А. Б. Григорьев

bhv

О чем не пишут в книгах по Delphi

+cd

Виртуальный клуб программистов

Интеграция Windows API и VCL
Использование Windows Sockets API в Delphi
Неочевидные особенности целых чисел, вещественных чисел, строк
Коллекция коварных ошибок
Разбор и вычисление арифметических выражений в Delphi
Особенности различных версий от Delphi 3 до Delphi 2007

А. Б. Григорьев

О чем не пишут в книгах по Delphi

Санкт-Петербург
«БХВ-Петербург»
2008

УДК 681.3.068+800.92Delphi
ББК 32.973.26-018.1
Г83

Григорьев А. Б.

Г83 О чем не пишут в книгах по Delphi. — СПб.: БХВ-Петербург, 2008. — 576 с.: ил. + CD-ROM

ISBN 978-5-9775-0190-3

Рассмотрены малоосвещенные вопросы программирования в Delphi. Описаны методы интеграции VCL и API. Показаны внутренние механизмы VCL и приведены примеры вмешательства в эти механизмы. Рассмотрено использование сокетов в Delphi: различные режимы их работы, особенности для протоколов TCP и UDP и др. Большое внимание уделено разбору ситуаций возникновения ошибок и получения неверных результатов в "простом и правильном" коде. Отдельно рассмотрены особенности работы с целыми, вещественными и строковыми типами данных, а также приведены примеры неверных результатов, связанных с ошибками компилятора, VCL и др. Для каждой из таких ситуаций предложены методы решения проблемы. Подробно рассмотрен синтаксический анализ в Delphi на примере арифметических выражений. Многочисленные примеры составлены с учетом различных версий: от Delphi 3 до Delphi 2007. Прилагаемый компакт-диск содержит примеры из книги.

Для программистов

УДК 681.3.068+800.92Delphi
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Леонид Кочин*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Дизайн обложки	*Инны Тачиной*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 11.12.07.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 46,44.
Тираж 2500 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.02.953.Д.006421.11.04 от 11.11.2004 г. выдано Федеральной службой по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 978-5-9775-0190-3

Отпечатано с готовых диапозитивов
в ОАО "Техническая книга"
190005, Санкт-Петербург, Измайловский пр., 29

Отпечатано с готовых диапозитивов
в СПб ГУП "Петроцентр" ОП "Пушкинская типография"
196601, Санкт-Петербург, г. Пушкин, ул. Средняя, 3/8

Отпечатано с готовых диапозитивов в ФГУП "Печатный двор"
Министерства Российской Федерации по делам печати,
телерадиовещания и средств массовых коммуникаций.
197110, Санкт-Петербург, Чкаловский пр., 15.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

Отпечатано с готовых диапозитивов
в ГППО "Псковская областная типография"
180007, г. Псков, Рижский пр., 17

Отпечатано по технологии CtP
в ОАО «Печатный двор» им. А. М. Горького
197110, Санкт-Петербург, Чкаловский пр., 15.

Оглавление

Введение

Среда программирования Delphi заслуженно приобрела свою популярность. Этот удобный и красивый инструмент, основанный на не менее красивом языке Паскаль, первым избавил программиста от рутинных операций, сохранив при этом гибкость, присущую универсальным языкам программирования. Не удивительно, что об этой среде написано много книг, а в Интернете можно найти множество различных сайтов, посвященных Delphi.

Автор данной книги с 1999 года является постоянным посетителем (а с 2004 года — еще и модератором) сайта "Королевство Delphi" (**www.delphikingdom.ru**; подробнее этот сайт описан в *приложении 1*), одного из самых известных русскоязычных ресурсов по Delphi. За это время изучено, какие вопросы интересуют посетителей сайта и какие ответы они хотели бы получить. Наблюдения показывают, что существует целый ряд тем, традиционно вызывающих большой интерес, но информацию по которым найти сложно. Авторы книг по Delphi стараются как можно быстрее перейти к описанию возможностей различных библиотек, оставляя в стороне другие важные вопросы.

"За бортом" остается множество тем, касающихся более низкоуровневых средств, которые в большинстве своем неплохо документированы и описаны в книгах, проиллюстрированы готовыми примерами, которые, правда, обычно даются на C++. Конечно, немного опыта — и код переведен с C++ на Delphi. Только обидно программировать в Delphi, никак не задействуя ее высокоуровневые библиотеки — хочется как-то "сшить" гибкость низкоуровневых вещей и удобство библиотек, использовав библиотеки везде, где это возможно, а низкоуровневый код — только там, где без него никак не обойтись. И вот как раз по этим вопросам и наблюдается острая нехватка информации.

Природа не терпит пустоты, поэтому в Интернете время от времени появляются рекомендации по поводу того, как же все-таки "впрячь в одну упряжку" библиотеки и низкоуровневый код. Но эти советы, за редким исключением,

страдают тем, что дают готовое решение с минимальными объяснениями, почему надо делать это именно так (иногда авторы этих рекомендаций при всем желании не смогли бы дать такое объяснение, т. к. сами дошли до этого "методом тыка"). Так что такие советы могут оказаться очень полезными в конкретной ситуации, но не дают общего понимания проблемы.

Эта книга призвана заполнить информационный вакуум по некоторым из таких вопросов. Но самое главное — это то, что мы здесь принципиально будем избегать изложения в стиле "делай так, и будет тебе счастье", т. е. уклон будет не в сторону готовых рецептов, а в сторону объяснения, как это все устроено, чтобы читатель потом сам был в состоянии искать решения для своих проблем. Для этого мы будем разбирать стандартные средства Delphi с позиции "а как оно все работает".

Глава 1 книги посвящена интеграции библиотеки VCL и Windows API, прежде всего той части Windows API, которая управляет окнами на экране. Не секрет, что в VCL отсутствуют многие возможности по управлению окнами, которые предоставляет операционная система Windows, но их можно задействовать и в VCL-приложении, если знать, как и куда можно вставить код, использующий API, так, чтобы это не нарушило работу VCL. В данной главе даются базовые сведения о Windows API и о том, как VCL использует API. Здесь приведен ряд примеров программ различной степени сложности, которые демонстрируют методы применения Windows API совместно с VCL в различных ситуациях. Особое внимание уделяется тому, как использовать Windows API, не нарушая работу VCL.

Глава 2 посвящена применению сокетов в Delphi. Сокеты — это самые низкоуровневые сетевые средства Windows, своего рода ассемблер сетевого программирования. Они хорошо документированы, но из-за обилия возможностей эта документация оказывается практически "неподъемной" для человека, который только-только начал знакомиться с сокетами и которому не нужны все эти возможности, а требуется просто научиться передавать данные с помощью TCP/IP. Книг по сокетам очень мало, а по использованию сокетов в Delphi — вообще ни одной. Между тем сокеты в Delphi имеют свою специфику из-за отличий языка C, на который ориентирована библиотека сокетов, и Delphi: макросы заменены функциями, изменены параметры некоторых функций, определения типов приспособлены к возможностям Delphi. Кроме того, стандартный модуль Delphi для импорта функций из библиотеки сокетов импортирует библиотеку не полностью и содержит некоторые неточности. Все это делает освоение сокетов "с нуля" очень сложным делом. Вторая глава книги ориентирована на человека, который не имеет опыта сетевого программирования и не знаком с терминологией. Даются все необходимые определения и пояснения, чтобы можно было полностью понять, как работа-

ют примеры, но при этом читатель не перегружается избыточной на данном этапе информацией. Попутно излагаются особенности работы с сокетами, присущие именно Delphi. После прочтения данной главы читатель получает достаточно полное представление о протоколах стека TCP/IP и об основных режимах работы сокетов, и после этого способен дальше читать документацию самостоятельно.

Глава 3 посвящена ситуациям, в которых стандартные средства ведут себя не так, как этого ожидает программист. Иногда это объясняется ошибками в реализации этих средств, но чаще тем, что программист просто не знает некоторых особенностей данной реализации. Конечно, детально изучив документацию, можно не только дать объяснение ошибкам, но и научиться предсказывать, где они могут возникнуть, но такой метод имеет очевидные сложности. В данной главе выбран другой подход — "от ошибки". Мы сначала будем рассматривать ситуацию, в которой возникает ошибка, а потом объяснять, какие особенности реализации приводят к этому. Такой порядок изложения позволяет одновременно и рассказать о том, как соответствующие средства реализованы, и предостеречь от неверного их использования.

Глава 4 целиком посвящена одному очень популярному в форумах вопросу: как вычислить арифметическое выражение, которое становится известным только во время выполнения программы (т. е., например, у нас есть арифметическое выражение в строковой переменной, а требуется вычислить его значение). С одной стороны, в Интернете можно найти немало самодельных библиотек, решающих эту задачу. Но не все из них работают достаточно корректно, потому что их авторы зачастую реализуют доморощенные методы анализа выражений, дающие в некоторых случаях сбои. С другой стороны, существует литература, в которой довольно полно изложена формальная теория создания трансляторов. Но эта теория выходит далеко за рамки простого вычисления арифметических выражений и требует соответствующих усилий для ее освоения. Мы же, с одной стороны, ограничимся только той частью теоретических сведений, которая необходима для построения вычислителя, но с другой, — будем этой теории строго следовать, создавая действительно работоспособное решение. И хотя на выходе мы получим несколько вполне работоспособных примеров, не они будут нашей главной целью. Мы будем стремиться к тому, чтобы читатель понял сам принцип построения таких анализаторов и при необходимости смог вносить в них изменения. В дальнейшем это поможет при изучении теории синтаксического анализа по специализированным книгам.

Прим

Книга ориентирована на Delphi для Win32. О Delphi для .NET мы здесь говорить не будем. При написании книги были исследованы особенности всех версий Delphi от 3-й до 2007-й, за исключением BDS 2005, и если какая-то

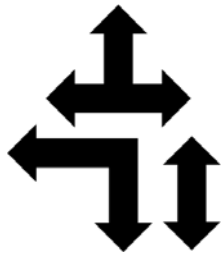
особенность или ошибка, описанная в данной книге, присутствует только в некоторых из этих версий, то это обстоятельство обязательно отмечается. Но из-за того, что с BDS 2005 автор книги не имел возможности ознакомиться, фраза "до Delphi 7 включительно" может означать "до BDS 2005" включительно, а фраза "начиная с BDS 2006" — "начиная с BDS 2005".

Автор надеется, что книга действительно окажется полезной читателю. Актуальность изложенных в ней тем подтверждается многочисленными вопросами в форумах, а полезность сведений — многочисленными ответами.

Благодарности

- Елене Филипповой, создательнице и бессменному главному администратору сайта "Королевство Delphi".
- Алексею Ковязину (российское отделение CodeGear) за помощь в выпуске этой книги.
- Юрию Зотову, натолкнувшему меня на идею четвертой главы и некоторых разделов первой и третьей глав.
- Многочисленным посетителям сайта "Королевство Delphi", при общении с которыми выяснялись интересные факты и рождались идеи, нашедшие себе место в этой книге.

ГЛАВА 1



Windows API и Delphi

Библиотека VCL, делающая создание приложений в Delphi таким быстрым и удобным, все же не позволяет разработчику задействовать все возможности операционной системы. Полный доступ к ним дает *API* (Application Programming Interface) — интерфейс, который система предоставляет программам. С его помощью можно получить доступ ко всем документированным возможностям системы.

Программированию в Windows на основе API посвящено много книг, а также материалов в Интернете. Но если все делать только с помощью API, то даже для того, чтобы создать пустое окно, потребуется написать несколько десятков строк кода, а о визуальном проектировании такого окна придется вообще забыть. Поэтому желательно как-то объединить мощность API и удобство VCL. О том, как это сделать, мы и поговорим в этой главе.

В первой части главы рассматриваются общие принципы использования API и интеграции этого интерфейса с VCL. Во второй части разбираются простые примеры, иллюстрирующие теорию. В третьей части представлено несколько обобщающих примеров использования API — небольших законченных приложений, использующих различные функции API для решения комплексных задач.

1.1. Основы работы с Windows API в VCL-приложениях

В данном разделе будет говориться о том, как совместить Windows API и компоненты VCL. Предполагается, что читатель владеет основными методами создания приложений с помощью VCL, а также синтаксисом языка Delphi, поэтому на этих вопросах мы останавливаться не будем. Так как "официаль-

ные" справка и примеры работы с API предполагают работу на C или C++, и это может вызвать трудности у человека, знакомого только с Delphi, здесь также будет уделено внимание тому, как правильно читать справку и переводить содержащийся в ней код с C/C++ на Delphi.

1.1.1. Что такое Windows API

Windows API — это набор функций, предоставляемых операционной системой каждой программе. Данные функции находятся в стандартных *динамически компонуемых библиотеках* (Dynamic Linked Library, DLL), таких как kernel32.dll, user32.dll, gdi32.dll. Указанные файлы располагаются в системной директории Window. Вообще говоря, каждая программа должна самостоятельно заботиться о том, чтобы подключить эти библиотеки. DLL могут подключаться к программе *статически* и *динамически*. В первом случае связь с библиотекой прописывается в исполняемом файле программы, и система при запуске этой программы сразу же загружает в ее адресное пространство и библиотеку. Если требуемая библиотека на диске не найдена, запуск программы будет невозможен. В случае динамического подключения программа загружает библиотеку в любой удобный для нее момент времени с помощью функции `LoadLibrary`. Если при этом возникает ошибка из-за того, что библиотека не найдена на диске, программа может самостоятельно решить, как на это реагировать.

Статическая загрузка проще динамической, но динамическая гибче. При динамической загрузке программист может, во-первых, выгрузить библиотеку, не дожидаясь окончания работы программы. Во-вторых, программа может продолжить работу, даже если библиотека не найдена. В-третьих, возможна загрузка тех DLL, имена которых неизвестны на момент компиляции. Это позволяет расширять функциональность приложения после передачи его пользователю с помощью дополнительных библиотек (в англоязычной литературе такие библиотеки обычно называются plugin).

Стандартные библиотеки необходимы самой системе и всем программам, они всегда находятся в памяти, и поэтому обычно они загружаются статически. Чтобы статически подключить в Delphi некоторую функцию Windows API, например, функцию `GetWindowDC` из модуля user32.dll, следует написать конструкцию вида

```
function GetWindowDC(Wnd: HWnd); HDC; stdcall;
  external 'user32.dll' name 'GetWindowDC';
```

В результате в специальном разделе исполняемого файла, который называется таблицей импорта, появится запись, что программа импортирует функцию `GetWindowDC` из библиотеки user32.dll. После такого объявления компилятор будет знать, как вызывать эту функцию, хотя ее реальный адрес будет внесен

в таблицу импорта только при запуске программы. Обратите внимание, что функция GetWindowDC, как и все функции Windows API, написана в соответствии с *моделью вызова* stdcall, а в Delphi по умолчанию принята другая модель — register (модель вызова определяет, как функции передаются параметры). Поэтому при импорте функций из стандартных библиотек необходимо явно указывать эту модель (подчеркнем, что это относится именно к стандартным библиотекам; другие библиотеки могут использовать любую другую модель вызова, разработчик библиотеки свободен в своем выборе). Далее указывается, из какой библиотеки импортируется функция и какое название в ней она имеет. Дело в том, что имя функции в библиотеке может не совпадать с тем, под которым она становится известной компилятору. Это может помочь разрешить конфликт имен при импорте одноименных функций из разных библиотек, а также встречается в других ситуациях, которые мы рассмотрим позже. Главным недостатком DLL следует считать то, что в них сохраняется информация только об именах функций, но не об их параметрах. Поэтому если при импорте функции указать не те параметры, которые подразумевались автором DLL, то программа будет работать неправильно (вплоть до зависания), а ни компилятор, ни операционная система не смогут указать на ошибку.

Обычно программе требуется много различных функций Windows API. Декларировать их все довольно утомительно. К счастью, Delphi избавляет программиста от этой работы: многие из этих функций уже описаны в соответствующих модулях, достаточно упомянуть их имена в разделе uses. Например, большинство общеупотребительных функций описаны в модулях Windows и Messages.

Функции API, которые присутствуют не во всех версиях Windows, предпочтительнее загружать динамически. Например, если программа статически импортирует функцию SetLayeredWindowsAttributes, она не запустится в Windows 9*x*, где этой функции нет — система, встретив ее упоминание в таблице импорта, прервет загрузку программы. Поэтому, если требуется, чтобы программа работала и в Windows 9*x*, эту функцию следует импортировать динамически. Отметим, что компоновщик в Delphi помещает в таблицу импорта только те функции, которые реально вызываются программой. Поэтому наличие декларации SetLayeredWindowsAttributes в модуле Windows не помешает программе запускаться в Windows 9*x*, если она не вызывает эту функцию.

1.1.2. Как получить справку по функциям Windows API

Для тех, кто решил работать с Windows API, самым необходимым инструментом становится какая-либо документация по этим функциям. Их так мно-

го, что запомнить все совершенно нереально, поэтому работа без справочника под рукой просто невозможна.

Первоисточник информации по технологиям Microsoft для разработчика — *Microsoft Developer's Network* (MSDN). Это отдельная справочная система, не входящая в комплект поставки Delphi. MSDN можно приобрести отдельно или воспользоваться online-версией, находящейся по адресу: **http://msdn.microsoft.com** (доступ к информации свободный, регистрация не требуется). MSDN содержит не только информацию об API, но и все, что может потребоваться программисту, использующему различные средства разработки от Microsoft. Кроме справочного материала, MSDN включает в себя спецификации стандартов и технологий, связанных с Windows, статьи из журналов, посвященных программированию, главы из некоторых книг. И вся эта информация крайне полезна разработчику. Кроме того, MSDN постоянно обновляется, информация в нем наиболее актуальна. Пример справки из MSDN показан на рис. 1.1.

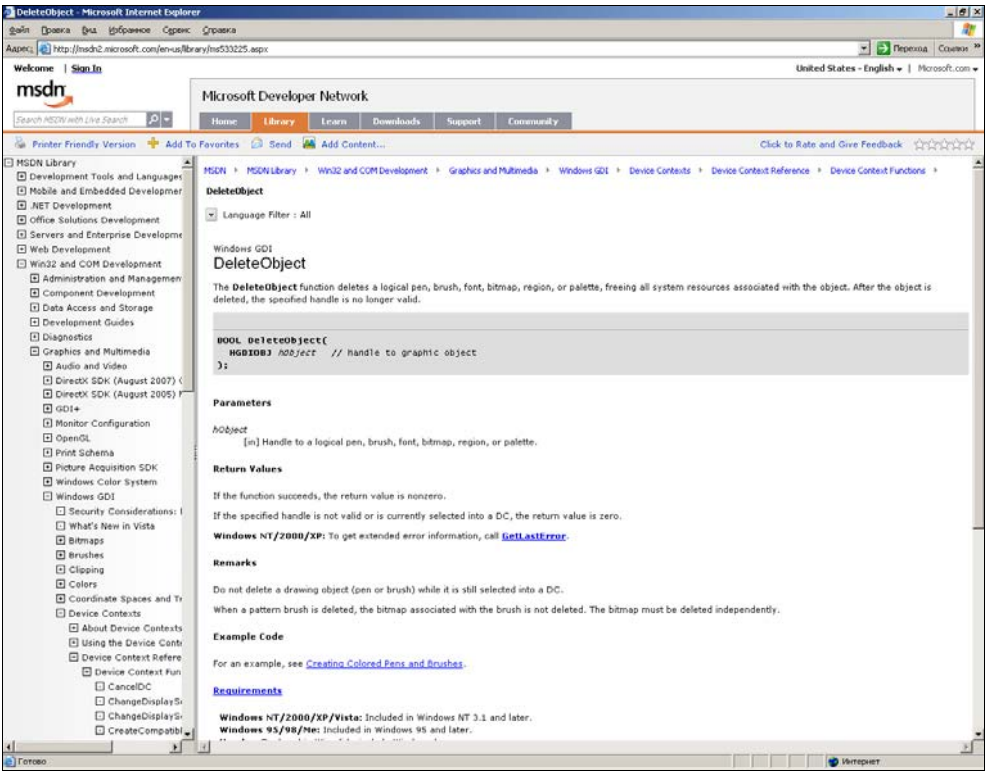


Рис. 1.1. Online-вариант MSDN (показана справка по функции `DeleteObject`)

Примечание

Отметим, что MSDN содержит также описание функций операционной системы Windows CE. Интерфейс Windows CE API на первый взгляд очень похож на Windows API, но различия между ними есть, и иногда весьма значительные. Поэтому при использовании MSDN не следует выбирать раздел API Reference — он целиком посвящен WinCE API.

В комплект поставки Delphi входит справочная система, содержащая описание функций Windows API. Справочная система в Delphi до 7-й версии включительно была построена на основе hlp-файлов. Применительно к справке по Windows API это порождало две проблемы. Во-первых, hlp-файлы имеют ограничение по числу разделов в справочной системе, поэтому объединить в одной справке информацию и по Delphi, и по Windows API было невозможно, — эти две справки приходилось читать по очереди. Чтобы открыть файл справки по Windows API, нужно было в редакторе кода поставить курсор на название какой-либо функции API и нажать клавишу <F1> — в этом случае вместо справки по Delphi открывалась справка по Windows API. Второй вариант — в меню **Программы** найти папку **Delphi**, а в ней — папку **Help\MS SDK Files** и выбрать требуемый раздел. Можно также вручную открыть файл MSTools.hlp. В ранних версиях Delphi он находится в каталоге $(Delphi)\Help, в более поздних его нужно искать в $(Program Files)\Common Files. Окно старой справки показано на рис. 1.2.

Вторая проблема, связанная со справкой на основе hlp-файлов, — это то обстоятельство, что разработчики Delphi, разумеется, не сами писали эту справку, а взяли ту, которую предоставила Microsoft. Microsoft же последнюю версию справки в формате HLP выпустила в тот момент, когда уже вышла Windows 95, но еще не было Windows NT 4. Поэтому про многие функции, прекрасно работающие в NT 4, там написано, что в Windows NT они не поддерживаются, т. к. в более ранних версиях они действительно не поддерживались. В справке, поставляемой с Delphi 7 (и, возможно, с некоторыми более ранними версиями), эта информация подправлена, но даже и там отсутствуют функции, которые появились только в Windows NT 4 (как, например, `CoCreateInstanceEx`). И уж конечно, бесполезно искать в этой справке информацию о функциях, появившихся в Windows 98, 2000, XP. Соответственно, при работе в этих версиях Delphi даже не возникает вопрос, что предпочесть для получения информации о Windows API, — справку, поставляемую с Delphi, или MSND. Безусловно, следует выбрать MSDN. Справка, поставляемая с Delphi, имеет только одно преимущество по сравнению с MSDN: ее можно вызывать из среды нажатием клавиши <F1>. Но риск получить неверные сведения слишком велик, чтобы это преимущество могло быть серьезным аргументом. Единственная ситуация, когда предпочтительна справка, поставляемая с Delphi, — это случай, если у вас нет достаточно быстрого

доступа к Интернету для работы с online-версией MSDN и нет возможности приобрести и установить его offline-версию.

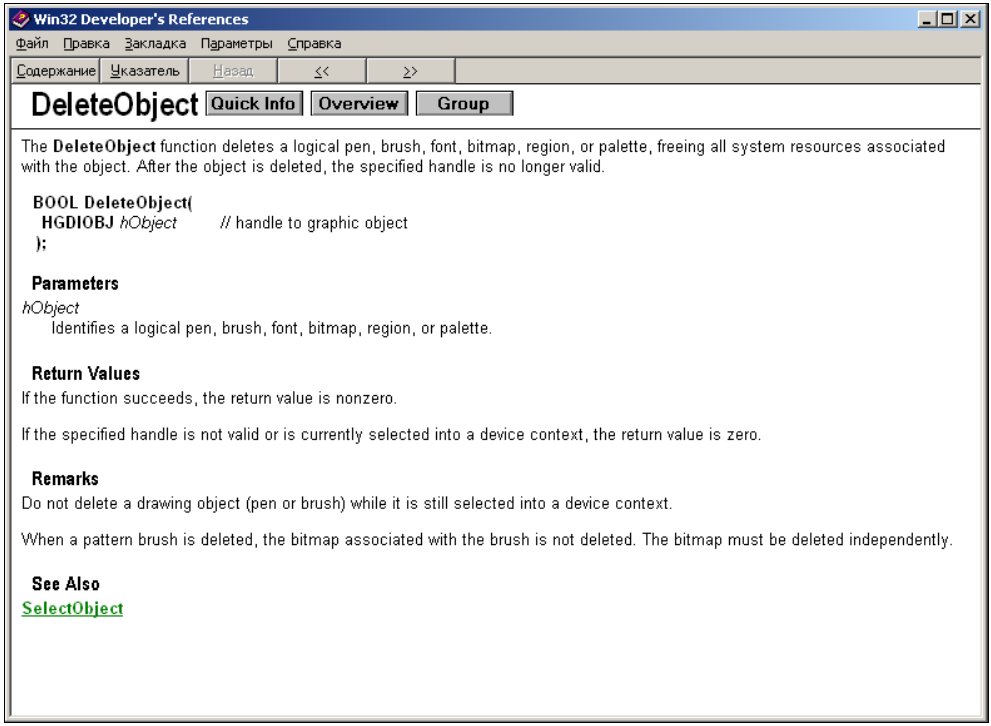


Рис. 1.2. Старая (на основе hlp-файлов) справка по Windows API (показана функция `DeleteObject`)

Начиная с BDS 2006, Borland/CodeGear реализовала новую справочную систему Borland Help (рис. 1.3). По интерфейсу она очень напоминает offline-версию MSDN, а также использует файлы в том же формате, поэтому технологических проблем интеграции справочных систем по Delphi и по Windows API больше не существует. В справку BDS 2006 интегрирована справка по Windows API от 2002—2003 годов (разные разделы имеют разную дату). Справка Delphi 2007 содержит сведения по Windows API от 2006 года, т. е. совсем новые. Таким образом, при работе с Delphi 2007 наконец-то можно полностью отказаться от offline-версии MSDN, а к online-версии обращаться лишь изредка, когда требуется информация о самых последних изменениях в Windows API (например, о тех, которые появились в Windows Vista).

Примечание

Несмотря на очень высокое качество разделов MSDN, относящихся к Windows API, ошибки иногда бывают и там. Со временем их исправляют. Поэтому, если

вы столкнулись с ситуацией, когда есть подозрение, что какая-либо функция Windows API ведет себя не так, как это описано в вашей offline-справке, есть смысл заглянуть в online-справку — возможно, там уже появились дополнительные сведения по данной функции.

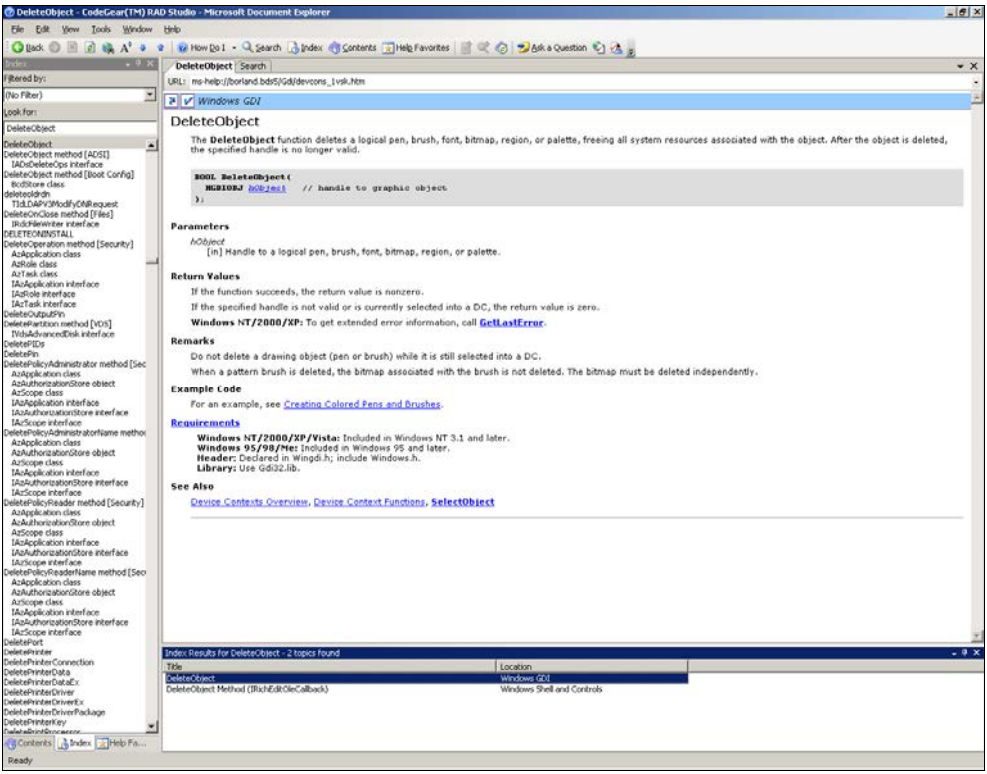


Рис. 1.3. Окно справки Delphi 2007 (функция `DeleteObject`)

Система Windows написана на C++, поэтому все описания функций Windows API, а также примеры их использования приведены на этом языке (это касается как MSDN, так и справки, поставляемой с Delphi). При этом, прежде всего, необходимо разобраться с типами данных. Большинство типов, имеющихся в Windows API, определены в Delphi. Соответствие между ними показано в табл. 1.1.

Таблица 1.1. *Соответствие типов Delphi системным типам*

Тип Windows API	Тип Delphi
`INT`	`INT`
`UINT`	`LongWord`

Таблица 1.1 *(окончание)*

Тип Windows API	Тип Delphi
WORD	Word
SHORT	SmallInt
USHORT	Word
CHAR	Чаще всего соответствует типу Char, но может трактоваться также как ShortInt, т. к. в C++ нет разницы между символьным и целочисленным типами
UCHAR	Чаще всего соответствует типу Byte, но может трактоваться также как Char
DWORD	LongWord
BYTE	Byte
WCHAR	WideChar
BOOL	LongBool
int	Integer
long	LongInt
short	SmallInt
unsigned int	Cardinal

Название типов указателей имеет префикс *P* или *LP* (Pointer или Long Pointer; в 16-разрядных версиях Windows были короткие и длинные указатели. В 32-разрядных все указатели длинные, поэтому оба префикса имеют одинаковый смысл). Например, LPDWORD эквивалентен типу ^DWORD, PUCHAR — ^Byte. Иногда после префикса *P* или *LP* стоит еще префикс *C* — он означает, что это указатель на константу. В C++ возможно объявление таких указателей, которые указывают на константное содержимое, т. е. компилятор разрешает это содержимое читать, но не модифицировать. В Delphi такие указатели отсутствуют, и при портировании эти типы заменяются обычными указателями, т. е. префикс *C* игнорируется.

Типы PVOID и LPVOID соответствуют нетипизированным указателям (Pointer).

Для передачи символов чаще всего используется тип TCHAR. Windows поддерживает две кодировки: ANSI (1 байт на символ) и Unicode (2 байта на символ; о поддержке Unicode в Windows мы будем говорить далее). Тип CHAR соответствует символу в кодировке ANSI, WCHAR — Unicode. Для программ, которые используют ANSI, тип TCHAR эквивалентен типу CHAR, для использующих

Unicode — `WCHAR`. В Delphi нет прямого аналога типу `TCHAR`, программист сам должен следить за тем, какой символьный тип требуется в данном месте.

Строки в Windows API передаются как указатели на цепочку символов, завершающихся нулем. Поэтому указатель на `TCHAR` может указывать как на единичный символ, так и на строку. Чтобы было легче разобраться, где какой указатель, в Windows API есть типы `LPTCHAR` и `LPTSTR`. Они эквивалентны друг другу, но первый принято использовать там, где требуется указатель на одиночный символ, а второй — на строку. Если строка передается в функцию только для чтения, обычно используют указатель на константу, т. е. тип `LPCTSTR`. В Delphi это соответствует `PChar` для ANSI и `PWideChar` для Unicode.

Здесь следует отметить особенность записи строковых литералов в языках C/C++. Символ `\` в литерале имеет специальное значение: после него идет один или несколько управляющих символов. Например, `\n` означает перевод строки, `\t` — символ табуляции и т. п. В Delphi таких последовательностей нет, поэтому при переводе примеров из MSDN следует явно писать коды соответствующих символов. Например, литерал `"a\nb"` в Delphi превращается в `'a'#13'b'`. После символа `\` может идти число — в этом случае оно трактуется как код символа, т. е. литерал `"a\0b\9"` в C/C++ эквивалентен литералу `'a'#0'b'#9` в Delphi. Если нужно, чтобы строковый литерал включал в себя сам символ `\`, его удваивают, т. е. литерал `"\\"` в C++ соответствует `'\'` в Delphi. Кроме того, в примерах кода, приведенных в MSDN, можно нередко увидеть, что строковые литералы обрабатываются макросами `TEXT` или `_T`, которые служат для унификации записи строковых литералов в кодировках ANSI и Unicode. При переводе такого кода на Delphi эти макросы можно просто опустить. С учетом сказанного такой, например, код (взят из примера использования Named pipes):

```
LPTSTR lpszPipename = TEXT("\\\\.\\pipe\\mynamedpipe");
```

на Delphi будет выглядеть так:

```
var
  lpszPipeName: PChar;
...
lpszPipeName := '\\.\pipe\mynamedpipe';
```

Большинство названий типов из левой части табл. 1.1 в целях совместимости описаны в модуле `Windows`, поэтому они допустимы наравне с обычными типами Delphi. Кроме этих типов общего назначения существуют еще специальные. Например, дескриптор окна имеет тип `HWND`, первый параметр сообщения — тип `WPARAM` (в старых 16-разрядных Windows он был эквивалентен типу `Word`, в 32-разрядных — `LongInt`). Эти специальные типы также описаны в модуле `Windows`.

Записи (record) в C/C++ называются структурами и объявляются с помощью слова struct. Из-за особенностей описания структур на языке C структуры в Windows API получают два имени: одно основное имя, составленное из заглавных букв, которое затем и используется, и одно вспомогательное, получающееся из основного добавлением префикса tag. Начиная с четвертой версии Delphi приняты следующие правила именования таких типов: простое и вспомогательное имена остаются без изменений и еще добавляется новое имя, получающееся из основного присоединением общеупотребительного в Delphi префикса *T*. Например, в функции CreatePenIndirect один из параметров имеет тип LOGPEN. Это основное имя данного типа, а вспомогательное — tagLOGPEN. Соответственно, в модуле Windows определена запись tagLOGPEN и ее синонимы — LOGPEN и TLogPen. Эти три идентификатора в Delphi взаимозаменяемы. Вспомогательное имя встречается редко, программисты, в зависимости от личных предпочтений, выбирают либо основное имя типа, либо имя с префиксом *T*.

Описанные здесь правила именования типов могут внести некоторую путаницу при использовании VCL. Например, для описания растра в Windows API определен тип BITMAP (он же — tagBITMAP). В Delphi соответствующий тип имеет еще одно имя — TBitmap. Но такое же имя имеет класс TBitmap, описанный в модуле Graphics. В коде, который Delphi создает автоматически, модуль Graphics находится в списке uses после модуля Windows, поэтому идентификатор TBitmap воспринимается компилятором как Graphics.TBitmap, а не как Windows.TBitmap. Чтобы использовать Windows.TBitmap, нужно явно указать имя модуля или воспользоваться одним из альтернативных имен.

В более ранних версиях Delphi были другие правила именования типов. Например, в Delphi 2 существовал тип BITMAP, но не было TBitmap и tagBITMAP, а в Delphi 3 из этих трех типов был только TBitmap.

Все структуры в Windows API описаны без *выравнивания*, т. е. компилятор не вставляет между полями неиспользуемые байты, чтобы границы полей приходились на начало двойного или четверного слова, поэтому в Delphi для описания соответствующих структур предусмотрено слово packed, запрещающее выравнивание.

При описании структур Windows API можно иногда встретить ключевое слово union (см., например, структуру in_addr). Объединение нескольких полей с помощью этого слова означает, что все они будут размещены по одному адресу. В Delphi это соответствует вариантным записям (т. е. использованию case в record). Объединения в C/C++ гибче, чем вариантные записи Delphi, т. к. позволяют размещать вариантную часть в любом месте структуры, а не только в конце. При переносе таких структур в Delphi иногда приходится вводить дополнительные типы.

Теперь рассмотрим синтаксис описания самой функции в C++ (листинг 1.1).

Листинг 1.1. Синтаксис описания функции на C++

```
<Тип функции> <Имя функции> '('
    [ <Тип параметра> {<Имя параметра>}
      {',' <Тип параметра> {<Имя параметра>} }
    ]
 ')';
```

Как видно из листинга 1.1, при объявлении функции существует возможность указывать только типы параметров и не указывать их имена. Однако это считается устаревшим и применяется крайне редко (если не считать "параметров" типа VOID, о которых написано далее).

Необходимо помнить, что в C/C++ различаются верхний и нижний регистры, поэтому HDC, hdc, hDC и т. д. — это разные идентификаторы (автор C очень любил краткость и хотел, чтобы можно было делать не 26, а 52 переменные с именем из одной буквы). Поэтому часто можно встретить, что имя параметра и его тип совпадают с точностью до регистра. К счастью, при описании функции в Delphi мы не обязаны сохранять имена параметров, значение имеют лишь их типы и порядок следования. С учетом всего этого функция, описанная в справке как

```
HMETAFILE CopyMetaFile(HMETAFILE hmfSrc, LPCTSTR lpszFile);
```

в Delphi имеет вид

```
function CopyMetaFile(hmfSrc: HMETAFILE;
  lpszFile: LPCTSTR): HMETAFILE;
```

или, что то же самое,

```
function CopyMetaFile(hmfSrc: HMETAFILE; lpszFile: PChar): METAFILE;
```

Примечание

Компилятор Delphi допускает, чтобы имя параметра процедуры или функции совпадало с именем типа, поэтому мы в дальнейшем увидим, что иногда имя параметра и его тип совпадают, только записываются в разном регистре, чтобы прототип функции на Delphi максимально соответствовал исходному прототипу на C/C++. При этом следует учитывать, что соответствующий идентификатор внутри функции будет рассматриваться как имя переменной, а не типа, поэтому, например, объявлять локальную переменную данного типа придется с явным указанием имени модуля, в котором данный тип объявлен.

Несколько особняком стоит тип VOID (или void, что то же самое, но в Windows API этот идентификатор встречается существенно реже). Если

функция имеет такой тип, то в Паскале она описывается как процедура. Если вместо параметров у функции в скобках указан VOID, это означает, что функция не имеет параметров. Например, функция

```
VOID CloseLogFile(VOID);
```

в Delphi описывается как

```
procedure CloseLogFile;
```

Примечание

Язык C++, в отличие от C, допускает объявление функций без параметров, т. е. функцию CloseLogFile можно было бы объявить так: VOID CloseLogFile();. В C++ эти варианты объявления эквивалентны, но в Windows API вариант без явного параметра встречается существенно реже из-за несовместимости с C.

Когда тип параметра является указателем на другой тип (обычно начинается с букв LP), при описании этой функции в Delphi можно пользоваться параметром-переменной, т. к. в этом случае функции передается указатель. Например, функция

```
int GetRgnBox(HRGN hrgn, LPRECT lprc);
```

в модуле Windows описана как

```
function GetRgnBox(RGN: HRGN; var p2: TRect): Integer;
```

Такая замена целесообразна в том случае, если значение параметра не может быть нулевым указателем, потому что при использовании var передать такой указатель будет невозможно. Нулевой указатель в C/C++ обозначается константой NULL. NULL и 0 в этих языках взаимозаменяемы, поэтому в справке можно и про целочисленный параметр встретить указание, что он может быть равен NULL.

И наконец, если не удается понять, как функция, описанная в справке, должна быть переведена на Паскаль, можно попытаться найти описание этой функции в исходных текстах модулей, поставляемых вместе с Delphi. Эти модули находятся в каталоге $(DELPHI)\Source\RTL\Win (до Delphi 7) или $(BDS)\Source\Win32\RTL\Win (BDS 2006 и выше). Можно также воспользоваться подсказкой, которая всплывает в редакторе Delphi после того, как будет набрано имя функции.

Если посмотреть справку, например, по функции GetSystemMetrics, то видно, что эта функция должна иметь один целочисленный параметр. Однако далее в справке предлагается при вызове этой функции подставлять в качестве параметра не числа, а SM_ARRANGE, SM_CLEANBOOT и т. д. Подобная ситуация и со многими другими функциями Windows API. Все эти SM_ARRANGE, SM_CLEANBOOT и т. д. являются именами числовых констант. Эти константы описаны в том

же модуле, в котором описана функция, использующая их, поэтому можно не выяснять численные значения этих констант, а указывать при вызове функций их имена, например, `GetSystemMetrics(SM_ARRANGE);`.

Если по каким-то причинам все-таки потребовалось выяснить численные значения, то в справочной системе их искать не стоит — их там нет. Их можно узнать из исходных текстов модулей Delphi, в которых эти константы описаны. Так, например, просматривая Windows.pas, можно узнать, что `SM_ARRANGE = 56`.

В справке, поставляемой вместе с Delphi до 7-й версии включительно, в описании многих функций Windows API вверху можно увидеть три ссылки: **QuickInfo**, **Overview** и **Group**. Первая дает краткую информацию о функции: какой библиотекой реализуется, в каких версиях Windows работает и т. п. (напоминаю, что к информации о версиях в этой справке нужно относиться очень критично). **Overview** — это обзор какой-то большой темы. Например, для любой функции, работающей с растровыми изображениями, обзор будет объяснять, зачем в принципе нужны эти самые растровые изображения и как они устроены. Страница, на которую ведет ссылка **Overview**, обычно содержит весьма лаконичные сведения, но, нажав кнопку >>, расположенную в верхней части окна, можно получить продолжение обзора. И, наконец, **Group**. Эта ссылка приводит к списку всех функций, родственных данной. Например, для функции `CreateRectRgn` группу будут составлять все функции, имеющие отношение к регионам. Если теперь нажимать на кнопку <<, то будут появляться страницы с кратким описанием возможных применений объектов, с которыми работают функции (в приведенном примере — описание возможностей регионов). Чтобы читать их в нормальной последовательности, лучше всего нажать на кнопку << столько раз, сколько возможно, а затем пойти в противоположном направлении с помощью кнопки >>.

MSDN (а также справка BDS 2006 и выше) предоставляет еще больше полезной информации. В нижней части описания каждой функции есть раздел **Requirements**, в котором написано, какая библиотека и какая версия Windows требуется для ее использования. В самом низу описания функции расположены ссылки **See also**. Первая ссылка — обзор соответствующей темы (например, для уже упоминавшейся функции `CreateRectRgn` она называется **Regions Overview**). Вторая — список родственных функций (**Region Functions** в данном случае). Она ведет на страницу, где перечислены все функции, родственные выбранной. После этих двух обязательных ссылок идут ссылки на описание функций и типов, которые обычно используются совместно с данной функцией.

Основные типы, константы и функции Windows API объявлены в модулях `Windows` и `Messages`. Но многие функции объявлены в других модулях, кото-

рые не подключаются к программе по умолчанию, программист должен сам выяснить, в каком модуле находится требуемый ему идентификатор, и подключить этот модуль. Ни справка, поставляемая с Delphi, ни MSDN, разумеется, не могут дать необходимую информацию. Чтобы выяснить, в каком модуле объявлен нужный идентификатор, можно воспользоваться поиском по всем файлам с расширением pas, находящимся в папке с исходными кодами стандартных модулей. Этим методом можно, например, выяснить, что весьма популярная функция `ShellExecute` находится в модуле `ShellAPI`, а `CoCreateInstance` — в модуле `ActiveX` (а также в модуле `Ole2`, оставленном для совместимости со старыми версиями Delphi).

Еще несколько слов о числовых константах. В справке можно встретить числа вида, например, `0xC56F` или `0x3341`. Префикс `0x` в C/C++ означает шестнадцатеричное число. В Delphi его следует заменить на `$`, т. е. эти числа должны быть записаны как `$C56F` и `$3341` соответственно.

1.1.3. Дескрипторы вместо классов

Программируя в Delphi, мы быстро привыкаем к тому, что каждый объект реализуется экземпляром соответствующего класса. Например, кнопка реализуется экземпляром класса `TButton`, контекст устройства — классом `TCanvas`. Но когда создавались первые версии Windows, объектно-ориентированный метод программирования еще не был общепризнанным, поэтому он не был реализован. Современные версии Windows частично унаследовали этот недостаток, поэтому в большинстве случаев приходится работать "по старинке", тем более что DLL могут экспортировать только функции, но не классы. Когда мы будем говорить об объектах, создаваемых через Windows API, будем подразумевать не объекты в терминах ООП, а некоторую сущность, внутренняя структура которой скрыта от нас, поэтому с этой сущностью мы можем оперировать только как с единым и неделимым (атомарным) объектом.

Каждому объекту, созданному с помощью Windows API, присваивается уникальный номер *(дескриптор)*. Его конкретное значение не несет для программиста никакой полезной информации и может быть использовано только для того, чтобы при вызове функций из Windows API указывать, с каким объектом требуется выполнить операцию. В большинстве случаев дескрипторы представляют собой 32-значные числа, а значит, их можно передавать везде, где требуются такие числа. В дальнейшем мы увидим, что Windows API несколько вольно обращается с типами, т. е. один и тот же параметр в различных ситуациях может содержать и число, и указатель, и дескриптор, поэтому знание двоичного представления дескриптора все-таки приносит программисту пользу (хотя если бы система Windows была "спроектирована по правилам", тип дескриптора вообще не должен был интересовать программиста).

Таким образом, главное различие между методами класса и функциями Windows API заключается в том, что первые связаны с тем экземпляром класса, через который они вызываются, и поэтому не требуют явного указания на объект. Вторым необходимо указание объекта через его дескриптор, т. к. они сами по себе никак не связаны ни с одним объектом.

Компоненты VCL нередко являются оболочками над объектами Delphi. В этом случае они имеют свойство (которое обычно называется `Handle`), содержащее дескриптор соответствующего объекта. Иногда класс Delphi инкапсулирует несколько объектов Windows. Например, класс `TBitmap` включает в себя `HBITMAP` и `HPALETTE` — картинку и палитру к ней. Соответственно, он хранит два дескриптора: в свойствах `Handle` и `Palette`.

Следует учитывать, что внутренние механизмы VCL не могут включиться, если изменение объекта происходит через Windows API. Например, если спрятать окно не с помощью метода `Hide`, а путем вызова функции Windows API `ShowWindow(Handle, SW_HIDE)`, не возникнет событие `OnHide`, потому что оно запускается теми самыми внутренними механизмами VCL. Но такие недоразумения случаются обычно только тогда, когда функциями Windows API дублируется то, что можно сделать и с помощью VCL.

Все экземпляры классов, созданные в Delphi, должны удаляться. В некоторых случаях это происходит автоматически, а иногда программист должен сам позаботиться о "выносе мусора". Аналогичная ситуация и с объектами, создаваемыми в Windows API. Если посмотреть справку по функции, создающей какой-то объект, то там обязательно будет информация о том, какой функцией можно удалить объект и нужно ли это делать вручную, или система сделает это автоматически. Во многих случаях совершенно разные объекты могут удаляться одной и той же функцией. Так, функция `DeleteObject` удаляет косметические перья, геометрические перья, кисти, шрифты, регионы, растровые изображения и палитры. Обращайте внимание на возможные исключения. Например, регионы не удаляются системой автоматически, однако если вызвать для региона функцию `SetWindowRgn`, то он переходит в собственность операционной системы. Никакие дальнейшие операции с ним, в том числе и удаление, совершать нельзя.

Если системный объект используется только одним приложением, то он будет удален при завершении работы приложения. Тем не менее хороший стиль программирования требует, чтобы программа удаляла объекты явно, а не полагалась на систему.

1.1.4. Формы VCL и окна Windows

Под словом "окно" обычно подразумевается некоторая форма наподобие тех, что можно создать с помощью класса `TForm`. Однако это понятие существенно

шире. В общем случае окном называется любой объект, который имеет экранные координаты и может реагировать на мышь и клавиатуру. Например, кнопка, которую можно создать с помощью класса TButton, — это тоже окно.

VCL вносит некоторую путаницу в это понятие. Некоторые визуальные компоненты VCL не являются окнами, а только имитируют их, как, например, TImage. Это позволяет экономить ресурсы системы и повысить быстродействие программы. Механизм этой имитации мы рассмотрим позже, а пока следует запомнить, что окнами являются только те визуальные компоненты, которые имеют в числе предков класс TWinControl. Разработчики VCL постарались, чтобы разница между *оконными* и *неоконными визуальными компонентами* была минимальной. Действительно, на первый взгляд неоконный TLabel и оконный TStaticText кажутся практически близнецами. Разница становится заметной тогда, когда используется Windows API. С неоконными компонентами можно работать только средствами VCL, они даже не имеют свойства Handle, в то время как оконными компонентами можно управлять с помощью Windows API.

Отметим также еще одно различие между оконными и неоконными компонентами: неоконные компоненты рисуются непосредственно на поверхности *родительского компонента*, в то время как оконные как бы "кладутся" на родителя сверху. В частности, это означает, что неоконный TLabel, размещенный на форме, не может закрывать собой часть кнопки TButton, потому что TLabel рисуется на поверхности формы, а кнопка — это независимый объект, лежащий на форме и имеющий свою поверхность. А TStaticText может оказаться над кнопкой, потому что он тоже находится над формой.

Примечание

Чтобы разместить неоконный визуальный компонент над оконным, если в этом есть необходимость, можно поступить следующим образом. Положить на форму панель (TPanel) — она является оконным компонентом и может быть размещена поверх других оконных элементов. На панель теперь можно положить любой неоконный визуальный компонент, и он будет рисоваться не на поверхности формы, а на поверхности панели. Если теперь убрать у панели рамку и уменьшить ее до размеров содержащегося в ней неоконного компонента, панель станет незаметной, и все вместе это будет выглядеть так, будто неоконный компонент находится над оконным.

Каждое окно принадлежит к какому-то *оконному классу*. Не следует путать оконный класс с классами Delphi. Это некий шаблон, определяющий базовые свойства окна. Каждому такому шаблону присваивается имя, уникальное в его области видимости. Перед использованием класс необходимо зарегистрировать (функция RegisterClassEx). В качестве параметра эта функция принимает запись типа TWndClassEx, поля которой содержат параметры класса.

С каждым окном должна быть связана специальная функция, называющаяся *оконной процедурой* (подробнее мы рассмотрим ее чуть позже). Она является параметром не отдельного окна, а всего оконного класса, т. е. все окна, принадлежащие данному классу, будут использовать одну и ту же оконную процедуру. Эта процедура может размещаться либо в самом исполняемом модуле, либо в одной из загруженных им DLL. При создании класса указывается дескриптор модуля, в котором находится оконная процедура.

Примечание

Здесь следует отметить некоторую путаницу в терминах. В англоязычной справке есть слово module, служащее для обозначения файла, отображенного в адресное пространство процесса, т. е., в первую очередь, exe-файла, породившего процесс, и загруженных им DLL. И есть слово unit, которое обозначает модуль в Delphi и которое также переводится как модуль. Ранее мы говорили о модулях как об отображаемых в адресное пространство файлах — это они имеют дескрипторы. Модули Delphi не являются системными объектами и дескрипторов не имеют.

Дескриптор модуля, загруженного в память, можно получить с помощью функции `GetModuleHandle`. Функция `LoadLibrary` в случае успешного завершения также возвращает дескриптор загруженной DLL. Кроме того, Delphi предоставляет две переменные: `MainInstance` из модуля `System` и `HInstance` из модуля `SysInit` (оба этих модуля подключаются к программе автоматически, без явного указания в списке `uses`). `MainInstance` содержит дескриптор exe-файла, породившего процесс, `HInstance` — текущего модуля. В исполняемом файле `MainInstance` и `HInstance` равны между собой, в DLL `HInstance` содержит дескриптор самой библиотеки, а `MainInstance` — загрузившего ее главного модуля.

Каждое окно в Windows привязывается к какому-либо модулю (в Windows 9*x*/ME необходимо явно указать дескриптор этого модуля, NT/2000/XP определяет модуль, из которого вызвана функция создания окна, автоматически). Соответственно, оконные классы делятся на *локальные* и *глобальные*: окна локальных классов может создавать только тот модуль, в котором находится оконная процедура класса, глобальных — любой модуль данного приложения. Будет ли класс локальным или глобальным, зависит от значений полей `TWndClassEx` при регистрации класса.

Оконный класс, к которому принадлежит окно, указывается при его создании. Это может быть зарегистрированный ранее класс или один из *системных классов*. Системные классы — это `'BUTTON'`, `'COMBOBOX'`, `'EDIT'`, `'LISTBOX'`, `'MDICLIENT'`, `'SCROLLBAR'` и `'STATIC'`. Назначение этих классов понятно из их названий (класс `'STATIC'` реализует статические текстовые или графические элементы, т. е. не реагирующие на мышь и клавиатуру, но имеющие дескриптор). Кроме этих классов существуют также классы из биб-