключения, используемой при работе через ODBC, а именно такому ее формату, который пригоден для взаимодействия со стандартным провайдером ODBC. Формат строки подключения провайдеров может слегка различаться, но в основном она выглядит примерно так:

```
Provider=SQLOLEDB.1;Persist Security Info=False;
↳ User ID=sa;Initial Catalog=mydatabase
```

Ниже перечислены отдельные компоненты строки подключения.

- `Provider`. Этот компонент содержит имя COM-объекта провайдера, которое в процессе подключения следует отыскать в системном реестре. На практике разработчик может и не знать заранее этого имени. Вместо него можно вставить в строку приглашение пользователю ввести имя провайдера. Получить имена провайдеров, доступных на определенном компьютере, можно либо с помощью довольно сложной технологии перечисления OLE DB-объектов (эта технология описана в документе по адресу `http://msdn.microsoft.com/library/default.asp?URL=/library/psdk/dasdk/olpr6igj.htm`), либо используя специальное диалоговое окно, формирующее строку подключения (оно обсуждается в следующем разделе).
- `User ID` и `Password`. Эти данные формируются так же, как и в свойстве `LoginPrompt` BDE-компонентов, за исключением того, что при работе с ODBC ключевые слова всегда состоят из прописных букв, а в ADO можно использовать любые.
- `Initial Catalog`. Через этот компонент передается имя базы данных, с которой необходимо установить связь.

Транзакции

В объекте TADOConnection можно организовать запуск, подтверждение или откат транзакции.

Если отдельный объект подключения использовать не предполагается, то, как следует из документации, для этой цели можно было бы воспользоваться свойством Connection объектов TADOTable или TADOQuery аналогично тому, как используется свойство Database в BDE-компонентах. (Свойство Database позволяет получить доступ к объекту базы данных по умолчанию, заданному в объекте BDE-класса TSession.) Но, к сожалению, реальность не совсем согласуется с тем, что изложено в документации. Свойство Connection в объекте ADO-компонента остается равным NULL, даже когда ADO-объект открыт и имеет собственную строку подключения. Таким образом, если необходимо выполнять транзакции, придется создать отдельный объект TADOConnection и использовать его для подключения к объекту класса-наследника TADOCustomDataSet.

Использование значений по умолчанию

В структуре класса TADOConnection имеется множество свойств и методов. Особые случаи, когда имеет смысл изменить поведение этого объекта, определяемое значением свойств по умолчанию, обсуждаются при анализе механизма выполнения транзакций и методов оптимизации производительности.

Доступ к наборам записей

После подключения БД-приложения к базе данных следует получить доступ к наборам записей (данные могут быть извлечены как из отдельной таблицы, так и из нескольких связанных). Для доступа к данным используется несколько типов ADO-компонентов.

Доступ к набору записей с помощью компонента TADOTable

Для того чтобы получить доступ к набору записей с помощью объекта компонента TADOTable, потребуется выполнить следующие операции.

1. Установить соединение с базой данных, из которой предполагается извлекать данные.
2. Указать имя нужной таблицы из этой базы данных.
3. Открыть таблицу.

После этого нужно подключить объект TADOTable через объект TDataSource к одному из визуальных компонентов работы с данными, который берет на себя функции отображения информации и ее редактирования.

Подключение к базе данных с помощью TADOTable

Как уже упоминалось, организовать подключение базы данных к БД-приложению, использующему технологию ADO, можно по-разному.

Во-первых все объекты ADO-компонентов в приложении могут пользоваться соединением, организованным с помощью объекта TADOConnection. Для этого в свойстве Connection ADO-объекта должна быть указана ссылка на соответствующий объект TADOConnection, причем эта ссылка может быть установлена как на этапе разработки программы, так и в процессе ее выполнения.

Во-вторых, открытый ADO-объект может организовать подключение и самостоятельно, передав соответствующую строку подключения. Эта строка может быть сформирована заранее, на этапе разработки программы, или оперативно, в процессе выполнения программы. Хочу обратить ваше внимание на то, что программно изменять строку подключения в свой-

стве ConnectionString или ссылку в свойстве Connection объекта класса-наследника TADOCustomDataSet можно только в том случае, если объект неактивен. Это справедливо и в отношении объекта класса TADOConnection, если вы собираетесь программно изменять значение строки подключения в его свойстве ConnectionString.

Задание имени таблицы в TADOTable

Имя таблицы подключенной базы данных, из которой будет извлекаться информация, задается в свойстве TableName объекта TADOTable. Задать имя таблицы можно как при разработке программы, так и в процессе ее выполнения, но в последнем случае объект TADOTable сначала должен быть закрыт, а уже потом можно изменить значение его свойства.

Открытие объекта TADOTable

Для того чтобы открыть (активизировать) объект класса TADOTable, нужно, как и при работе с BDE-компонентами, либо установить значение **true** в его свойстве Active, либо вызвать метод Open(). При этом будет открыта специфицированная в этом объекте таблица.

Связь между объектом TADOTable и визуальными компонентами

Объекты класса TADOTable, как и всех других классов-наследников TDataSet, можно использовать в приложении в сочетании с объектом класса TDataSource, и на такой объект может быть установлена ссылка в свойстве, которое имеет тип TDataSet. Обращаю ваше внимание на то, что в том классе, который вы собираетесь использовать для вывода или редактирования данных, соответствующее свойство должно иметь тип именно TDataSet *, а не TADOTable * или TTable *. Это позволит достичь максимальной гибкости в использовании объектов этого класса, поскольку их можно будет применять в сочетании как с BDE-компонентами, так и с ADO-компонентами.

Перемещение по таблице с помощью TADOTable

Класс TADOTable наследует от TDataSet методы First(), Eof(), Next(), Prior() и Last(), с помощью которых можно перемещаться по открытой таблице.

Добавление или редактирование записей с помощью TADOTable

Для создания новой записи используются методы Append() или Insert(), а для подготовки к изменению значений определенных полей — метод Edit(). Свойства FieldByName, Fields->Field[Index] и другие свойства и методы, которые имеют в классе одинаковые имена со свойствами и методами BDE-класса-"двойника", выполняют такие же функции. Более подробную информацию о них вы можете получить в оперативной справке.

Поиск записи с помощью TADOTable

Для поиска записи в открытой таблице в классе TADOTable используются те же методы, что и в аналогичных BDE-классах. Это справедливо для всех прочих классов-наследников TCustomADODataSet. Например, методы Locate() и Lookup() можно использовать, соответственно, для перемещения курсора базы данных на какую-нибудь запись или для извлечения значений из записи без изменения положения курсора.

Использование фильтров в сочетании с TADOTable

ADO-компоненты поддерживают фильтры для обеспечения функциональной совместимости с BDE-компонентами. Но работать с фильтрами в сетевой среде нерационально. Фильтр — это средство клиентского приложения, а потому клиенту передается полный набор записей, а уже на стороне клиента производится их фильтрация. Поэтому, совершенно очевидно, через сеть пере-

дается большое количество данных, которые затем, по сути, отправляются "на свалку". Если вы не можете позволить себе такую роскошь, используйте механизм запросов и включите условия фильтрации в настройку объекта TADOQuery — фильтрация будет выполняться на стороне сервера, и информационный мусор не будет забивать сетевые каналы связи. Конечно, при работе с базами данных локальных СУБД, таких как dBase, Paradox или Access, все эти рассуждения теряют смысл. При использовании ADO-компонентов для работы с фильтрами нужно учитывать, что эти компоненты более строго относятся к спецификации условий фильтрации, чем аналогичные BDE-компоненты. В частности, требуется точное соответствие в расстановке пробелов по обе стороны от оператора. Так, выражение фильтра `SomeField = 'String'` будет воспринято нормально, а выражение `SomeField='String'` — нет.

Кроме того, BDE-компоненты позволяют выполнять частичное сравнение строковых полей, если в спецификации фильтра используется символ универсальной подстановки "звездочка" (*) в литерале. Этот прием можно использовать и в сочетании со знаком сравнения на равенство, например в выражении `SomeField='String*'`, которое даст совпадение с любым значением, начинающимся с String. При работе с ADO-компонентоми такое "приблизительное" сравнение также можно организовать, но для этого потребуется использовать в выражении ключевое слово LIKE на месте знака равенства. Учтите, что при использовании в объекте ADO-компонента символов универсальной подстановки (*) в выражении фильтра никакого сообщения об ошибке не последует (правда, и записей тоже).

Доступ к набору записей с помощью TADOQuery

Компонент TADOQuery предлагает разработчику практически тот же набор "услуг" — свойств и методов, — что и компонент TADOTable (вернее, BDE-компонент TQuery). Но имеются определенные нюансы, которые мы рассмотрим ниже.

- Естественно, в объекте этого компонента нужно явно специфицировать SQL-выражение запроса. Но учтите, что, в отличие от своего BDE-собрата, ADO-компонент не предлагает вам воспользоваться "локальным SQL", который поможет пренебречь спецификой диалекта определенной СУБД. Поэтому при формировании SQL-выражения нужно жестко следовать синтаксису именно того диалекта, который используется в СУБД. Желательно избегать выражений, которые требуют специального синтаксиса. Это поможет без труда переключаться на работу с разными типами СУБД. Нельзя также использовать в SQL-выражениях ссылок на другие базы данных или СУБД.
- Как и при работе с компонентом TQuery, используемые SQL-выражения могут содержать параметры. В компоненте TADOQuery параметры SQL-выражения передаются через свойство Parameters (а не Params), причем тип данных в этом свойстве TParameter, а не TParam. Как и при работе с запросами, вставка или изменение записей в таблицах не отображается в данных запроса до обновления набора записей. Для этого набор записей нужно закрыть, а затем открыть повторно. Компонент располагает рядом свойств, которые помогают повысить скорость обновления — о них мы поговорим в разделе *Настройка оптимальной производительности* далее в этой же главе.

Работа с хранимыми процедурами с помощью TADOStoredProc

Хранимые процедуры — одна из главных "изюминок" программирования баз данных типа клиент/сервер. Хотя технология хранимых процедур поддерживается не во всех СУБД (в частности, в локальных СУБД, как Access или dBase), эти сценарии, написанные на языке SQL, могут существенно упростить и ускорить выполнение повторяющихся операций на стороне сервера в системе клиент/сервер.

Настройка TADOStoredProc

Как при работе с любыми ADO-компонентами, сначала необходимо организовать подключение к базе данных. Это можно сделать посредством соответствующей установки свойства Connection или через свойство ConnectionString. Имя хранимой процедуры нужно задать в свойстве ProcedureName. Поскольку форматы имен хранимых процедур в разных СУБД слегка отличаются не поленитесь заглянуть в соответствующую документацию. Учтите, что имена хранимых процедур для базы данных можно получить с помощью методов, которыми располагает класс TADOConnection. И наконец, установите необходимые значения параметров — для этого нужно использовать объекты класса TParameter, которые входят в состав объекта класса-коллекции TParameters, хранящегося в свойстве Parameters. Обратите внимание, что параметры должны иметь в качестве типа квалификаторы направления передачи (input, output и т.д.), а значения могут быть только типа Value (как и при работе с компонентом TADOQuery).

Выполнение процедур с помощью TADOStoredProc

Если хранимая процедура представляет собой команду или набор команд, не возвращающих никакого значения (например, UPDATE или DELETE), воспользуйтесь методом ExecProc(). Если же процедура должна вернуть какое-либо значение, например результат выполнения запроса, воспользуйтесь методом Open() или присвойте свойству Active значение true.

Извлечение результата с помощью TADOStoredProc

Хранимая процедура может возвращать результат выполнения через параметры общего назначения либо через специальный зарезервированный параметр, который имеет квалификатор направления pdReturnValue. Последний вариант позволяет возвращать и результат в виде набора записей, который образуется при выполнении SQL-команды select. Параметр типа pdReturnValue автоматически связывается с объектом TDataSource, на который в свойстве DataSet ссылается объект TADOStoredProcedure. Это позволяет отображать полученный набор записей в сетке или с помощью любого другого визуального элемента управления.

Класс TADOCommand

Класс TADOCommand можно использовать для выполнения SQL-команд, которые не возвращают данных (для выполнения команд, возвращающих данные, следует использовать класс TADOQuery).

Настройка объекта класса TADOCommand выполняется точно так же, как и объектов прочих ADO-компонентов. Команда выполняется при вызове метода Execute(). Если в свойстве ExecuteOptions установлено одно из значений типа eoAsync, то продолжительный процесс выполнения команды может быть прерван пользователем. Для этого по команде пользователя должен вызываться метод Cancel(), прежде чем поступит сообщение CommandTimeout. Возможность прерывания продолжительного процесса выполнения команды — отличительная черта класса TADOCommand; выполнить такую операцию с помощью объекта TADOQuery невозможно. Хотя объект TADOQuery позволяет выполнять команды в асинхронном режиме и время от времени получает сообщения о протекании процесса выполнения команды, прервать его он не может.

С помощью объекта TADOCommand можно выполнять SQL-команды, которые возвращают набор записей. Но для этого объект TADOCommand должен быть связан с объектом TADODataSet. Связывание выполняется присвоением результата выполнения свойству Recordset объекта TADODataSet:

```
ADODataSet1->Recordset = ADOCommand->Execute();
```

Полученный результат через объект TDataSource может передаваться в любой визуальный компонент, позволяющий работать с данными.

Использование компонента TADODataset для доступа к набору записей

Компонент TADODataSet предназначен специально для выполнения SQL-выражений, возвращающих набор записей. В принципе он не обладает никакими преимуществами перед TADOQuery или TADOCommand. Единственное отличие — этот класс позволяет работать с функциями доступа к многоуровневым базам данных, которыми располагает Microsoft RDS (Remote Data Space). Для этого в свойство RDSConnection объекта TADODataSet устанавливается ссылка на объект TRDSConnection. Но не забудьте, что такое связывание несовместимо со связыванием через свойство Connection.

Управление транзакциями

Компонент TADOConnection используется и для управления транзакциями. Это происходит почти так же, как и при использовании BDE-компонентов, за исключением того, что несколько изменены имена методов.

- Метод BeginTrans() выполняет те же функции, что и метод StartTransaction() компонента TDatabase.
- Для подтверждения транзакции используется метод CommitTrans().
- Для отказа от транзакции используется метод RollbackTrans().

Обработка событий в ADO-компонентах

ADO-компоненты располагают множеством обработчиков событий, специфичных для выполняемых с их помощью операций. Компоненты этой группы поддерживают и обработку стандартного набора событий, используемых во всех классах, наследующих TDataSet.

Обработчики событий компонента TADOConnection

- OnWillConnect(). Извещает приложение, что согласие на подключение базы данных получено, но само подключение еще не завершено.
- OnConnectComplete(). Извещает приложение, что подключение к базе данных выполнено.
- OnInfoMessage(). Извещает приложение, что провайдер готов передать информационное сообщение. Это событие генерируется сразу же после выполнения подключения.
- OnBeginTransComplete(). Извещает приложение, что транзакция началась.
- OnCommitTransComplete(). Извещает приложение, что подтверждение транзакция воспринято.
- OnRollbackTransComplete(). Извещает приложение, что отказ от транзакции воспринят.
- OnWillExecute(). Извещает приложение, что переданная команда принята к выполнению, но процесс выполнения еще не начат. Обращаю ваше внимание на то, что это событие генерируется для всех компонентов, которые использует подключение через объект TADOConnection. Если же компонент использует собственный механизм подключения, то для такого объекта это событие генерируется только в том случае, если объект запрашивает выполнение команды. Поскольку класс TADOCommand не располагает собственными обработчиками событий, то единственная возможность отслеживать события, связанные с выполнением команд, запущенных с помощью объекта TADOCommand, — использовать обработчик OnWillExecute() объекта TADOConnection.

- `OnExecuteComplete()`. Извещает приложение, что выполнение команды завершено.
- `OnDisconnect()`. Извещает приложение, что завершен процесс разрыва соединения с базой данных.

Обработчики события компонента TADOCommand

В структуре компонента `TADOCommand` обработчики событий отсутствуют.

Обработчики события компонентов, производных от TADOCustomDataSet

- `OnEndOfRecordset()`. Извещает приложение, что достигнут конец набора записей.
- `OnFetchComplete()`. Извещает приложение, что асинхронно выполняемая SQL-команда завершила формирование набора записей.
- `OnFetchProgress()`. Периодически извещает приложение о состоянии продолжительного процесса выполнения SQL-команды в асинхронном режиме.
- `OnFieldChangeComplete()`. Извещает приложение о том, что произошло изменение соответствующего поля базы данных.
- `OnMoveComplete()`. Обрабатываемое событие генерируется в момент завершения перемещения курсора базы данных (этот обработчик события выполняет те же функции, что и обработчик `OnDataChange()` компонента `TDataSource`).
- `OnRecordChangeComplete()`. Извещает приложение, что изменение содержимого записи завершено.
- `OnRecordsetChangeComplete()`. Извещает приложение, что завершено изменение содержимого набора записей.
- `OnWillChangeField()`. Извещает приложение, что программа поддержки ADO готова к модификации содержимого *поля*, но операция еще не начата.
- `OnWillChangeRecord()`. Извещает приложение, что программа поддержки ADO готова к модификации содержимого *записи*, но операция еще не начата.
- `OnWillChangeRecordset()`. Извещает приложение, что программа поддержки ADO подготовлена к модификации содержимого *набора записей*, но операция еще не начата.
- `OnWillMove()`. Извещает приложение, что программа поддержки ADO готова к *перемещению курсора*, но операция еще не начата.

Создание прототипа БД-приложения

В большинстве БД-приложений пользователь в процессе работы с программой указывает, с какой базой данных или таблицей ей предстоит иметь дело. Методика построения такого типа приложений с использованием BDE-компонентов достаточно хорошо отработана и описана. Этого не скажешь о приложениях, в которых используются ADO-компоненты. Мы постараемся восполнить этот пробел.

Получение от пользователя строки подключения

В составе C++Builder 5 имеется недокументированная функция `PromptDataSource()`, которая позволяет вводить строку подключения. Объявление этой функции находится в файле `ADODB.hpp`:

```
WideString PromptDataSource
   (int theWindowHandle, WideString theOriginalConnectionString);
```

Эта функция выводит на экран диалоговое окно, в котором пользователь может сформировать строку подключения или отредактировать ту, которая передана параметром `theOriginalConnectionString`. Это такое же диалоговое окно, как и то, что выводится при редактировании значения свойства `ConnectionString` в процессе разработки программы.

Считывание имен таблиц базы данных

Метод `TADOConnection::GetTableNames()` выполняет те же функции, что и метод `TSession::GetTableNames()`. С помощью этого метода заполняется передаваемый ему список строк (этим списком может быть свойство `Items` объекта `TComboBox` или `TListBox`). Естественно, перед выбором таблицы нужно организовать подключение к базе данных.

Извлечение имен полей

При использование ADO-компонентов имена полей извлекаются по той же методике, что и при работе с BDE-компонентами. Нужно организовать цикл просмотра содержимого поля `Fields` объекта класса-наследника `TDataSet`:

```
for (int Index =0; Index < ADOTable->FieldCount; Index++)
     Field = ADOTable->Fields->Fields[Index ]->FieldName;
```

Другой вариант — извлечь список имен в переменную типа `TStringList`, вызвав метод `TADOConnection::GetFieldNames(TStringList *theList)`. Но учтите, что перед этим нужно выделить память под этот список либо воспользоваться списками элементов управления `TComboBox` или `TListBox`.

Извлечение имен хранимых процедур

Для извлечения имен хранимых процедур вызывается метод `TADOConnection::GetProcedureNames(TStringList *theList)`. Учтите, что перед вызовом метода нужно выделить память под указанный список либо воспользоваться списками элементов управления `TComboBox` или `TListBox`.

Настройка оптимальной производительности

Набор свойств ADO-компонентов позволяет разработчику выбрать оптимальный режим работы с базой данных. Оптимизация настройки связана с учетом особенностей структуры конкретной базы данных и приложения в целом. В частности по-разному нужно настраивать приложения, работающие в режиме клиент/сервер или с локальной базой данных. Ниже мы познакомим вас с тем, как те или иные варианты настройки влияют на производительность БД-приложения.

Работа с запросами или с таблицами

Работая с удаленными базами данных в режиме клиент/сервер, следует отдавать предпочтение запросам, которые выполняются на стороне сервера. Это связано с тем, что компоненты, работающие с полными таблицами, требуют передачи всех данных из таблицы на компьютер клиента, что приводит к нерациональной пересылке больших объемов информации по сети. Выбор в пользу запросов особенно важен в тех приложениях, где интенсивно используется фильтрация записей. При использовании запросов фильтрация выполняется на стороне сервера, а значит, объем информации, передаваемой по сети, существенно снижается, и скорость работы приложения возрастает.

Выбор места размещения курсора

Курсор определенного набора записей содержит всю информацию о текущем состоянии этого набора — о том, какая запись в нем является текущей, какие удаления или модификации данных выполнены и т.д. При работе с ADO-компонентами разработчик должен сам выбрать, на какой стороне — клиента или сервера — размещать курсор. Размещение курсора на стороне клиента может потребовать довольно больших затрат на начальной стадии, но если данные изменяются незначительно и пользователь работает с ними продолжительное время, такой выбор уменьшит загрузку сети в процессе обработки данных. В начале сеанса на компьютер клиента передается копия набора данных с компьютера сервера, и все последующие операции с ними выполняются на компьютере клиента, не загружая сеть. Только после завершения операций данные возвращаются на компьютер сервера, и происходит обновление информации в хранимых файлах базы данных. Такая же технология используется и при работе с BDE-компонентом `ClientDataSet`. Разница только в том, что в случае BDE-компонентов эта технология использовалась по умолчанию, и дополнительная настройка не требовалась. ADO-компоненты позволяют разработчику выбрать способ внесения изменения в набор записей. Если свойству `MarshalOptions` присвоить значение **`moMarshalModifiedOnly`**, то изменения будут накапливаться на стороне клиента до полного завершения сеанса работы с набором записей. Если же этому свойству присвоить значение **`moMarshalAll`**, то при каждом изменении набор записей будет копироваться на компьютер сервера.

Но при работе с большими динамическими наборами записей более эффективным является размещение курсора на стороне сервера. Это особенно сказывается в тех случаях, когда пользователь на стороне клиента фактически работает только с небольшим фрагментом этого набора данных. Если размещение курсора на стороне клиента позволяет работать с наборами в несколько тысяч записей, то размещение курсора на стороне сервера дает возможность оперировать миллионами записей. Учтите, что в документации рекомендуется на стороне сервера использовать только *однонаправленные* (unidirectional) курсоры типа `forward only`.

Типы курсоров

Существует несколько настроек типа курсора, которые ограничивают возможности оперирования ими. Эти настройки также сказываются на производительности приложения.

- Курсор типа `ctOpenForwardOnly` работает очень быстро, поскольку не требует никаких дополнительных средств для просмотра ранее обработанной порции данных. Такой курсор всегда выполняет просмотр с начала набора данных и перемещается только в одном направлении. Учтите, что курсор имеет статус “только для чтения”.
- Курсор типа `ctStatic` работает почти так же быстро, как и `ctOpenForwardOnly`, но позволяет перемещаться по набору записей в обоих направлениях. Курсор этого типа имеет статус “только для чтения” и изолирован от любых изменений, вносимых в базу данных другими пользователями.
- Курсор типа `ctKeyset` имеет статус “чтение/запись”, может перемещаться на любую позицию и в любом направлении, но изолирован от любых изменений, вносимых в базу данных другими пользователями.
- Курсор типа `ctDynamic` отличается от типа `ctKeyset` тем, что позволяет отслеживать в наборе записей все изменения, вносимые другими пользователями.

Буферизация

Размер локального кэш-буфера записей задается в свойстве `CacheSize`. Чем больше размер кэш-буфера, т.е. чем больше записей хранится в оперативной памяти компьютера, тем выше потенциальная производительность приложения. Но эта зависимость справедлива только в том случае, если пользователь работает в основном с одним и тем же набором записей. Если выполняются случайные перемещения по набору записей большого объема или последовательно просматриваются записи из длинной таблицы, то обновление кэш-буфера требует слишком больших расходов и сводит на нет все преимущества локальной буферизации. Выбор способа буферизации зависит и от того, где размещается курсор. Если он размещается на стороне клиента, то нет смысла организовывать кэш большого размера, поскольку набор данных все равно полностью хранится на компьютере клиента.

Обработка ошибок

Сложность в работе с сообщениями об ошибках, которые формируются ADO-компонентами, в основном связана с тем, что они маловразумительны. Очень часто вы можете встретить сообщения вроде **`Errors occurred`** (возникли ошибки). Другое сообщение может гласить, что обнаружено несоответствие типа поля, но при этом не указывается, о каком поле идет речь.

Правда, объект `TADOConnection` имеет свойство `Errors`, которое можно индексировать. Но и сообщения, которые можно получить с помощью этого свойства, также не отличаются большой информативностью. Во всяком случае, попробуйте воспользоваться этим свойством в своей программе. Для этого нужно организовать цикл доступа к отдельным сообщениям в свойстве `TADOConnection::Errors`:

```
for (int Index =0; Index < ADOConnection1->Errors->Count;
               Index++)
{
   String Message =
          ADOConnection1->Errors->Item[Index]->Description;
};
```

Более подробную информацию о свойстве `Error` можно найти в файле `$(BCB)\source\vcl\ADOInt.pas`. Здесь `$(BCB)` — стандартная системная переменная, в которой хранится имя корневого каталога C++Builder. Учтите, что примеры в этом файле написаны на языке Object Pascal (Delphi), а не на C++.

Многоуровневые приложения и ADO

ADO можно использовать в сочетании с разработанной в Microsoft многоуровневой технологией RDS (Remote Data Space). Для этого предназначен специальный компонент `TRDSConnection`, который применяется в сочетании с `TADODataSet`. Но обсуждение темы совместной работы ADO и RDS выходит за рамки этой книги. Читатели, которых эта тема заинтересует, могут получить подробную информации на Web-сервере Microsoft по адресу `http://msdn.microsoft.com/`. Архитектура удаленных модулей данных, разработанная фирмой Borland, также позволяет использовать ADO-компоненты. При этом для реализации многоуровневой архитектуры никаких специальных мер в отношении ADO-компонентов предпринимать не нужно. Такие компоненты можно использовать в модулях данных, предназначенных для работы со средствами поддержки распределенных объектов CORBA, MTS, ISAPI или ASP.

Извлечение данных в приложении

В этом разделе мы рассмотрим архитектуру подсистемы извлечения данных в БД-приложениях и имеющиеся для этого в составе C++Builder средства.

При проектировании такой подсистемы у разработчиков, как правило, возникает множество вопросов.

- Нужно ли использовать компонент `TSession`?
- Какой из компонентов — `TQuery`, `TTable` или специальный компонент, предназначенный для работы с определенной СУБД, — имеет смысл использовать для доступа к таблице?
- Какой из компонентов — `TQuery` или `TStoredProc` — лучше использовать для доступа к хранимым процедурам базы данных?
- Следует ли кэшировать обновления набора записей?
- Должен ли набор записей быть однонаправленным?
- Включать ли объекты компонентов доступа к данным непосредственно в состав класса экранной формы или лучше использовать модуль данных?
- Использовать ли фильтр в `TQuery` или изменять SQL-выражение и повторно извлекать данные?

Этот список, конечно же, нельзя считать исчерпывающим. Если в приложении нужно использовать информацию из множества таблиц или хранимых процедур, то способов организации взаимной работы компонентов становится еще больше.

Базовые решения

Давайте проанализируем основные возможности, которые предоставляет разработчику C++Builder. Мы не будем входить во все детали, поскольку они достаточно подробно освещены в сопроводительной документации.

Выбор между `TTable`, `TStoredProc` и `TQuery`

Преимуществом компонента `TTable` является возможность легко организовать доступ к информации из отдельной таблицы. Этот компонент обладает множеством разнообразных методов, но не отличается гибкостью. Хотя с его помощью можно работать и с SQL-базами данных, основное его назначение — работа с "родными" форматами BDE — xBase, CSV или Paradox.

Компонент `TStoredProc` хорошо справляется с функциями доступа к хранимым процедурам базы данных. С его помощью можно получить список хранимых процедур, но что более важно — информацию о параметрах выбранной процедуры. Напомню, что хранимые процедуры могут возвращать значения двумя способами:

- через набор записей, который содержит одну или более строк данных;
- через параметры с квалификатором `output`.

Компонент `TQuery` предоставляет разработчику наиболее широкие возможности, но с ним и сложнее всего работать. Этот компонент позволяет работать с SQL-выражениями, а потому обладает всей мощью и гибкостью этого языка. С его помощью можно получить доступ одновременно к нескольким таблицам или представлениям, извлекать из них информацию, обновлять, вставлять записи или удалять их. Компонент `TQuery` позволяет работать и с храни-

мыми процедурами базы данных. Более того, этот компонент позволяет обновлять данные, полученные с помощью хранимых процедур или из представлений, имеющих статус "только для чтения". Более подробно о таких уникальных возможностях `TQuery` будет рассказано в разделе *Использование кэш-буфера обновлений запроса*.

В БД-приложении можно использовать расширенные возможности, которыми обладает BDE. В частности, с помощью BDE можно предварительно считывать данные в кэш и выполнять однородное объединение таблиц из разных баз данных. Первая операция позволяет подготовить данные для заполнения элементов управления в экранной форме еще до того, как пользователь даст команду на переход к следующему фрагменту набора записей. Если, например, в экранной форме имеется элемент `TDBGrid`, в котором выведено пять записей, то следующие пять записей уже будут находиться в кэш-буфере и появятся на экране, как только пользователь щелкнет на соответствующей кнопке панели навигации. В кэш-буфер можно помещать и содержимое полей типа `BLOB` (такие поля обычно хранят тексты большого объема или изображения), а это также повышает эффективность работы приложения. *Однородное объединение* (*heterogeneous joins*) — это результат выполнения SQL-запроса к таблицам из нескольких баз данных, причем эти базы могут иметь различные форматы.

Свойство `CachedUpdates`

Несмотря на то что свойство `CachedUpdates` может принимать только одно из двух значений — **`true`** или **`false`**, — оно очень серьезно влияет на процесс обработки наборов записей. Если `CachedUpdates` имеет значение **`true`**, то не только выполняется кэширование всех извлеченных данных, но всех внесенных в эти данные изменений. Таким образом, процесс обработки данных разделяется на три этапа.

1. Данные обновляются в буфере записей приложения.
2. Изменения вносятся в локальный кэш-буфер в ходе выполнения транзакции.
3. Транзакция фиксируется в базе данных.

Но если кэширование обновлений требует дополнительной работы, зачем его использовать? Во-первых кэширование повышает производительность приложения, хотя и требует дополнительных ресурсов памяти. Во-вторых, как уже отмечалось в предыдущем разделе, кэширование позволяет редактировать данные из тех наборов записей, которые в обычном режиме работы имели бы статус "только для чтения". Это обычно происходит при выполнении запроса, который объединяет две таблицы. Другой случай — использование запроса, который вызывает хранимую процедуру. Как реализовать эту методику на практике, будет рассказано далее, в разделе *Использование кэш-буфера обновлений запроса*. Но хочу обратить ваше внимание на то, что при работе с транзакциями совсем необязательно использовать кэширование обновлений.

Поля просмотра

Элементы управления, применяемые для создания полей просмотра, например компонент `TDBLookupComboBox`, располагают средствами отбора данных в одной таблице на основе информации, взятой из другой таблицы. Как правило, они используются для связывания полей первичных идентификаторов с полями, хранящими осмысленный текст, который можно предъявить конечному пользователю. Например, если таблица рассылки содержит только идентификационные номера клиентов, то для конечного пользователя такие номера ничего не говорят — ему нужны имена клиентов. Поле просмотра и позволяет связать идентификационный номер клиента в таблице рассылки с его именем в таблице имен клиентов, а затем вывести имя в элемент управления экранной формы.

Поля просмотра позволяют выполнять те же операции, что и специализированные элементы управления, но они могут работать с элементами типа TDBGrid, а содержащиеся в них данные можно кэшировать.

Извлечение данных из нескольких источников

Очень часто БД-приложениям приходится работать с данными из нескольких источников — таблиц, представлений (view) или наборов записей, полученных с помощью хранимых процедур. Для извлечения данных из нескольких источников можно использовать разные подходы.

Использование кэш-буфера обновлений запроса

Этот способ рекомендуется использовать в тех ситуациях, когда данные из нескольких источников можно рассматривать как единую запись или когда нужно отредактировать результат выполнения хранимой процедуры, полученный в виде набора записей со статусом "только для чтения".

Поскольку компоненты TQuery используют SQL-выражения, с их помощью можно считывать данные из нескольких источников, используя SQL-оператор JOIN. Данные из разных источников объединяются, и их можно считать единой записью. Результат выполнения такого запроса получает статус "только для чтения". Однако если свойство CachedUpdates в объекте TQuery имеет значение **true** и в свойстве UpdateSQL установлена ссылка на объект класса TUpdateSQL, то такую запись можно редактировать. При этом должно быть соблюдено еще одно требование — в объекте TUpdateSQL должна быть задана одна из трех SQL-команд: INSERT, UPDATE или DELETE. При выполнении всех этих условий совместное использование объектов TQuery и TUpdateSQL позволяет обрабатывать такой набор записей несмотря на то, что он первоначально имел статус "только для чтения". Когда свойство CachedUpdates имеет значение **true**, данные изменяются в той копии набора записей, которая размещена в локальном кэш-буфере. Затем внесенные изменения фиксируются в базе данных на сервере в результате выполнения транзакции. Фактически изменение данных происходит с "черного хода" — через объект TUpdateSQL.

Связь „главный–подчиненный"

Использование связи "главный–подчиненный" (master–slave) — это, пожалуй, наиболее распространенный способ извлечения информации из нескольких источников данных. Такой способ идеально подходит при наличии в базе данных отношений типа "один-ко-многим". Этот способ не возбраняет и использование кэширования обновлений.

Реализуется этот способ следующим образом. Сначала обычным способом создается главный набор записей. Далее создается подчиненный объект TQuery, в SQL-выражении которого используется один или несколько параметров в операторе WHERE. О том, как в компоненте TQuery работать с параметрами (свойством Params), можно познакомиться в сопроводительной документации C++Builder. Обращаю ваше внимание на то, что параметры должны иметь такие же имена, как и соответствующие поля в созданном ранее главном наборе записей. Последней операцией является указание в свойстве DataSource объекта подчиненного набора записей ссылки на объект главного набора записей. Теперь при активизации запроса подчиненного набора записей объект этого набора сопоставит параметры запроса с полями главного набора записей и извлечет их значения. Как видите, все очень просто! Подчиненный запрос получает данные в динамическом режиме. Он будет автоматически обновляться при изменении позиции текущей записи в главном наборе, и оба набора будут синхронизированы.

Связи „главный — подчиненный — подчиненный“ и „главный — подчиненный/главный — подчиненный“

Если в отношения вовлечен еще и третий набор записей, то нужно решить, какой из первых двух будет играть роль главного для третьего набора данных. Если вы остановитесь на варианте организации связей “главный–подчиненный–подчиненный”, то свойства `DataSource` в объектах `TQuery` второго и третьего наборов записей должны содержать ссылку на объект `TQuery` первого, главного набора. Иными словами, данные для второго и третьего наборов будут извлекаться параллельно при перемещении текущей позиции в первом наборе, но оба подчиненных набора будут независимы друг от друга.

Второй вариант предполагает, что третий набор записей является подчиненным по отношению ко второму, который, в свою очередь, является подчиненным по отношению к первому набору. Для того чтобы организовать такую связь между наборами записей, нужно в свойстве `DataSource` в объекта `TQuery` третьего набора установить ссылку на объект `TQuery` второго набора. В результате данные для второго и третьего наборов будут извлекаться последовательно, причем данные в третьем наборе будут зависеть от данных во втором.

Связь „главный (только для чтения) — главный (чтение/запись) — подчиненный “

Такая организация связей между наборами используется реже, чем рассмотренные ранее варианты, но мы обсудим и такой способ, чтобы продемонстрировать, как в определенных ситуациях можно сберечь ресурсы, пользуясь возможностями компонентов C++Builder. Если речь идет о вычислительных ресурсах, то, пожалуй, самым расточительным будет вариант, когда к объекту `TDBGrid` подсоединяется набор записей, имеющий двунаправленный статус доступа. При этом не только расходуется память для хранения набора записей, но и BDE должен организовать кэш-буфер, вмещающий столько записей, сколько выводится на экран в полях элемента управления. Если экранная форма приложения занимает весь экран, причем все поле формы отдано в распоряжение объекта `TDBGrid`, то в нем может уместиться довольно внушительное количество записей. Если же среди прочих на экран выводятся и вычисляемые поля и поля просмотра, то для соответствующих записей также нужно предусмотреть место в кэш-буфере BDE.

Чтобы уменьшить потребность в ресурсах, нужно внести некоторые изменения в организацию наборов данных. Во-первых, набору данных, который присоединяется к элементу управления `TDBGrid`, нужно установить статус “только для чтения”. При этом, хотя BDE по-прежнему организует кэш-буфер для хранения выводимых записей, размер этого буфера оказывается существенно меньшим — чем больше записей умещается в сетке, тем экономнее расходуется память, по сравнению с предыдущим вариантом. Далее нужно организовать подчиненный набор записей, который по структуре будет идентичен главному, но в него будут помещаться только те записи, которые выделены пользователем на экране, как подлежащие редактированию (“живые” записи). Идея состоит в том, чтобы все операции редактирования выполнять именно с этим, усеченным по объему набором записей. Для его редактирования можно организовать отдельное всплывающее окно или вкладку главного окна приложения. В результате BDE будет использовать ресурсы только для работы с одной редактируемой записью.

Оба набора записей должны быть синхронизированы. Поэтому, после завершения редактирования в дополнительном окне, данные в главном окне должны быть обновлены.

Конструирование модулей данных

В составе C++Builder имеется специальный программный компонент ***Data Module Designer***, который выполняет при создании модуля данных те же функции, что и ***Form Designer*** по отношению к экранным формам. С его помощью можно размещать, отбирать, удалять или копировать объекты VCL-компонентов в модуле данных. Но прежде чем начать обсуждение методики работы с ***Data Module Designer***, рассмотрим, что представляет собой модуль данных и какую роль он играет в БД-приложении.

Назначение модуля данных в БД-приложении

Модули данных позволяют разработчику отделить программирование средств доступа к данным от программирования пользовательского интерфейса. Имеющиеся в составе C++Builder средства визуальной разработки модуля данных поддерживают ту же парадигму визуального программирования, что и при создании экранных форм.

Модуль данных представляет собой по сути специальный тип формы, в которую могут быть помещены любые компоненты, имеющиеся в составе VCL, в том числе и невизуальные. Чаще всего в модуль данных помещаются компоненты, специально предназначенные для работы с базами данных, такие как `TTable`, `Tquery` и `TDataSource`, но это отнюдь не означает, что в него нельзя помещать и другие компоненты.

При создании простых приложений, особенно на стадии освоения C++Builder, вполне достаточно ограничиться использованием невизуальных компонентов типа таймера, запросов, источников данных в экранной форме. Но использование этой методики для более сложных приложений приводит к тому, что экранная форма оказывается буквально набитой множеством компонентов, за которыми разработчику сложно уследить. Использование модулей данных позволяет выделить объекты невизуальных компонентов и скомпоновать их в отдельном модуле.

Кроме того, если вам приходится разрабатывать множество приложений, работающих с одной и той же или похожими базами данных, то такой модуль данных можно переносить из проекта в проект, не повторяя однажды сделанную работу. Модуль данных может инкапсулировать все функции анализа допустимости данных и поддержания целостности. Это имеет смысл делать даже при работе с СУБД, располагающими собственными средствами поддержания целостности. Если в приложении предполагается использовать MIDAS, то специальная форма модулей данных — удаленные модули данных (Remote Data Module — RDM) — могут играть в этой архитектуре роль среднего уровня многоуровневой распределенной системы.

И, наконец, в модуле данных отражается структура спроектированной базы данных. Если при проектировании структуры базы данных использовалась методика диаграмм взаимосвязей между объектами (Entity-Relationship Diagram — EDR) или другая аналогичная, то каждый модуль данных будет представлять отдельных объект этой диаграммы. Например, модуль данных `Order` на диаграмме, представленной на рис. 14.3, содержит таблицы заказов (`Table`), таблицы продуктов (`Product`) и таблицу статуса заказов (`Status`).

Такой стиль разработки модуля данных позволяет легко разобраться в обозначениях отношений, например

```
OrderStatus = Order->Table->FieldByName("Status")->AsString;
```

и повторно использовать фиксированные имена, как в приведенном ниже выражении:

```
OrderStatus = Order->Table->FieldByName("Status")->AsString;
AccountStatus =
    AccountStatus->Table->FieldByName("Status")->AsString;
```

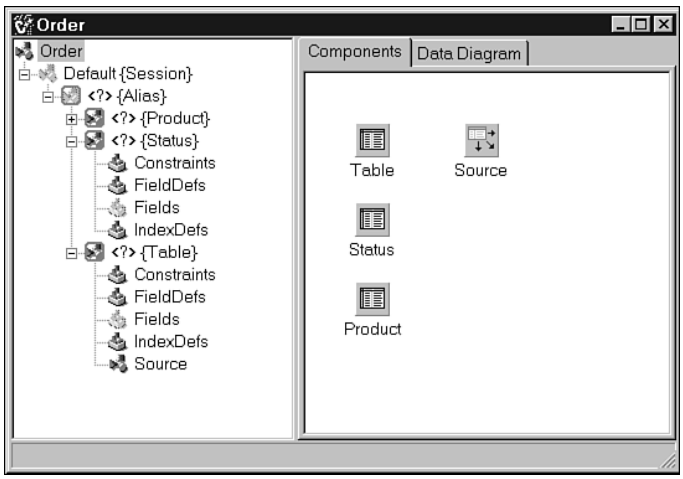


Рис. 14.3. Модуль данных `Order`

Использование модулей данных в приложениях, DLL, и распределенных объектах

Модули данных во многом напоминают модули экранных форм. Они имеют собственные заголовочные файлы с расширением `.h`, файлы реализации с расширением `.cpp` и файлы описания компонентов с расширением `.dfm`. Модули данных могут ссылаться на модули экранных форм и другие модули данных. Объекты модулей данных могут создаваться автоматически (соответствующую настройку можно выполнить в списке Project⇨Options или отредактировав список модулей проекта с помощью команды Project⇨View Source) или в процессе выполнения приложения с помощью оператора `new`, как это сделано в приведенном ниже выражении:

```
SomeDataModule = new TSomeDataModule(Application)
```

Объект модуля данных в таком случае имеет "владельца", который должен удалить его после окончания работы и освободить выделенную для него память. Как правило, владельцем объекта модуля данных является объект приложения. В этом случае объект модуля данных автоматически удаляется после завершения выполнения приложения. В качестве владельца объекта модуля данных может выступать и объект экранной формы (в этом случае объект формы автоматически удаляет объект модуль данных при закрытии формы), объект компонента (объект компонента автоматически удаляет объект модуль данных при закрытии объекта-владельца компонента). В качестве указателя на объект-владелец можно задать и `NULL`, но в этом случае программист должен самостоятельно написать программу закрытия объекта модуля данных.

Если предполагается использовать модуль данных в сочетании с экранной формой, то нужно выполнить следующее.

- Объект модуля данных должен быть создан до создания объекта экранной формы, который этот модуль использует. Это организуется изменением порядка создания модулей в списке настройки состава проекта или списка автоматического создания объектов. Естественно, если объект экранной формы не создается автоматически, его следует создать в программном коде с помощью оператора `new`, а после использования удалить.

- При разработке экранной формы, которая будет использовать модуль данных, вызовите команду File⇨Include Unit Hdr и укажите файл заголовка модуля данных. В результате этот файл будет указан в директиве `#included` в файле `.cpp` модуля экранной формы.

Использование модулей данных в сочетании с DLL-модулями, COM- или CORBA-объектами организуется так же.

Если предполагается использовать модуль данных в модуле динамически подгружаемой библиотеки DLL и нужно обеспечить, чтобы модуль данных оставался открытым все время, пока DLL будет загружен в память, то объект модуля данных нужно открыть в теле функции `DllEntryPoint()`. Программный код, представленный в листинге 14.1, показывает, как это выглядит на практике.

Листинг 14.1. Создание объекта модуля данных в DLL

```
#include <vcl.h>
#include <windows.h>
#pragma hdrstop
#include <Forms.hpp>
#include <TestDataModuleUnit.h>
#pragma argsused

String InternallyMaintainedResultString;
int WINAPI DllEntryPoint(HINSTANCE hinst,unsigned long reason,
        void* lpReserved)
{
   // Анализ глобальных переменных.
   if (TestDataModule ==NULL)
   {
      Application->Initialize();
      Application->CreateForm(
            __classid(TTestDataModule),&TestDataModule);
   };
   return 1;
}

char *__declspec(dllexport)TestDLLInterfaceFunction(void)
{
   TestDataModule->Query->First();
   InternallyMaintainedResultString =
      TestDataModule->Query->FieldByName("SUMMARY")->AsString;
   return InternallyMaintainedResultString.c_str();
}
```

Аналогично организуется открытие модуля данных и при работе с COM-объектами и CORBA-объектами. При работе с COM-объектами модуль создается функцией `DllEntryPoint()` сгенерированного объекта Active Server Library. При использовании CORBA-технологии сервер сам является модулем данных, поэтому использовать архитектуру модулей данных в сочетании с CORBA не представляет особого труда.

Структура модуля данных

Основной является такая структура модуля данных, которая содержит наборы записей и источники данных. Такой модуль данных позволяет инкапсулировать объекты базы данных, отслеживаемых полей, а также методы обработки событий, связанных с поддержанием целостности базы данных, изменением содержимого отслеживаемых полей, распространением изменений в связанных таблицах базы данных. Модуль данных, если в нем отсутствуют ссылки на определенные элементы управления, можно использовать в сочетании с любыми средствами пользовательского интерфейса. В большинстве случаев для подключения модуля данных к пользовательскому интерфейсу вполне достаточно визуальных компонентов работы с данными, которые имеются в состава VCL, но нужно быть готовым и к тому, что придется разработать собственный специальный компонент.

Модуль данных часто содержит объект класса, производного от `TCustomConnection`. Этот объект может использоваться компонентами данных как в этом же модуле, так и в других модулях данных.

Иногда в приложениях используются модули данных, которые компонентов работы с собственно базами данных не содержат. В состав таких модулей включаются невизуальные компоненты общего назначения, например таймер, специализированные невизуальные компоненты и т.п. Разработчики часто создают специализированные компоненты для доступа к системному реестру, последовательным или параллельным портам или другим аппаратным средствам. Если компоненты такого рода разместить в специализированном модуле данных, то они, как и компоненты доступа к данным, могут использоваться всеми модулями приложения.

Добавление свойств в класс модуля данных

Для включения в класс модуля данных нового экспортируемого (published) свойства нужно приложить определенные усилия. Лучше всего с этой целью создать специальный класс, производный от `TComponent`, с экспортируемыми свойствами, идентичными тем, которые планируется добавить в класс модуля данных. Таким компонентом можно манипулировать и присваивать значения его свойствам как на этапе разработки программы, так и во время ее выполнения. Обращение к этому компоненту осуществляется из обработчиков событий модуля данных во время работы программы или из объекта класса экранной формы, которому необходимо задать значения (в том числе и указатели на элементы управления) так, чтобы они были доступны и в модуле данных, причем при этом не нужно подключать объект экранной формы к объекту модуля данных.

Конечно, можно использовать для этой цели и общедоступные (public) методы, переменные или свойства, но в таком случае присваивать им значения можно только в процессе выполнения программы. Это чревато появлением дополнительных ошибок, которые приходится “отлавливать” в работающей программе. Например, в пользовательском интерфейсе приложения может потребоваться использовать элемент управления, который будет отображать ход выполнения процесса в модуле данных. На этапе разработки это можно сделать, только создав ссылку на модуль экранной формы в модуле данных, что потребует задания общедоступных переменных или свойств в программном коде.

Использование *Data Module Designer*

Диалоговое окно Data Module Designer разделено на две панели. Слева выводится древовидная структура компонентов модуля данных (наборов записей, источников данных и отслеживаемых полей). В правой части окна имеются две вкладки, одна из которых предназначена для вставки компонентов в модуль данных, а вторая — для формирования диаграммы структуры базы данных.

Вкладка Components окна Data Module Designer

При включении в состав модуля данных новых компонентов они перетаскиваются именно на вкладку Components диалогового окна Data Module Designer. При разработке модуля данных не пожалейте времени и скомпонуйте значки компонентов на этой вкладке таким образом, чтобы по ее внешнему виду было понятно, как компоненты функционально связаны друг с другом. Учтите, что в определенных ситуациях Data Module Designer может "реорганизовать" размещение значков компонентов на поле вкладки, добиваясь того, чтобы все они уместились в ее видимой области. Чтобы избежать такой "самовольной" реорганизации постарайтесь с самого начала выделить для модуля данных как можно больше места на экране.

В левой части окна выведена древовидная схема иерархии компонентов. Это позволяет просмотреть иерархические зависимости между объектами таких типов, как `TSession`, `TDatabase`, `TDataSet`, `TDataSource` и `TField` (рис. 14.4).

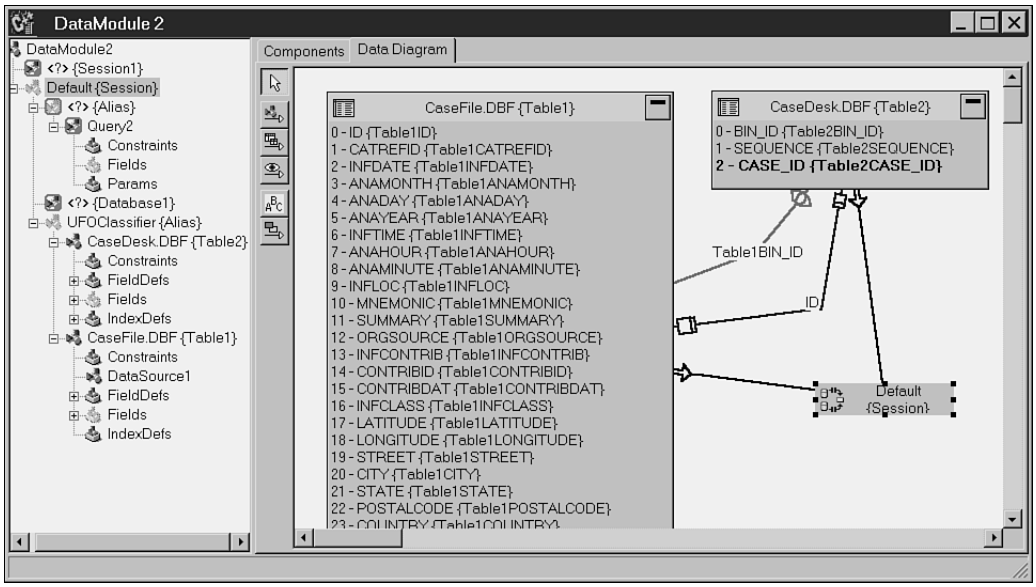


Рис. 14.4. Представление иерархии компонентов модуля данных в окне Data Module Designer

Видно, что объект `Table1` подключен к объекту `DataSource1` (поскольку узел `DataSource1` расположен в ветви узла `Table1`) и что ни один компонент не использует объект базы данных `Database1` и объект сеанса `Session1`.

Редактор структуры базы данных

Определенная в модуле данных структура базы данных отображается на вкладке Data Diagram. Обычно разработчики мало знакомы с этой вкладкой, поскольку информацию о ней не так просто найти в системе оперативной справки C++Builder. Найдите в предметном указателе системы оперативной справки раздел `Data Module Designer` и дважды щелкните на имени раздела — вы увидите перечень тем, касающихся работы с вкладкой Data Diagram.

На рис. 14.5 показано, как выглядит схема базы данных во вкладке Data Diagram. Вы видите три таблицы из тех, что представлены в древовидной структуре в левой части окна, список их полей и схему связей между таблицами. Там же представлен и объект `TSession`, который используется объектами таблиц.

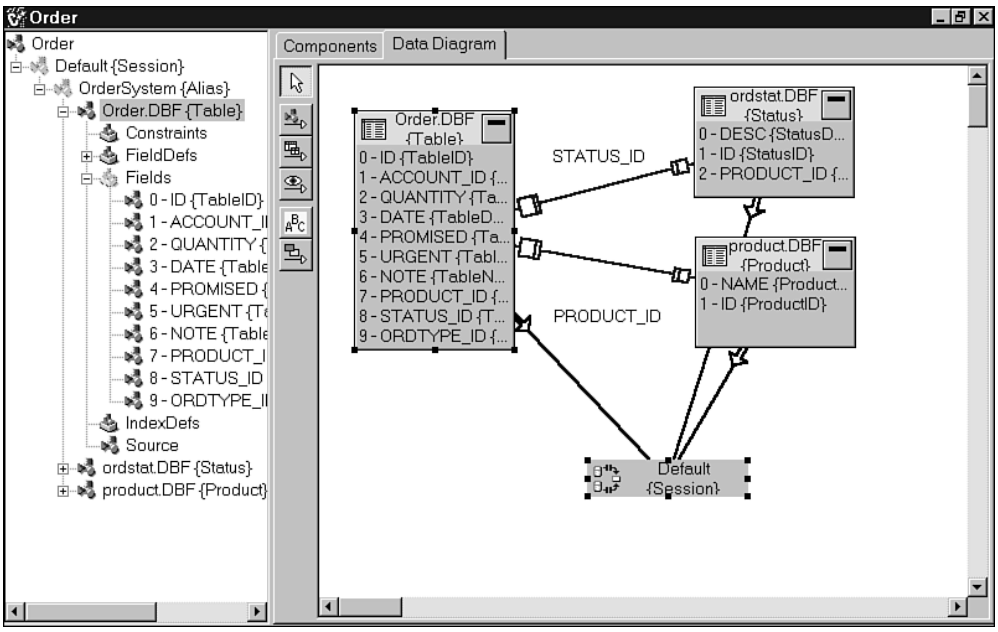


Рис. 14.5. Вкладка Data Diagram *диалогового окна программы* **Data Module Designer**

При изменении структуры компонентов в модуле данных — добавлении или удалении полей, изменении свойств объектов, подключении компонентов к новым источникам данных или удалении компонентов из модуля — все внесенные изменения сразу же отображаются на этой схеме, по крайней мере для тех компонентов, которые на ней изображены. Компоненты можно убрать из схемы с помощью команды Delete, а при щелчке правой кнопкой мыши на изображении какого-либо объекта открывается контекстное меню. В этом меню имеются не только стандартные команды копирования/удаления, но и команды вызова редактора полей в таблице или запуска SQL Explorer. При щелчке правой кнопкой мыши на изображении связи между компонентами также открывается контекстное меню, которое среди прочих содержит и команду удаления представленной связи.

Слева в верху вкладки Data Diagram расположены кнопки редактирования. Ниже приведено их описание в том порядке, как они изображены на вкладке, начиная с самой верхней.

- Select. Позволяет указать и выбрать объект на схеме.
- Property. Позволяет отобразить связь между свойством одного компонента и тем компонентом, на который это свойство ссылается. Если эта команда применяется по отношению к компонентам, связь между которыми отсутствует, то редактор автоматически устанавливает в указанном свойстве соответствующую ссылку. Процесс задания новой связи выглядит следующим образом: щелкните на кнопке Property а затем прочертите мышью линию между выбранными компонентами.
- Master/Detail. Позволяет организовать отношение “главный–подчиненный” между двумя наборами данных, представленными на схеме. Для организации такого отношения щелкните на кнопке Master/Detail а затем прочертите мышью линию от того набора данных, который будет в отношении играть роль главного, к объекту подчиненного набора данных. После этого на экране появится окно редактора полей, в котором нужно будет указать параметры отношения, в том числе и наименования связанных полей.

- Lookup. Позволяет установить поля просмотра (подстановки) в таблице. В такие поля подставляются данные из другой таблицы. Манипуляции при выполнении этой команды практически те же, что и при выполнении команды Master/Detail. Щелкните мышью на поле, в котором нужно выполнить подстановку, и прочертите мышью линию к таблице, из которой будут выбираться данные для подстановки. После этого на экране появится окно редактора полей просмотра Lookup, в котором нужно будет указать наименования связанных полей.
- Comment. С помощью этой команды можно добавить комментарий к любому компоненту схемы базы данных. Для редактирования текста комментария нужно дважды щелкнуть на изображении выбранного компонента и ввести в поле редактирования новый текст. Если после этого щелкнуть где-нибудь в стороне от поля редактирования, то введенный текст будет зафиксирован, а если редактирование завершается нажатием клавиши <Esc>, то сохранится прежний текст.
- Comment Allude. Эта команда позволяет подключить введенный ранее текст комментария к любому компоненту структуры базы данных. Таким образом, один и тот же текст комментария может относиться к разным компонентам, причем количество таких "пользователей" не ограничивается. Выполняется команда следующим образом: прочертите мышью линию между выбранным компонентом и полем комментария.

К сожалению, изображение на поле вкладки Data Diagram нельзя масштабировать. Поэтому, если база данных состоит из множества компонентов, все их сразу увидеть на экране не удастся — придется пользоваться прокруткой. Но всю схему целиком можно вывести на печать.

Учтите, что сформированная схема не наследуется в производном классе модуля данных. Поэтому, если вы будете создавать класс модуля данных, производный от существующего, придется воссоздать в нем эту схему.

Методика работы с модулями данных

Выше были описаны только базовые операции при работе с модулями данных. Помимо них существует еще множество методик, которые можно использовать в БД-приложениях, включающих такие модули. Ниже приведен краткий перечень более сложных методик.

- Инкапсуляция бизнес-правил приложения в специализированных объектах и обработчиках событий.
- Разделение модулей данных и средств пользовательского интерфейса.
- Использование механизма наследования классов модулей данных для организации иерархической системы классов связанных модулей данных.
- Использования специальных средств поддержки целостности ссылочных связей между компонентами класса-наследника и класса-родителя модуля данных.
- Использование модулей данных в пакетах.

Наследование в системе классов модулей данных

Использование в практике разработки БД-приложений наследования в системе классов модулей данных очень напоминает методику использования наследования в системе классов экранных форм. Включите разработанный класс модуля данных в состав хранилища Object Repository — после этого его можно использовать в новом проекте. Для включения такого модуля в новый проект воспользуйтесь уже неоднократно описанной в этой книге методикой. Выберите команду File⇨New. После этого, если вы собираетесь просто использовать соз-

данный ранее модуль данных в новом проекте, то выберите переключатель Use; если же собираетесь включить в проект его копию, которую можно будет редактировать, — выберите переключатель Copy, а если намереваетесь создать класс-наследник, то выберите переключатель Inherit. Диалоговое окно New Items, которое открывается с помощью команды File⇨New, показано на рис. 14.6.

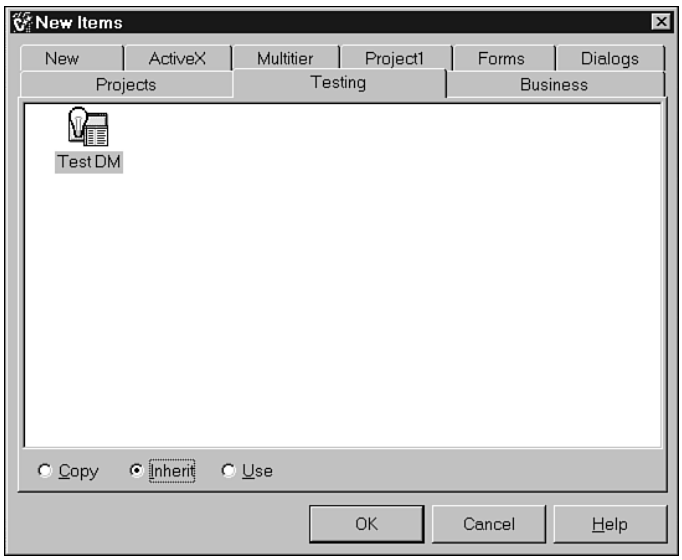


Рис. 14.6. Включение класса модуля данных в новый проект

Из этих трех вариантов, наиболее привлекательным является вариант наследования класса модуля данных. В классе-наследнике можно использовать дополнительные компоненты, добавлять поля, обработчики событий, модифицировать обработчики событий или свойства компонентов. Но учтите, что удалить компонент, наследованный от класса-родителя, нельзя.

Наследование класса модуля данных используется в следующих целях.

- Механизм наследования позволяет в базовом классе избежать подключения к определенному набору записей. Для этого в базовом классе создаются только связи с объектами источников данных. Подключение же объектов источников данных к конкретных наборам записей выполняется в производном классе. Эту методику можно использовать, организовав в базовом модуле поддержку как ADO-, так и BDE-компонентов и подключив в производном классе наборы записей именно к тем компонентам, которые нужно использовать в приложении. Учтите, что в базовом классе должны быть созданы члены-обработчики событий, которые можно будет вызывать в производном классе при обработке событий обработчиками, созданными в производном классе.
- Добавление в модуль данных функций, которые будут поддерживать спецификацию семейства баз данных. Все основные функции можно включить в базовый класс, а в классах наследниках включить дополнительные компоненты и реализовать специфические функции, нужные для определенной модификации. В частности, в производный класс можно включить дополнительные таблицы, поля или запросы.
- На производные классы удобно возлагать функции редактирования и просмотра данных, сохранив всю структуру базы данных в базовом классе. В производном классе

можно организовать специальные алгоритмы проверки корректности данных в отслеживаемых полях, выполнение специфических операций при изменении значений в этих полях, подстановку данных в поля просмотра из других таблиц.

Особенности использования механизма наследования при работе с модулями данных

Использование механизма наследования при работе с модулями данных имеет определенную специфику.

Если вы собираетесь использовать класс модуля данных, производный от класса, в котором имеются ссылки на компоненты базового класса, то нужно позаботиться о том, чтобы при выполнении программы система отыскивала ссылки на компоненты объекта производного класса. Как это организовать, показано в программе, приведенной в листинге 14.2. В этом фрагменте `Base1` и `Base2` — базовые классы, а `Descendant1` и `Descendant2` — производные классы.

Листинг 14.2. Поиск объекта унаследованного класса модуля данных в процессе выполнения программы

```
TFindGlobalComponent OldFindGlobalComponent;

TComponent*__fastcall FindGlobalComponentExtended(
       const System::AnsiString Name)
{
   if (Name == "Base1") return(TComponent *)(Base1 *)Descendant1;
   if (Name == "Base2") return(TComponent *)(Base2 *)Descendant2;
   return OldFindGlobalComponent(Name);
}

WINAPI WinMain(HINSTANCE,HINSTANCE,LPSTR,int)
{
   try
   {
      Application->Initialize();
      OldFindGlobalComponent =FindGlobalComponent;
      FindGlobalComponent =FindGlobalComponentExtended;
```

Кроме того, каждый конструктор в классе модуля данных (базовом или производном) должен устанавливать глобальную переменную модуля данных, поскольку в классе-родителе автоматическая инициализация такой переменной не выполняется. Например, в классе `Base1` должен присутствовать оператор

```
Base1 = this;
```

В производном классе `Descendant1` должен быть оператор

```
Descendant1 = this;
```

Как избежать зависимости от специфики интерфейса пользователя

Существует две методики создания модулей данных, не зависящих от специфики интерфейса пользователя. Чаще всего в классе модуля данных с этой целью применяются специальные элементы управления, предназначенные для работы с данными. Считается,

что такая методика наиболее безопасна для приложения и не таит в себе "подводных камней". Вторая методика, которую нужно использовать очень осмотрительно, предполагает включение в определение класса модуля данных специальных членов — переменных, функций или свойств. Затем средства реализации интерфейса пользователя — элементы управления в экранной форме — подключаются к этим членам по ссылке. Для того чтобы гарантировать нормальную работу объекта модуля данных при использовании этой методики, нужно присваивать в конструкторе класса всем этим указателям исходное значение **NULL**, а при дальнейшем использовании обязательно проверять, не равен ли такой указатель **NULL**. Строгое соблюдение такой методики обеспечит нормальную работу модуля данных даже в том случае, когда средства пользовательского интерфейса в приложении не содержат того или иного элемента управления.

Совместная работа модуля данных со специализированными компонентами

Еще один нюанс в использовании модулей данных связан с наличием в приложении компонентов, созданных специально для используемого приложения. Как правило, обработчики событий таких компонентов инкапсулируют бизнес-правила приложения.

Иногда в роли таких компонентов выступают родовые классы из состава VCL или библиотек, приобретенных у сторонних разработчиков. Логика работы приложения, в частности обновление определенных наборов записей после ввода каких-либо связанных с ним данных, реализуется обработчиками событий — чаще всего обработчиком `AfterPost()`.

Но можно создавать и такие специализированные компоненты, которые располагают собственным набором обработчиков. Пример такого необычного обработчика, который позволяет выполнять откат, представлен в листинге 14.3.

Листинг 14.3. Объявление класса специализированного компонента

```
class TUndoableFramework:public TComponent
{
public:
   void __fastcall (__closure *)TExecuteHandler
        (TUndoableFramework *theUndoableFramework);
   void __fastcall (__closure *)TRecordUndoHandler
        (TUndoableFramework *theUndoableFramework);
   void __fastcall (__closure *)TUndoHandler
        (TUndoableFramework *theUndoableFramework);
   void __fastcall (__closure *)TCleanupUndoHandler
        (TUndoableFramework *theUndoableFramework);
private:
   TExecuteHandler myToExecute;
   TRecordUndoHandler myToRecordUndo;
   TUndoHandler myToUndo;
   TCleanupUndoHandler myToCleanupUndo;
protected:
   virtual void __fastcall DoExecute(void);
   virtual void __fastcall DoRecordUndo(void);
   virtual void __fastcall DoUndo(void);
   virtual void __fastcall DoCleanupUndo(void);
public:
   virtual void __fastcall TUndoableFramework(void);
```

```
   virtual void __fastcall ~TUndoableFramework(void);
   virtual void __fastcall Execute(void);
     // Также вызывает ToRecordUndo
   virtual void __fastcall Undo(void);
   virtual void __fastcall CleanupUndo(void);
__published:
   __property TRecordUndoHandler ToRecordUndo =
         {read=myToRecordUndo,write=myToRecordUndo};
   __property TUndoHandler ToUndo =
         {read=myToUndo,write=myToUndo};
   __property TCleanupUndoHandler ToCleanupUndo =
         {read=myToCleanupUndo,write=myToCleanupUndo};
};
```

Использование таких компонентов в модуле данных упрощает создание приложения, в котором можно выполнять откат после внесения изменений в данные. Классы-наследники такого компонента могут включать методы, которые будут обращаться к специфицированным в базовом классе обработчикам при выполнении операций, сохранении информации на случай возможного отката, выполнения отката, очистки полей и т.д. В приложении можно использовать и сочетание разных технологий — например, в классе-наследнике компонента можно реализовать стек событий, в котором фиксируются обработанные ранее события, а класс модуля данных может включать специальные обработчики для выполнения операций с этим стеком.

В листинге 14.4 приведен фрагмент определения класса специализированного невизуального компонента, на который возлагается резервирование ресурсов. В этом фрагменте часть программного кода опущена, что отмечено троеточиями.

Листинг 14.4. Объявление класса невизуального компонента резервирования

```
class TReservationDesk:public TComponent
{
// Здесь пропущены объявления стандартных членов.
.........
public:
void __fastcall MakeReservation
       (TReservation *theReservation);
void __fastcall UpdateReservation
(TReservation *theReservation);
void __fastcall RemoveReservation
(TReservation *theReservation);
__published:
__property TQuery *Reservations =
       {read=myReservations,write=myReservations};
__property TAfterReservationAdded =
       {read=myAfterReservationAdded,write=myAfterReservationAdded};
__property TBeforeReservationUpdated BeforeReservationUpdated...
__property TAfterReservationUpdated...
__property TBeforeReservationRemoved...
__property TAfterReservationRemoved...
};
```

В БД-приложении можно использовать несколько объектов этого класса для разных таблиц резервирования. Конечно, можно продолжить специализацию инкапсулированных в нем функций в производных классах и создать в приложении собственную иерархию таких специализированных компонентов.

Использование модулей данных в составе пакета

Модули данных, так же как и модули экранных форм, можно включать в состав пакетов. Однако при этом часто возникает проблема с использованием модификатора пакетов: модуль данных почему-то преобразуется в обычный модуль экранной формы, и все особенности его структуры, сформированные с помощью ***Data Module Designer***, исчезают. Эту проблему можно решить, включив в программный код следующее определение:

```
class PACKAGE TMyDataModule;
class TMyDataModule...
```

Набор компонентов InterBase Express

В этом разделе вы познакомитесь с реляционной СУБД ***InterBase***, которая является сервером баз данных класса "клиент/сервер", и набором компонентов ***InterBase Express*** (IBX), которые входят в комплект программного продукта C++Builder 5.

В версию C++Builder 5 включен набор компонентов IBX, которые позволяют создавать мощные приложения класса "клиент/сервер", причем без использования ядра BDE и соответствующих BDE-компонентов. Эти приложения способны работать напрямую с клиентами InterBase. Хотя большинство IBX-компонентов имеют "собратьев" в наборе BDE-компонентов, методика их использования имеет свои нюансы, связанные с реализацией концепции "клиент/сервер" в InterBase.

Чтобы продемонстрировать, как работать с IBX-компонентами, в этом разделе будет описана разработка несложного БД-приложения `Bug Tracker`, в составе которого используются разные типы компонентов этого набора. Приложение предназначено для сопровождения базы данных программных проектов и их версий, а также сохранения информации об ошибках, обнаруженных в программах. Тексты программ этого приложения имеются на прилагаемом компакт-диске в папке `BugTracker`, имя файла проекта — `BugTracker.bpr`.

На заметку

Если у вас возникла идея модернизировать проекты, созданные с помощью C++Builder 4 или еще более ранних версий этой среды, оснастив их функциональными возможностями, доступными в InterBase, обратите внимание на набор компонентов InterBase Objects, созданный Джейсоном Уортоном (Jason Wharton). Информацию о нем можно получить на Web-сервере по адресу `//www.ibobjects.com`. В составе этого программного продукта предлагается и замена SQL-диалекту ISQL — диалект IB-WISQL, который поддерживает многие функции структурного синтеза и административного сопровождения баз данных InterBase.

Структура базы данных приложения `Bug Tracker`

В базе данных `Bug Tracker` имеется три таблицы: `Program`, `Revision` и `Bugs`. На рис. 14.7 показана схема связей между таблицами в диалоговом окне Database Design Studio. Табли-

ца Program является главной, а таблицы Revision и Bugs — подчиненными. Как видно на схеме, таблица Bugs связана и с Program, и с Revision, но связь Program является *обязательной*, а связь с Revision — *желательной*. Задание этих отношений при создании базы данных позволяет СУБД InterBase отслеживать ссылочную целостность базы данных на сервере. В листинге 14.5 представлен SQL-код создания таблицы Bugs.

Листинг 14.5. SQL-код создания таблицы Bugs

```
/*Bugs Table*/
create table bugs (
   bug_id integer not null,
   bug_name varchar (80)not null,
   bug_description varchar (255),
   bug_resolved smallint not null,
   bug_date date not null,
   pro_id integer not null,
   r_id integer,
   primary key (bug_id),
constraint fk_bugs_pro_id foreign key (pro_id)
        references program (pro_id),
constraint fk_bugs_r_id foreign key (r_id)
        references revision (r_id));
```

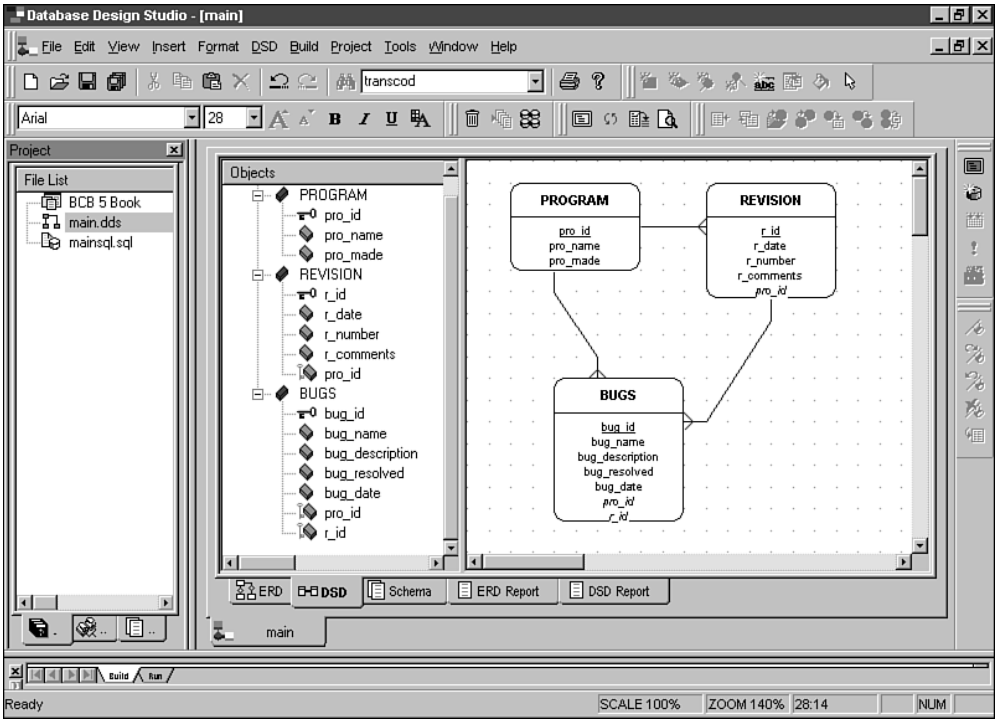


Рис. 14.7. Структура базы данных Bug Tracker

На заметку

В конец каждого объявления ограничений можно добавить выражения `on update (cascade||set null)` и `on delete (cascade||set null)`. Эти выражения определяют, какие операции выполняются с таблицей, когда выбранное поле в связанной таблице изменяется или удаляется. Включение этих выражений в определение ограничений ускоряет процесс обновления базы данных при изменении указанных полей, поскольку внесение изменений производится без запуска соответствующих триггеров.

Если, например, в определении имеется такая строка ограничения

```
constraint fk_bugs_pro_id foreign key (pro_id)references program
↳ (pro_id) on update cascade on delete cascade,
```

то при удалении записи в таблице `Program` или изменении значения поля `pro_id` в записи таблицы `Program`, связанная с ней запись в таблице `Bugs` будет автоматически удалена или модифицирована.

Вторичные (foreign) ключи можно определять, не присваивая им имени в SQL-выражении, а передоверить именование ключей СУБД InterBase. Но учтите — при этом "закладываются" определенные сложности последующей отладки такой базы данных. В смысле сообщения `violation of FOREIGN KEY constraint "FK_BUGS_PRO_ID" on table "BUGS"` (нарушение ограничения FK_BUGS_PRO_ID таблицы BUGS) разобраться гораздо легче, чем в смысле сообщения `constraint "INTEG_XX"` (ограничение INTEG_XX). К сожалению, первичным ключам таблиц назначать имена по собственному выбору нельзя.

На заметку

База данных `BCB5BOOK6.GDB`, которая имеется на прилагаемом к книге компакт-диске, разработана в СУБД InterBase 6.0, и с ней нельзя работать, используя более ранние версии InterBase.

В структуре таблицы `Bugs` имеются поля описания ошибок и поля вторичных ключей. Для каждого поля определены имя, тип, длина и индикатор и указано, должно ли полю присваиваться определенное значение. Если указано, что поле не должно иметь значения `null` (в выражении имеется квалификатор `not null`), InterBase не позволит ввести в таблицу запись, в которой это поле не определено.

Если это необходимо по логике работы БД-приложения, можно определить вторичный ключ таким образом, что InterBase будет автоматически поддерживать ссылочную целостность базы данных. В таблице `Bug` имеется два вторичных ключа: с помощью одного создается отношение с таблицей `Program`, а с помощью другого — отношение с таблицей `Revision`.

На заметку

При проектировании структуры базы данных рекомендуется использовать программу с графическим интерфейсом, например ***Database Design Studio*** (информацию о ней можно найти в сети Web по адресу `//www.chillisoft.com`). Эта и другие подобные ей инструментальные программы представляют разработчику наглядную схему отношений между таблицами и автоматически преобразуют результаты манипулирования графическими инструментами формирования схемы отношений в последовательность SQL-выражений.

СУБД InterBase способна автоматически отслеживать ссылочную целостность базы данных. В простенькой программе, которую мы сейчас рассматриваем, запись в таблице `Bug` ссылается на запись в таблице `Program`. Если будет предпринята попытка удалить запись в таблице `Program`, то появится предупреждающее сообщение `Violation of FOREIGN KEY constraint "INTEG_16" on table "BUGS"` (Нарушение ограничения вторичного ключа "INTEG_16" в таблице BUGS). Кроме того, при попытке вставить новую запись в таблицы `Bugs` или `Revision`, если в этой записи используется определенное значение поля `pro_id`, будет выведено сообщение об ошибке.

В таблице Revision вторичный ключ r_id объявлен таким образом, что он может содержать значение null. Если же полю r_id будет присвоено значение, отличное от null, то это значение должно удовлетворять заданным в определении ограничениям. В противном случае при попытке вставить запись появится сообщение об ошибке нарушения ограничений целостности.

Правила ссылочной целостности (точнее, средства проверки соблюдения этих правил) защищают базу данных от ошибок, которые могут поступать с данными от клиентских приложений. Чтобы воспользоваться такими средствами защиты в полной мере, очень важно не пожалеть времени и усилий в процессе проектирования и тестирования структуры базы данных. После того как база данных "заживет" собственной жизнью и наполнится данными, будет очень трудно что-либо скорректировать, не повредив уже имеющейся в базе информации. Предположим, что в недавно завершенном и сданном заказчику БД-приложении понадобилось изменить параметры первичного ключа одной из таблиц. Пока в базе данных нет информации, такая модификация займет несколько минут. Но если таблицы уже наполнены данными, понадобится затратить много часов, изобретая такую SQL-программу, которая внесла бы изменения в структуру, но не испортила уже имеющейся в базе информации.

Бизнес-правила базы данных

Бизнес-правила базы данных отражают взаимные связи между значениями полей в таблицах, которые существуют помимо системы отношений между таблицами. СУБД InterBase поддерживает реализацию таких бизнес-правил на стороне сервера независимо от клиентских приложений. Это одно из самых важных преимуществ архитектуры "клиент/сервер". Кроме того, InterBase позволяет изменять бизнес-правила на стороне сервера "на ходу", ничего не изменяя в клиентских приложениях. Пусть, например, вы пришли к выводу, что удаление записи в таблице Revision должно приводить к обнулению поля bug-resolved в соответствующей записи таблицы BUG. Для отслеживания соблюдения такого правила создается специальная программа, получившая в теории баз данных наименование триггер (trigger), которая автоматически выполняет необходимые модификации при наступлении указанного события.

В настоящее время не существует общепринятого соглашения, где именно следует размещать средства реализации бизнес-правил. Имеется множество доводов как в пользу размещения их на стороне сервера, так и на стороне клиента. Я рекомендую размещать на стороне сервера средства поддержки правил, которые связаны с механизмом обеспечения ссылочной целостности или затрагивают значительное число записей в таблицах базы данных. Средства поддержки прочих правил можете размещать по своему усмотрению.

Генераторы, триггеры и хранимые процедуры

В СУБД InterBase для создания программных средств поддержки бизнес-правил на стороне сервера можно использовать такие объекты как *генераторы*, *триггеры* и *хранимые процедуры*. Их используют для выполнения операций любой сложности — от простого наращивания значения какого-либо поля в отдельной записи до выполнения довольно сложных процедур над несколькими таблицами и тысячами записей одновременно. Поскольку все эти программные элементы интегрированы в базу данных на стороне сервера, их выполнение происходит значительно быстрее, чем в том случае, если бы они находились на стороне клиента.

Генераторы

Генераторы в СУБД можно рассматривать как своего рода глобальные переменные базы данных. Они не связаны непосредственно ни с какой отдельной таблицей или полем, а формируют последовательные числовые значения при вызове СУБД-функции GEN_ID(). Сами по

себе генераторы не используются, но к ним часто обращаются триггеры и хранимые процедуры при необходимости получить значения уникальных ключей.

Ниже представлена небольшая SQL-программа создания и начальной настройки генератора:

```
CREATE GENERATOR PRO_ID_GEN;
SET GENERATOR PRO_ID_GEN TO 1;
```

Эта программа создает генератор `PRO_ID_GEN` и устанавливает ему начальное значение `1`. Такой генератор можно использовать при формировании значения уникального ключа для новых записей таблицы `Program`.

Очень важно не забывать, что генераторы не должны вызываться в процессе транзакций. Если, например, после начала транзакции вызывается функция `GEN_ID()`, то откат транзакции не сможет вернуть прежнее значение генератора.

Триггеры

Триггер в InterBase аналогичен обработчику событий в классе C++. Назначение триггера — выполнить определенные операции при изменении таблицы или представления — вставке, удалении или обновлении записей.

Триггеры используются для реализации различных задач — поддержания ссылочной целостности, проверки корректности введенной информации ведения журнала учета действий пользователей, имеющих отношение к базе данных. Очень часто триггеры используются для автоматического наращивания значений полей (создания так называемых *автоинкрементных* полей). В СУБД InterBase не существует специального типа автоинкрементного поля — оно создается как сочетание обычного целочисленного поля и триггера. Как это выглядит в тексте программы на языке SQL, показано в листинг 14.6. В этом фрагменте используется ранее созданный генератор `PRO_ID_GEN`.

Листинг 14.6. SQL-программа создания триггера для таблицы `Program`

```
CREATE TRIGGER SET_PRO_ID FOR PROGRAM
BEFORE INSERT AS
BEGIN
   IF (NEW.PRO_ID IS NULL) THEN
   BEGIN
      NEW.PRO_ID = GEN_ID(PRO_ID_GEN,1);
   END
   IF (NEW.PRO_MADE IS NULL) THEN
   BEGIN
      NEW.PRO_MADE = 'TODAY';
   END
END
```

Этот триггер присоединяется к таблице `Program`, и заданные в нем операции выполняются перед вставкой записи в таблицу. Переменная `NEW` содержит вставляемые поля. Если значение поля `pro_id` не задано, вызывается функция `GEN_ID()`, которая увеличивает на 1 значение указанного в ней генератора и возвращает полученное значение. Затем полученное значение присваивается полю `pro_id` новой записи. Если явно не задано значение поля `pro_made`, триггер присваивает этому полю в новой записи значение текущей даты.

Есть один нюанс в использовании созданных таким способом автоинкрементных полей в сочетании с компонентами C++Builder. Дело в том, что IBX-компоненты не получают уведомления о том, что триггер СУБД InterBase изменил значение поля. Чтобы пользователь

смог увидеть в экранной форме, отражающей состояние записей, внесенные изменения, нужно, чтобы данные в элементах управления были каким-то образом обновлены. Существуют определенные методики, которые позволяют справиться с этой проблемой, но важно учитывать, что автоматически изображение в экранной форме БД-приложения не обновляется.

Хранимые процедуры

Хранимые процедуры — это SQL-программы. которые сохраняются в самой базе данных. Хранимые процедуры могут возвращать какие угодно значения — от значений базовых типов (целые, вещественные и т.д.) до наборов записей, включающих данные из разных таблиц. Например, в листинге 14.7 показан текст хранимой процедуры, которая получает имя созданной программы, формирует новую запись в таблице с соответствующей информацией и возвращает значение поля `pro_id`. В тексте процедуры содержится и обращение к генератору `PRO_ID_GEN`.

Листинг 14.7. SQL-текст определения хранимой процедуры `Create_Program`

```
CREATE PROCEDURE CREATE_PROGRAM /* имя процедуры */
(THE_PRO_NAME CHAR(80))/* входные параметры */
RETURNS (THE_PRO_ID INTEGER)/*возвращаемый параметр */
AS
BEGIN
   /* Формирование идентификатора новой программы */
   /* с помощью генератора */
   THE_PRO_ID = GEN_ID(PRO_ID_GEN,1);
    /* Вставка новой записи в таблицу PROGRAM */
   INSERT INTO PROGRAM(PRO_ID,PRO_NAME,PRO_MADE)
        VALUES(:THE_PRO_ID,:THE_PRO_NAME,'TODAY');
END
```

Эта процедура выполняет те же операции, что и рассмотренный ранее триггер, но, кроме того, еще и возвращает значение поля `PRO_ID` новой записи. Как и триггер, эта процедура сохраняется в базе данных на сервере (потому она и называется *хранимой*) и вносимые в тело процедуры изменения не требуют никакой модификации клиентских приложений. В приложении `Bug Tracker` будут использованы и триггеры, и хранимые процедуры.

На заметку

Иногда возникает необходимость извлечь значение генератора с помощью запроса к базе данных. Для этого можно воспользоваться следующим SQL-выражением:

```
SELECT GEN_ID(PRO_ID_GEN,1) NEXT_PRO_ID FROM RDB$DATABASE
```

При выполнении такого запроса объектом компонента `TIBQuery`, будет возвращено значение поля `NEXT_PRO_ID`, которое содержит очередное значение генератора `PRO_ID_GEN`.

Вариант, альтернативный обращению к хранимой процедуре, основан на использовании свойства `ForcedRefresh` IBX-компонентов. С помощью этого свойства несложно получить информацию об изменениях, внесенных в базу данных в результате активизации триггера. Основная роль использования свойства `ForcedRefresh` состоит в том, что клиентская программа должна располагать информацией о вторичном ключе, и при подстановке в оператор `WHERE` выражения `RefreshSQL` должен использоваться вторичный ключ вместо сформированного первичного. Об этом более подробно рассказано в разделе *Модификация, удаление, вставка и обновление* этой главы.

Как правило, клиентскому приложению запрещено изменять первичные ключи. Желательно модифицировать в клиентском приложении только вторичные ключи.

Реализация приложения Bug Tracker

Определившись со структурой базы данных, можно приступать к интеграции ее в приложение Bug Tracker. Разработку приложения начнем со средств, обеспечивающих создание новой базы данных или подключение к существующей. Затем добавим компоненты, необходимые для взаимодействия пользователя с размещенными на сервере данными.

Приложение Bug Tracker, файлы которого вы найдете на прилагаемом к книге компакт-диске, разработано в среде C++Builder 5 с использованием набора компонентов IBX 4.1. Если на вашем компьютере установлена одна из предшествующих версий среды или набора компонентов, то при загрузке проекта обнаружится исчезновение некоторых свойств. Обновленную версию набора компонентов можно найти в Internet по адресу **www.interbase.com**.

Отладка приложений, работающих с СУБД InterBase

Взаимодействие приложения с InterBase-сервером базы данных включает выполнение таких операций, как подключение к источникам информации, выполнение транзакций и отслеживание запросов со стороны клиента.

Отлаживать выполнение этих операций с помощью обычных средств среды C++Builder довольно трудно. Поэтому в состав набора IBX включен специальный отладочный компонент TIBSQLMonitor. Этот компонент позволяет отслеживать события, связанные с любой операцией InterBase-сервера, специфицированной в свойстве TraceFlags объекта компонента TIBDatabase. Если информация, сопровождающая такое событие, накапливается в объекте TMemo, то в распоряжении разработчика оказывается достаточно удобный инструмент протоколирования операций InterBase-сервера. Эта информация позволяет разобраться в причинах проблем, возникших в процессе выполнения множества транзакции и запросов.

Создание базы данных и подключение к базе данных

При создании базы данных используется компонент TIBDatabase. Метод CreateDatabase() этого компонента обращается к хранящейся в свойстве DatabaseName информации для определения сервера используемой базы данных и параметров, характеризующих зарегистрированного пользователя, — его имени в системе и пароля. Ниже представлен набор параметров, который используется на начальном этапе создания базы данных:

```
user "SYSDBA"
password "masterkey"
page_size 4096
```

Этот же компонент используется и для организации подключения к базе данных InterBase. При этом можно размещать сервер на локальном хосте и применять протоколы TCP/IP, NetBEUI или SPX. Но при подключении к уже существующей базе данных нужно передавать параметры, несколько отличные от тех, что используются при создании базы данных. Сервер и база данных, к которой организуется подключение, специфицируются в свойстве DatabaseName. В оперативной справке C++Builder приведена информация о форматировании этих параметров в DatabaseName для разных протоколов. В листинге 14.8 показано, как упростить процесс подключения с использованием разных протоколов при помощи перечислимого типа и функции, формирующей строку в нужном формате.

Листинг 14.8. Функция `CreateConnectionString()`

```
enum ConnectionType{ ctLocal, ctTCPIP, ctNetBEUI, ctSPX};
AnsiString __fastcall TdmMain::CreateConnectionString(
   AnsiString Server,.AnsiString FileName, ConnectionType CType)
{
   AnsiString ConnectionString ="";
   switch (CType)
   {
      case ctLocal:
           ConnectionString = FileName; break;
      case ctTCPIP:
           ConnectionString.sprintf("%s:%s", Server, FileName);
           break;
      case ctNetBEUI:
           ConnectionString.sprintf("\\%s\%s", Server, FileName);
           break;
      case ctSPX:
           ConnectionString.sprintf("%s@%s", Server, FileName);
           break;
   }
   return ConnectionString;
}
```

Иногда при подключении к СУБД InterBase с помощью протокола TCP/IP приходится использовать имя сервера вместо действительного IP-адреса. Например, спецификация `ibserver:c:\bugger.gdb` будет воспринята системой нормально, а `192.168.0.9:c:\bugger.gdb` иногда не воспринимается несмотря на то, что `192.168.0.9` является действительным адресом компьютера `ibserver`. Обычно это происходит в том случае, когда используется более ранняя, чем 2.0, версия Winsock. Чтобы устранить этот недостаток, добавьте в файл `hosts` новый элемент. В операционных системах Windows 95/98 этот файл размещается в каталоге `Windows`, а в Windows NT/2000 — в каталоге `%SystemRoot%\System32\Drivers\Etc`. Например, если вы собираетесь подключиться к серверу с IP-адресом `192.168.0.11`, добавьте в указанный файл строку: **`192.168.0.11 theserver`**. После сохранения отредактированного файла перезагрузите компьютер и запустите из командной строки DOS команду: **`nbtstat -R`**. Обращаю ваше внимание на то, что параметр **`-R`** нужно вводить прописными буквами.

В свойстве `DatabaseName` объекта `TIBDatabase` нужно указать системное имя пользователя и пароль. Это можно сделать двумя способами.

Можно присвоить свойству `LoginPrompt` значение **`true`**, тогда на экран будет выводиться диалоговое окно, в котором пользователю предлагается ввести нужную информацию.

Второй способ — в тексте программы присвоить нужные значения свойству `Params`. Ниже приведен фрагмент программного кода, в котором показано, как устанавливаются параметры регистрации пользователя:

```
user_name=SYSDBA
password=masterkey
```

Если параметры регистрации заданы, то установка значения **`true`** в свойстве `Connected` запускает процесс подключения к базе данных.

На заметку Если обнаруживаются проблемы с формированием строки подключения и заданием параметров в процессе выполнения программы, воспользуйтесь редактором компонента TIBDatabase.

Хотя с помощью InterBase Express можно создавать и переносить базы данных в модуль данных, этот набор компонентов в настоящее время не поддерживает выполнение SQL-программ (сценариев). Поэтому невозможно, запустив такую программу, создать с ее помощью в новой базе данных таблицы, триггеры и другие компоненты. Но на Web-сервере http://www.interbase.com можно найти компоненты, которые позволяют это сделать. При разработке приложения Bug Tracker считается, что база данных и все ее компоненты уже созданы ранее средствами самой СУБД.

Использование транзакций

В составе набора InterBase Express имеется специальный компонент TIBTransaction, который предназначен для управления транзакциями. Каждый объект TIBTransaction обрабатывает одну транзакцию с базой данных.

Свойства транзакции сохраняются в свойстве Params объекта TIBTransaction. Компонент оснащен встроенным редактором, который позволяет разработчику задавать четыре набора установок транзакции. В приложении Bug Tracker используется предопределенный набор Read Committed. По умолчанию в InterBase и IBX используется предопределенный набор Snapshot. Изменение заданного по умолчанию набора на Read Committed — это, как правило, первый этап настройки нового объекта TIBTransaction после его помещения в модуль данных или экранную форму.

Доступ к информации

При использовании IBX-компонентов в распоряжении разработчика имеется несколько способов доступа к информации, хранящейся в базе данных. Для этого можно использовать компоненты типов TIBTable, TIBQuery и TIBDataSet. Если вы остановились на компонентах TIBTable или TIBQuery, то нужно использовать и компонент TIBUpdateSQL.

Компонент TIBUpdateSQL

Компонент TIBUpdateSQL разработан специально для работы "в связке" с компонентами TIBTable или TIBQuery. Он позволяет выполнять выражения на языке SQL при удалении, вставке, модификации или обновлении записей.

Компонент оснащен собственным редактором, который упрощает процесс формирования SQL-выражения (рис. 14.8). Редактор можно использовать после того, как указатель на объект TIBUpdateSQL помещен в свойство UpdateObject объекта TIBTable или TIBQuery, а последний настроен на работу с определенной таблицей или запросом.

Редактор автоматически формирует базовую форму выражения для выполнения одной из четырех перечисленных выше операций, но чаще всего эту форму приходится использовать в качестве заготовки и затем модифицировать с учетом применяемых триггеров, процедур и т.п. Компонент TIBUpdateSQL предоставляет разработчику два набора параметров SQL-выражения. Первый набор включает имена полей выбранной таблицы или запроса (например, :PRO_ID). Во втором наборе добавлен префикс OLD к именам полей (например, :OLD_PRO_ID), и этот набор представляет значения полей в том виде, какими они были перед обновлением кэш-буфера.

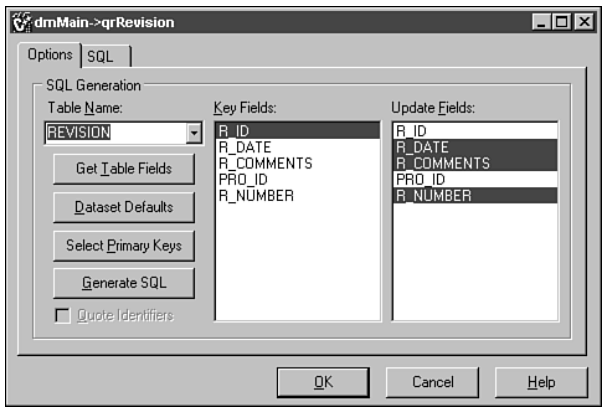


Рис. 14.8. Диалоговое окно встроенного редактора компонента `TIBUpdateSQL`

Компонент `TIBTable`

"Собрат" стандартного компонента C++Builder для доступа к базам данных, компонент `TIBTable` предназначен для формирования редактируемого набора записей на основе информации из заданной таблицы или представления (view).

Объект `TIBTable` должен формировать редактируемый набор записей, но при определенных обстоятельствах это не удается, особенно при работе с представлениями. В таком случае объект `TIBTable` может воспользоваться услугами объекта `TIBUpdateSQL`, который позволяет вносить изменения в набор записей через механизм выполнения SQL-выражений.

Нужно сказать, что компонент `TIBTable` не очень хорошо адаптирован к сетевой среде приложений типа "клиент/сервер". В процессе работы этого компонента выполняется в полтора-два раза больше пересылок информации по сети, чем требуется для выполнения аналогичного запроса `select *from <table>` компонентам `TIBQuery` или `TIBDataset`.

Не рекомендуется пользоваться в программе методом `Append()` компонента `TIBTable`. При работе с базами данных типа "клиент/сервер" добавление записи в конец таблицы не имеет никакого смысла. Это — "рудимент" методики работы с локальными базами данных, например Paradox. Базы данных на SQL-сервере распределены по множеству страниц и очень редко имеют непрерывную структуру, как это принято в локальных базах данных. Поэтому не следует ожидать, что после выполнения `Append` запись будет помещена в конец таблицы — СУБД InterBase может поместить ее куда угодно. Хотя с помощью метода `FetchAll()` объект `TIBTable` может выбрать все записи из таблицы в буфер на клиентской стороне и затем поместить новую запись в конец этого буфера, такая операция настолько загружает сеть, что теряет всякий смысл.

Компонент `TIBQuery`

С помощью компонента `TIBQuery` можно выполнять любые SQL-запросы к серверу InterBase. Результатом выполнения запроса будет набор записей, имеющий статус "только для чтения". Запросы могут выбирать информацию из нескольких связанных таблиц, что является весомым преимуществом компонента `TIBQuery`, по сравнению с `TIBTable`. Этот компонент хорошо согласуется с парадигмой клиент/сервер СУБД InterBase. Если приложение ориентировано на работу с удаленным InterBase-сервером, то компоненту `TIBQuery` следует отдать предпочтение перед `TIBTable`.

Если с помощью компонента TIBQuery планируется выполнять операции вставки, редактирования или удаления данных, то к нему следует подключить объект TIBUpdateSQL. Но в таком случае лучше использовать компонент TIBDataSet, который сочетает в себе возможности TIBQuery и TIBUpdateSQL.

Компонент TIBDataSet

Компонент TIBDataSet сочетает функции отбора данных TIBQuery с функциями редактирования TIBUpdateSQL. Отличие между способом спецификации запросов в TIBDataset и TIBQuery состоит в том, что свойство Params в TIBDataset имеет тип не TParams, а TIBXSQLDA. Такой тип параметров требуется для работы с InterBase версии 6.0, использующей параметры Int64. В приложении Bug Tracker объекты компонента TIBDataSet используются довольно широко.

Компоненты TIBSQL и TIBStoredProc

Компонент TIBSQL располагает большим набором методов выполнения операций с текущими данными СУБД InterBase, которые выполняются чрезвычайно быстро. Этот компонент является производным от TComponent и в нем не поддерживаются операции с базами данных; его основное назначение — быстрый доступ к переменным и генераторам СУБД.

Хотя компонент TIBStoredProc и может работать с таблицами базы данных, основное его назначение — обращение к хранимым процедурам. С помощью этого компонента можно организовать обращение к выбранным хранимым процедурам и передачу им необходимых параметров. Учтите, что любая информация, возвращаемая после выполнения хранимой процедурой, будь то простая переменная или набор из тысяч записей, доступна только через соответствующие параметры процедуры.

Компонент TIBEvents

При использовании хранимых процедур и триггеров на стороне сервера InterBase можно заказать генерацию сервером соответствующих событий. Например, командой POST_EVENT 'BUG_DELETE' задается генерация события BUG_DELETE.

Компонент TIBEvent позволяет клиентскому приложению известить сервер о том, о каких событиях оно хочет получить уведомление от сервера, получить такое уведомление и вызвать обработчик OnEventAlert(). В приложении Bug Tracker таким способом организован вызов системной функции ShowMessage(), когда на стороне сервера удаляется запись в таблице Bug. Но при желании, конечно же, можно вызывать и гораздо более сложные функции.

Установка приложения Bug Tracker

IBX-компоненты упрощают доступ к данным и их модификацию в базе данных InterBase, но их настройка на этапе разработки приложения должна выполняться очень тщательно. При правильной настройке компонентов конечный пользователь избавлен от необходимости выполнения рутинных операций — поддержании ссылочной целостности базы данных, модификации определенных полей, выполнения транзакций и т.п. Приложение Bug Tracker разработано таким образом, чтобы изолировать пользователя от этих нюансов и предоставить ему простой и понятный интерфейс.

Начнем процедуру установки с объекта компонента TIBDataSet, который должен работать с таблицей Program. Сначала перетащим компонент в модуль данных, подключим его к объекту TIBDatabase и активному объекту TIBTransaction (это упростит процесс установки). После этого нужно специфицировать SQL-выражение select. Поскольку компонент уже связан с функционирующим подключением к базе данных, можно использовать редактор CommandText и с его помощью создать нужное выражение — в нашем случае:

```
select * from PROGRAM
```

Модификация, удаление, вставка и обновление

После того как задана информация, подлежащая выводу на экран с помощью компонента, нужно указать, что будет делать компонент при выполнении модификации, удаления, вставки и обновления. Выше мы уже отмечали, что компонент TIBUpdateSQL оснащен средствами формирования базовых SQL-выражений. Но созданные таким образом базовые выражения нужно внимательно просмотреть и подкорректировать. Они не должны оставлять пользователю н малейшего шанса ввести некорректную информацию в базу данных. Ниже показан SQL-код для команды modify, который подготовлен IBX-компонентом.

```
update PROGRAM
set
   PRO_ID = :PRO_ID,
   PRO_NAME = :PRO_NAME,
   PRO_MADE = :PRO_MADE
where
   PRO_ID = :OLD_PRO_ID
```

Теоретически это выражение позволяет пользователю изменить поле pro_id, которое является главным ключом таблицы Program. Этого допускать никак нельзя, поэтому оператор PRO_ID =: PRO_ID нужно удалить из выражения.

Нужно отредактировать и базовое выражение для команды insert. Выше уже говорилось о том, что в поле pro_id новой записи таблицы Program нужно установить значение, сформированное генератором pro_id_gen. Именно эта операция и включается в модифицированное SQL-выражение:

```
insert into PROGRAM
   (PRO_ID,PRO_NAME,PRO_MADE)
values
   ((select GEN_ID(PRO_ID_GEN,1)FROM RDB$DATABASE),:PRO_NAME,
:PRO_MADE)
```

В этом выражении вообще можно было бы обойтись без установки значения поля pro_id, поскольку эта операция дублируется триггером BEFORE INSERT, связанным с таблицей Program.

Выражение для оператора delete, созданное компонентом по умолчанию, вполне "устраивает" приложение Bug Tracker, но выражение для оператора refresh нужно изменить. Компонент подготовит такое базовое выражение:

```
select
   PRO_ID,
   PRO_NAME,
   PRO_MADE
from PROGRAM
where
   PRO_ID = :PRO_ID
```

При выполнении этого выражения могут возникнуть проблемы, поскольку значение pro_id неизвестно на стороне клиента в то время, когда пользователь вставляет запись. Поэтому в сетке на экране это поле останется пустым, если использовать базовые формы выражений для операторов insert и refresh.

Решить проблему позволяет использование вторичного ключа, в данном случае — поля pro_name. Выражение refresh select должно вернуть только одну запись. Можно таким

образом изменить выражение refresh, чтобы в нем для отбора использовалось совпадение значения поля pro_name, а не pro_id. Это становится возможным благодаря тому, что в базе данных специфицировано ограничение, обеспечивающее уникальность поля pro_name. Механизм отслеживания этого ограничения гарантирует, что запрос вернет только одну запись — ту, которая соответствует значению pro_name, известному на стороне клиента. Для того чтобы использовать такую методику на этапе спецификации схемы базы данных, нужно задать соответствующие ограничения. Например, при создании таблицы Revision должно быть специфицировано следующее ограничение:

```
add constraint con_rev_pro_id_r_number unique (pro_id,r_number)
```

Это позволит извлекать только одну запись из таблицы при использовании в операторе select следующего условия отбора:

```
where
   PRO_ID = :PRO_ID
   AND R_NUMBER = :R_NUMBER
```

Поля

Поля базы данных задаются таким же способом, как и при работе с "обычным" компонентом TQuery. Можно, однако, несколько изменить операции с полем pro_id. Выражение Set Required to false позволит пользователю вставлять запись, не задавая при этом определенного значения для поля pro_id. Выражение Set ReadOnly to true пресечет попытку пользователя изменить сгенерированное значение. В приложении Bug Tracker такого рода коррективы внесены в определения полей таблиц Revision и Bugs.

Сохранение изменений в кэш-буфере

Компоненты из набора InterBase Express поддерживают сохранение внесенных изменений в кэш-буфере. Для этого в структуре большинства компонентов имеется свойство CachedUpdates. Использование механизма сохранения изменений в кэш-буфере позволяет вносить изменения в копию набора записей на стороне клиента, не обновляя (до поры до времени) информации в базе данных на сервере. Обновленные данные будут переданы на сервер и внесены в базу данных только после вызова метода ApplyUpdates(). В приложении Bug Tracker временное сохранение обновлений в кэш-буфере не используется, а все изменения передаются на сервер через механизм транзакций.

Выполнение транзакций

В листинге 14.9 представлена функция, которую следует вызывать в обработчике события AfterPost() компонента TIBUpdateSQL для того, чтобы привести в соответствие данные на стороне клиента и на стороне сервера. Этот метод создан с использованием компонента qrProgram, и его можно применять в сочетании с компонентами TIBUpdateSQL или TIBQuery. Для того чтобы информация на экране обновлялась после завершения передачи, нужно установить значение true в свойстве ForcedRefresh соответствующего объекта набора записей.

Листинг 14.9. Обработка события AfterPost

```
void __fastcall TdmMain::qrProgramAfterPost(TDataSet *DataSet)
{
try
```

```
{
   trMain->CommitRetaining();
}catch (Exception &E)
   // Если возникнет ошибка, то будет выполнен откат
   // транзакции и повторный запуск.
{
   trMain->RollbackRetaining();
   ShowMessage("Error commiting changes." + E.Message);
}
}
```

В клиентском приложении всегда имеется активный объект транзакции. Если в данные, которые связаны с этим объектом, вносятся изменения, то выполняется транзакция, и изменения фиксируются в базе данных на стороне сервера. Немедленно после этого объект снова становится активным и ожидает внесения дальнейших изменений. Такая организация процесса предотвращает затор, причиной которого может стать попытка изменения одной и той же записи двумя пользователями. После того как произошло изменение в наборе данных, который используется еще одним пользователем (например, набор `qrBugs` использует `qrRevision` в качестве источника данных для подстановки), вызывается метод `qrProgramAfterPost()`, и данные обновляются обработчиком события `AfterPost()`.

```
void __fastcall TdmMain::qrRevisionAfterPost(TDataSet *DataSet)
{
   qrProgramAfterPost(DataSet);
   if (!qrBugs->IsEmpty()){
       qrBugs->Refresh();
   }
}
```

Выполнение приложения `Bug Tracker`

На рис. 14.9 показано, как выглядит экранная форма приложения `Bug Tracker` в процессе выполнения. Приложение подключено к серверу СУБД InterBase и позволяет просматривать и редактировать информацию о программах, поддерживая при этом ссылочную целостность базы данных.

Резюме

Вы убедились в том, что в составе C++Builder имеется большой набор средств для создания БД-приложений самого различного назначения и архитектуры. В заключение хотелось бы напомнить основные особенности C++Builder, которые позволяют говорить об этой среде как о мощном и перспективном инструменте в руках разработчиков приложений для баз данных.

- Поддержка разных вариантов архитектуры приложения — одноуровневой, многоуровневой и “клиент/сервер”.

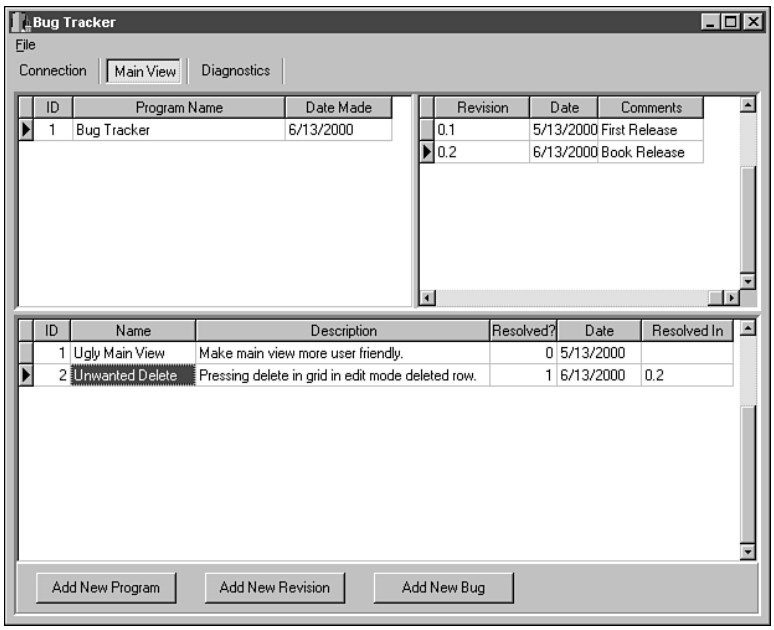


Рис. 14.9. Диалоговое окно приложения `Bug Tracker`

- Наличие средств создания специальных программных объектов — модулей данных.
- Большой выбор классов компонентов работы с базами данных — как универсальных (ориентированных на использование BDE), так и специализированных для работы с определенной СУБД (в частности, СУБД InterBase).

Думаем, вы согласитесь, что C++Builder — идеальная платформа для разработки широкого спектра БД-приложений.

Предметный указатель

M

О

П

Р