С.Гилберт, Б.Маккарти

САМОУЧИТЕЛЬ VISUAL C++ 6 В ПРИМЕРАХ

Книга *Самоучитель Visual C++ 6 в примерах,* написанная авторами множества бестселлеров Стивеном Гилбертом и Биллом Маккарти, представляет собой введение в искусство программирования в среде Visual C++ 6 с использованием как Windows API, так и библиотеки Microsoft Foundation Classes (MFC).

Простой и доступный стиль изложения, изобилие примеров, доступных в виде исходных кодов на сопровождающем CD-ROM, позволяют за короткое время научиться основам создания Windows-приложений и приступить к самостоятельной работе. Книга является неформальным руководством по технологии программирования с использованием MFC. Подробно рассматриваются вопросы, связанные с построением форм, диалоговых окон, приложений для работы с базами данных, приложений с архитектурой "документ-представление" (DocView), программ рисования и Web-браузеров. Немалое внимание уделяется проблемам повторного использования кода, а также работе с компонентами и элементами управления ActiveX.

Внимательно ознакомившись с книгой, любой читатель сможет немедленно приступить к самостоятельному созданию эффективных бизнес-приложений.

Для широкого круга разработчиков программного обеспечения на платформе Microsoft.

Оглавление

Предметный указатель

Иностранные термины

Несколько тысяч лет назад царь Соломон изрек такую мудрость: "Завершение дела предпочтительнее его начала, а терпение ценнее гордыни". По окончании этого дела я хотел бы поблагодарить свою жену Кэтлин, чье терпение выше всяких похвал, а также своего соавтора Билла Мак-Карти (Bill McCarty), без участия которого работа была бы еще далека до завершения. В конечном итоге, однако, вся благодарность моя вершителю Начала и Конца, Альфы и Омеги, Создателю и Спасителю моей души — Иисусу Христу.

— Стивен Гилберт (Stephen Gilbert)

Стивен Гилберт, мой соавтор, заслуживает самой большой благодарности. Все хорошие части книги — его, я же делал остальное. Как всегда, мое семейство (Дженнифер (Jennnifer), Патрик (Patric) и Сара (Sara)) поддерживало и хранило меня в трудные периоды. Я клянусь поступать точно также, когда каждый из них будет занят написанием своих первых четырех книг. После этого они обретут самостоятельность. Глубокая благодарность Иисусу Христу, хранителю моей души, который призвал меня к себе, и я буду с Ним вечно. Моя доля заработка от этой книги в основном будет потрачена на приобретение домашней студии для записи современной богоугодной музыки, посвященной восхвалению Создателя.

— Билл Маккарти (Bill McCarty)

Благодарности

В процессе работы над книгой с нами сотрудничали многие люди. Самый опытный выпускающий редактор издательства Coriolis, Стефани Уолл (Stephanie Wall), оказала нам неоценимую услугу. Прими нашу благодарность, Стефани. К работе были привлечены высококлассные редакторы. Мы любим их не только за то, что они взяли на себя тяжкий труд, участвуя в этом проекте, но и за то, что они веселые товарищи, с которыми приятно работать. Если мы когда-либо примемся за другую книгу, то нам хотелось бы сохранить нашу команду. Наши благодарности редактору проекта Мишелю Строупу (Michelle Stroup) редактору издания Тиффани Тэйлор (Tiffany Taylor) и техническому редактору Джону Коксу (John Cox). Благодарим также всех остальных участников группы, а именно — производственного координатора Венди Литтли (Wendy Littley) и разработчика CD-ROM Роберта Кларфилда (Robert Clarfield).

От редактора

При подготовке русскоязычной редакции этой книги была предпринята попытка тщательно сохранить оригинальный авторский стиль. Поскольку книга задумывалась авторами как неформальное руководство, побуждающее к дальнейшим действиям, нельзя не заметить некоторые вольности в применении терминологии, принятой в технологии объектно-ориентированного анализа, проектирования и реализации.

Вышеозначенное особенно касается таких сущностей, как *member variable* и *member function*. В книге *member variable* переводится и как *элемент данных*, и как

переменная (разумеется, какого-то класса). Существуют и другие трактовки упомянутого термина, например, *переменная-член*. То же самое можно сказать и о *member function*. Здесь было применено две версии перевода: *метод* и *функция* (естественно, класса). С другой стороны, зачастую используют термин *функция-член*. Очень трудно определить, какой из вариантов претендует на абсолют. Мне кажется, что **все**. Главное — это четкое понимание **что** собой представляет сущность, а уж как ее назвать — дело второстепенное. Тем более, что хороший тон в объектно-ориентированном программировании не допускает никаких переменных, кроме переменных, принадлежащих классу, и никаких функций, за исключением тех, которые являются частью класса.

Честно говоря, лично я предпочитаю трактовки известных гуру, основоположников формального представления объектно-ориентированной методологии, господ Г.Буча и Дж.Румбо: данные (или элементы данных) и методы вместе образуют класс (разумеется, утверждение приведено в упрощенной форме).

Еще одно замечание. Несмотря на то что комментарии в сгенерированных кодах (например, при помощи AppWizard), кажется, никогда не будут русскоязычными (хотя, кто знает?), в книге они, все же, переведены. Просто хотелось немного помочь разобраться в методике генерации кода, предложенной компанией Microsoft. Естественно, перевод неформальный, и никоим образом не претендует на использование в качестве стандарта в будущих локализованных версиях Visual C++!

Успехов в постижении Visual C++!

Введение

Мы написали эту книгу, чтобы помочь вам овладеть Visual C++, разработанным компанией Microsoft, и методикой построение приложений, основанных на Microsoft Foundation Classes (MFC). Мы отнюдь не обещаем, что сразу же после прочтения этой книги вы станете "неуязвимым орлом" среди пользователей Visual C++/MFC. Но однозначно можно утверждать, что эта книга указывает верный путь к совершенству, и вы приобретете с дюжину ценных очков, приближающих вас к заветному званию. На самом деле Visual C++ и MFC представляют собой слишком крупный и сложный программный продукт, чтобы овладеть им в полной мере после прочтения единственной книги. Свобода в его применении приходит не сразу, но с практикой. Начните с известного материала и постепенно посчитайте, "что к чему" в MFC.

Изложение в этой книге подчинено методике постепенного овладения материалом. Шаг за шагом излагаются способы применения Visual C++ Interactive Development Environment (IDE) и методы программирования при помощи MFC. Если у вас нет предварительных знаний о программировании с использованием MFC, не стоит впадать в отчаяние. Просто вам придется внимательно проработать несколько начальных глав этой книги. И довольно скоро вы будете иметь рабочую программу MFC, NotePod, которую можно использовать при редактировании текстовых файлов. Кроме этого, вы узнаете принцип работы программы NotePod. В последующих главах содержатся более сложные проекты, включая PaintORama и WordZilla. Если вам нравится работать в процессе обучения, то эта книга — для вас.

Несмотря на простой подход к изложению материала, эта книга — не для новичков. Авторы исходят из того, что вы знакомы с языком C++ и его стандартными библиотеками. Если это не так, необходимо воспользоваться книгой *Object-*

Oriented Programming in C++ Роберта Лафора (Robert Lafore) (Corte Madera, California: The Waite Group Press). Если ваш C++ немного староват, возможно, у вас будут небольшие проблемы в процессе изучения примеров программ. Можно также воспользоваться книгой *The Visual C++ Programmer's Reference* (Scottsdale, Arizona: The Coriolis Group), которая чрезвычайно полезна в случае возникновения затруднений, связанных с синтаксисом или операциями языка C++.

При изучении материала книги необходимо вводить и выполнять каждый проект. Только в этом случае можно с пониманием освоить его. Вероятно, вы обратите внимание на то, что материал одних глав посвящен конкретным проектам, а в других содержатся пояснения к ним. Если это возможно, изучайте главу с пояснениями сразу же после рассмотрения главы, в которой содержится собственно проект. Таким образом, во время чтения соответствующих объяснений типовые проекты будут свежи в вашей памяти.

В этой книге первостепенное внимание уделено основополагающим принципам, однако базовые принципы MFC трудно причислить к "скучным материям". Вы составите собственное представление о некоторых увлекательных технологиях, среди которых:

- Архитектура DocView
- Средства управления ActiveX
- FormView
- Базы данных ODBC
- Работа с сетями и Internet

Сопровождающий CD-ROM содержит весь исходный код примеров программ, набор полезных MFC-утилит и самую необходимую техническую информацию. Хочется надеяться, что вы будете читать эту книгу с таким же упоением, которое мы испытывали при ее написании.

Глава 1

Создание вашего первого приложения: изучение методов использования VC++

Система Visual C++ позволяет создавать программы, аналогичные по сложности Windows Notepad, не написав при этом ни одной строки кода. Рассмотрим метод создания собственного редактора пользователя.

Знакома ли вам подобная ситуация?

Представьте себе празднование Рождества или, еще лучше, своего дня рождения. Пирог уже съеден, кредитные карточки уже просмотрены и подтверждены, а ваши обязательства забыты, прощены или проигнорированы. Самое время перейти к делу. Там, на полу, в центре комнаты, в нетронутой упаковке находится предмет, выстраданный в ваших мечтах: Gizmo-2000. На данном этапе не имеет большого значения, что собой представляет Gizmo-2000 — компьютер нового поколения, современный радиопередатчик или электронный микроскоп. Важнее всего, что это является самым большим, наиболее современным, внушительным Gizmo на планете, и вы не можете дождаться момента его запуска.

Естественно, нужно учитывать одну небольшую проблему. Справа от вашего нового блестящего Gizmo-2000 находится такая большая стопка книг, какую вам до сих пор и видеть не приходилось. Книга, венчающая эту стопку, озаглавлена *Исторический экскурс и принципы работы с Gizmo. Достаточно основательное и серьезное изложение начального курса.*

У вас существует следующий выбор:

A. Аккуратно запакуйте Gizmo-2000 обратно в картонную коробку и приступайте к планированию расписания занятий с учетом того, чтобы обучающие курсы по превращению вас в гуру Gizmo проходило в удобное время.

B. Найдите кабель питания, включите Gizmo в розетку и нажмите несколько кнопок с тем, чтобы наблюдать за работой этого технического чуда.

Если был выбран пункт A, то эта книга, вероятно, вам не подойдет. С другой стороны, если был выбран пункт B, а пункту A не было уделено должное внимание, то эта книга как раз для вас и предназначена.

Данная книга адресована исследователям — для тех, которые сначала включат Gizmo, а потом будут выяснять, как и почему он работает. Конечно, такого рода исследование может занять много времени и представлять определенную опасность. Подобно переселенцам на дикий Запад конца XVIII века, вы нуждаетесь в четком руководстве — и оно находится перед вами. Авторы книги помогут вам быстро добиться поставленной цели, не рискуя при этом попасть в лапы медведя.

Готовы ли вы к этому? Если так, то двигаемся дальше.

Запуск VC++

Самый быстрый способ запуска Visual C++ состоит в том, что выбирается соответствующий ярлык в меню Start, как показано на рис. 1.1. Обратите внимание на то, что Visual C++ находится внутри папки Visual Studio 6, которая размещается в основной папке Programs. (Возможно, используемая копия Visual C++ 6 была приобретена как отдельный программный продукт, или же в составе пакета, включающего C++, Visual Basic и большой набор других инструментальных средств разработки.)

Интегрированная среда разработки (IDE — Integrated Development Environment), используемая для написания программ на языке Visual C++, немного напоминает мастерскую художника: здесь размещаются "под одной крышей" все инструментальные средства, необходимые для написания программ. Переходим к объяснению принципов совместной работы различных компонентов Visual C++ IDE.

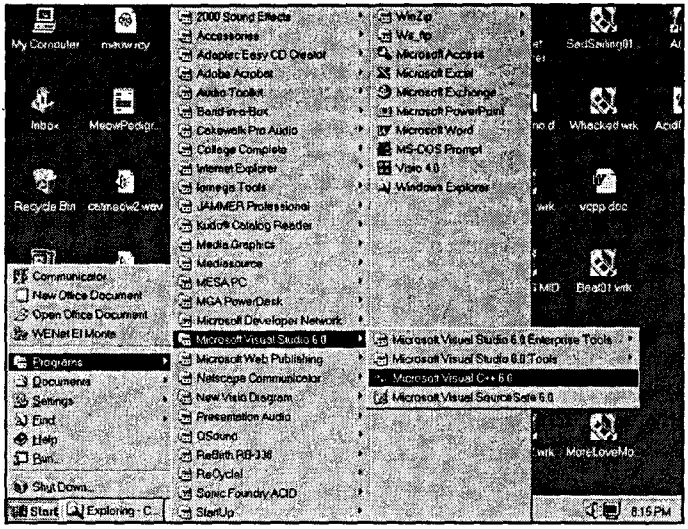


РИСУНОК 1.1
Запуск Visual C++ 6 при помощи меню Start.

Что представляет собой Visual C++?

Visual C++ включает следующие основные компоненты:

- Редактор (Editor), используемый для ввода, просмотра и проверки исходного кода C++.
- Компилятор (Compiler), выполняющий трансляцию исходного кода C++ в объектный код.
- Компоновщик (Linker), создающий выполняемые файлы за счет объединения объектного кода и библиотечных модулей.
- Библиотеки (Libraries), поддерживающие предварительно написанные модули, которые можно применять в пользовательских программах. Одна из наиболее важных библиотек включает Microsoft Foundation Classes (базовые классы Microsoft, или MFC), используемые при написании программ, работающих под управлением Microsoft Windows. Помимо этого, стандартная библиотека C++ поддерживает операции ввода-вывода и другие стандартные возможности языка C++.
- Другие инструментальные средства, включая AppWizard, ClassWizard и Resource Editor (Редактор ресурсов). AppWizard будет еще рассматриваться в этой главе, а дополнительные инструментальные средства — в последующих главах.

Использование Visual C++

При запуске Visual C++ на короткое время появляется красочная заставка. После ее исчезновения отображается рабочий экран Visual C++, выдержанный в строгой цветовой гамме, как показано на рис. 1.2. Намек на цвет появляется только в вездесущем окне Tip Wizard (Мастер советов), всплывающем при каждом запуске Visual C++. Если нет необходимости в ежедневном углублении своих познаний, Tip Wizard можно отключить, сбросив флажок Show Tip At Startup (Показывать совет при запуске).

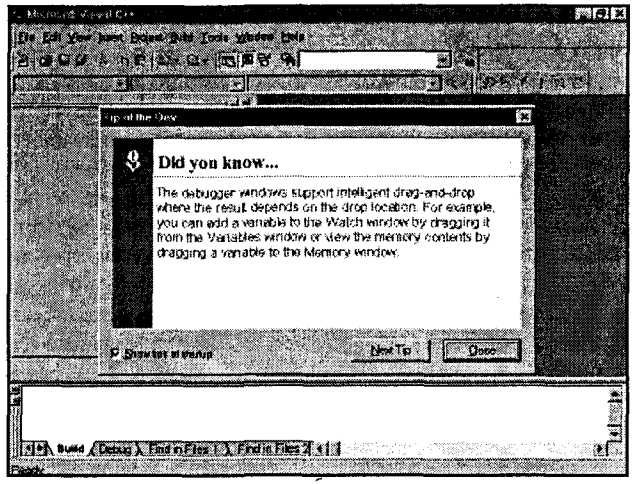


РИСУНОК 1.2
Открытое окно Tip Wizard в Visual C++.

Рабочий экран Visual C++ включает три окна:

- Окно Project Workspace (Рабочая область проекта) появляется с левого края экрана. Это окно предназначено для оказания помощи при управлении большими программами, включающими множество файлов исходного кода.
- Окно Editor (Редактор) появляется справа от окна Project Workspace. Это окно используется для ввода и проверки исходного кода.
- Окно Output (Вывод) появляется в нижней части экрана. В этом окне отображаются сообщения о ходе выполнения, о возникающих ошибках и резюме, относящееся к выполняемым командам. Например, сообщения об ошибках компиляции будут появляться в окне Output при попытке откомпилировать программу, содержащую ошибки.

Проект NotePod: предварительное обсуждение

На самом деле в Visual C++ ничего не происходит до тех пор, пока не создается проект, — потому пришло время для выполнения этого шага. Вместо рассмотрения бесполезного примера, давайте создадим реальную программу на языке Visual C++, которая будет выполнять хоть что-нибудь полезное. Здесь также не будет рассматриваться некая скучная программа типа "Hello World". Вместо этого создается промышленный текстовый редактор, включающий меню и панель инструментов — пригодный даже для печати протоколов спортивных состязаний и обеспечивающий возможность предварительного просмотра печати. Наш проект выглядит аналогично программе Windows Notepad, только на самом деле он будет лучшим. Поскольку авторы питают уважение к классическому фильму *Invasion of the Body Snatchers* и не хотят допустить путаницы рассматриваемого примера с Windows Notepad, этот проект получил название *NotePod.* (В конце концов, бытует мнение, что Notepad далек от совершенства.)

При создании нового проекта в Visual C++ необходимо начать с выбора команды File | New (Файл | Создать) из главного меню. В появившемся диалоговом

окне выберите закладку Projects (Проекты) и, далее, конкретный тип проекта из списка проектов.

К сожалению, Visual C++ предлагает для выбора обширный список различных типов проекта. Что при этом плохо — так это то, что проекты выводятся в простом алфавитном порядке. При этом выбор подходящего проекта немного напоминает выбор незнакомого блюда из большого меню — если вы впервые находитесь в этом ресторане, то должны будете положиться на простое везение.

Однако в данное время вы не находитесь в столь затруднительном положении — на столе лежит наше руководство. Просто просмотрите список, найдите пункт "MFC AppWizard (exe) ", а затем выберите его, как показано на рис.1.3.

В дополнение к выбору типа проекта диалоговое окно New также предлагает озаглавить проект и определить место его размещения. Введите имя проекта (NotePod) в текстовое поле Project Name (Имя проекта). После ввода имени проекта оно добавляется в конец текстового поля Location (Размещение), находящегося под полем Project Name.

Выбираемое имя проекта выполняет несколько задач. Во-первых, оно является именем для вашей выполняемой программы, полученной после сборки проекта. Во-вторых, оно используется как имя подкаталога (называемого *рабочей областью (workspace)*, говоря языком Visual C++). Это то место, где будут автоматически создаваться файлы проекта. В-третьих, при создании проекта Visual C++ это имя используется для образования имен классов приложения.

По умолчанию при создании нового проекта Visual C++ создает новую рабочую область. При необходимости можно щелкнуть на кнопке, находящейся за текстовым полем Location, для выбора местоположения проекта. Для проекта NotePod введите имя "NotePod" в текстовом поле Project Name и используйте значения по умолчанию для всех других опций. По завершению этой работы щелкните на кнопке OK для начала выполнения Visual C++ AppWizard.

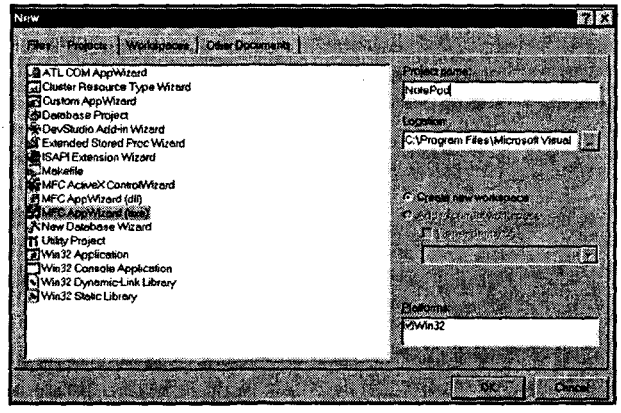


РИСУНОК 1.3
Выбор проекта AppWizard.

Начинаем работать с AppWizard

Visual C++ AppWizard — это генератор объектного кода. Относитесь к нему как к персональному помощнику программиста. AppWizard сопровождает вас при прохождении набора диалоговых окон, запрашивая о том, какими свойствами долж-

но обладать ваше приложение. При наличии всей необходимой информации AppWizard генерирует каркас (skeleton) пользовательского приложения. Этот каркас программы содержит указанные вами свойства, и его можно продолжать настраивать с целью предоставления ему специфических возможностей, отличающих эту программу от любого другого приложения Windows. Таким образом можно "сделать все своеобразно", избрав собственный путь.

Необходимо учитывать одно важное обстоятельство, имеющее отношение к AppWizard: этот мастер используется только в начале работы над проектом. Лишь в этом случае можно использовать AppWizard. Упомянутый программный продукт используется для создания лишь начального приложения, его нельзя применять в процессе поддержки или расширения возможностей вашей программы. Вследствие этого необходимо заниматься планированием. Нужно твердо знать, *что* именно должна выполнять ваша программа, прежде чем ее запускать. Невозможно вернуться в AppWizard для того, чтобы добавить предварительный просмотр печати или поддержку базы данных после завершения работы с ним.

Четко уяснив этот момент, рассмотрим, каким образом AppWizard создает программу NotePod.

AppWizard 1: придание приложению определенного стиля

Как видно из рис. 1.4, AppWizard сначала запрашивает о том, какой тип приложения необходимо создать. На выбор предлагаются:

- *Single Document Interface (SDI, или однодокументный интерфейс)* — Этот тип приложения позволяет открывать только один документ в каждый момент времени. Документ автоматически помещается в основном окне вашего приложения, не оставляя места для дополнительных документов. Приложение Windows Notepad является приложением SDI; при открытии текстового файла при помощи меню File новый файл помещается на место текущего. Более подробная информация о приложениях SDI содержится в главе 11.

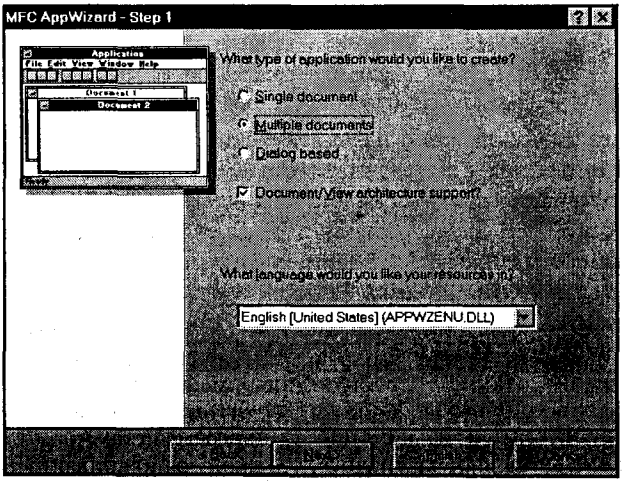


РИСУНОК 1.4

Окно MFC AppWizard Step 1: Выбор стиля вашего приложения.

- *Multiple Document Interface (MDI, или многодокументный интерфейс)* — Этот тип приложения позволяет одновременно открывать несколько документов. Все известные программные продукты Microsoft Office являются приложениями MDI. Например, в Excel можно одновременно открывать несколько электронных таблиц. При этом каждая электронная таблица появляется в собственном окне, и каждое индивидуальное окно документа остается внутри основного окна приложения. Наша программа NotePod является приложением MDI.
- *Dialog based (Основанный на диалоге)* — Этот тип приложения использует диалоговое окно в качестве главного. Приложения подобного типа можно часто встретить в форме программ-утилит типа приложения Date/Time Properties, которое устанавливает дату и время в Windows 95. Visual C++ обеспечивает доступ к редактору диалоговых окон для разработки внешнего вида вашего приложения, основанного на использовании диалоговых окон. Приложения такого типа используют относительно небольшое количество классов, так что в них легко разобраться на этапе создания. Более полная информация, связанная с применением подобного вида приложений, находится в главе 3.

Диалоговое окно MFC AppWizard Step 1 также позволяет определить две специализированных опции. Первая опция появилась в версии Visual C++ 6 и дает возможность генерировать AppWizard более простой код для некоторых видов приложений. Для включения этой опции потребуется установить флажок Document/View Architecture Support. Большую часть времени вы будете удовлетворены значением этой опции по умолчанию (флажок установлен). Более подробная информация об архитектуре Document/View содержится в главе 12. Вторая опция позволяет сохранять ваши ресурсы, подобные меню и системным подсказкам, на ином, не обязательно английском, языке. Необходимый язык выбирается из раскрывающегося списка, который находится в диалоговом окне.

Попробуем создать для программы NotePod приложение MDI, оставляя установленным флажок Document/View и выбрав для ресурса DLL, включающую английский язык. Если указанный выбор осуществлен, щелкните на кнопке Next, и теперь можно перейти к выполнению второго этапа (Step 2).

AppWizard 2: данные, данные, повсюду данные

Следующий экран AppWizard включает запрос о типе поддерживаемой базы данных. AppWizard автоматически создает код, позволяющий обращаться к базам данных при помощи стандарта Open Database Connectivity (ODBC), который обеспечивает практически универсальный доступ к сотням различных баз данных. AppWizard также позволяет пользователю обращаться к базам данных, использующим механизм Jet Engine компании Microsoft. Именно этот механизм используется в Access и Visual Basic. В Visual C++ 6 можно также использовать объекты ActiveX Data Objects (ADO) для получения доступа к технологии связывания и внедрения объектов (OLE — Object Linking and Embedding) для провайдера баз данных. (Более подробную информацию по упомянутым вопросам можно получить из глав 19 и 20.) Конечно, если посмотреть на варианты выбора, предлагаемые AppWizard на рис. 1.5, то ни ODBC, ни ADO, ни OLE DB вы не сможете обнаружить. Вместо этого необходимо выбрать один из четырех уровней поддержки базы

данных, которые кажутся немного неопределенными. В конце концов, как можно узнать, выбирать ли опцию "Header files only" (Только файлы заголовков) или "Database view with file support" (Представление базы данных с файловой поддержкой)?

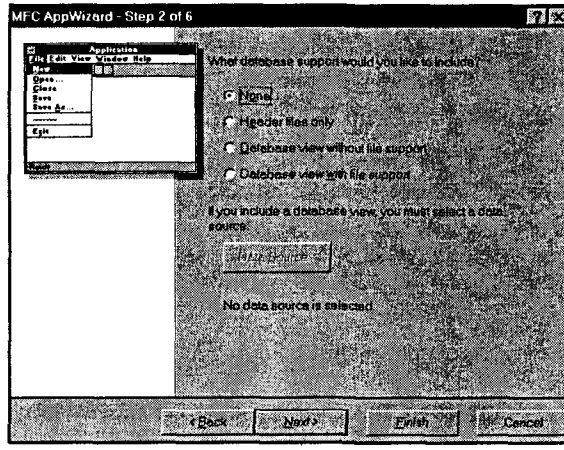


РИСУНОК 1.5

Окно MFC AppWizard Step 2: Определение опции поддержки базы данных.

К счастью, в этом случае ответ довольно прост. Приложение NotePod вообще не нуждается в какой-либо поддержке баз данных, так что истолкование этих опций откладывается до главы 19. (На этом этапе поддержка базы данных со стороны Visual C++ служит для создания приложения базы данных типа "query-and-update" (запросить и обновить).) Выберите опцию None (Нет поддержки) и щелкните на кнопке Next (Далее) для перехода к следующему этапу.

AppWizard 3: вновь нас окружает OLE

На этапе 3 во время диалога с AppWizard можно выбирать необходимый уровень "Compound document support" (Поддержка составного документа). При помощи этого экрана можно указать AppWizard на дополнительную поддержку для модели Component Object Model (COM) компании Microsoft.

Вообще говоря, рассмотрение COM представляет собой неисчерпаемую тему, характеризующуюся многими аспектами. Изложение этой темы займет значительно больше места, чем вся эта книга (или, возможно, даже десяток книг). В главе 17 рассматривается, каким образом COM позволяет вашим программам, написанным на языке Visual C++, использовать тот же самый набор компонентов ActiveX, которые применяются и в программах на языке Visual Basic.

Однако возможности COM значительно шире, чем реализация средств управления ActiveX. При использовании COM можно позволять другим программам *автоматически управлять вашим приложением Visual C++, как будто это приложение* является их компонентом. Возможно, вы уже заметили, что программы типа Microsoft Word позволяют включать в документ электронную таблицу или рисунок, взятый из другого приложения. Эта чудесная возможность — лишь небольшая толика функциональности COM. На рис. 1.6 показаны опции COM, доступные в Visual C++.

Если нет необходимости использовать какую-либо поддержку составного документа в программе NotePod, снимите отметку с флажка ActiveX Controls, кото-

рый находится в нижней части диалогового окна AppWizard Step 3. Затем щелкните на кнопке Next.

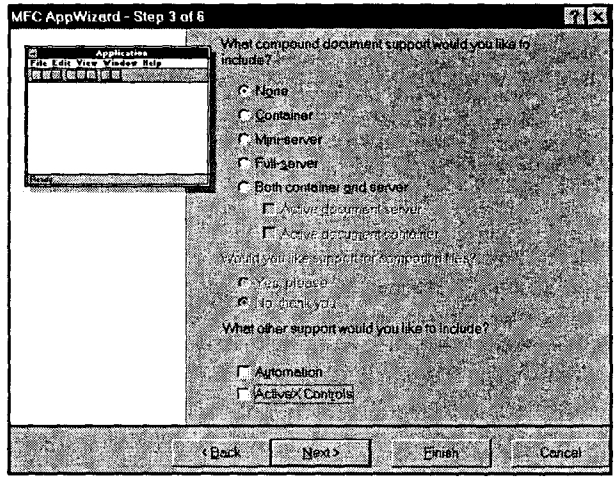


РИСУНОК 1.6
Окно MFC AppWizard Step 3: Установка опций COM.

AppWizard 4: оформление

На этапе 4 AppWizard предоставляет возможность выбора из обширного множества различных свойств, используемых для оформления интерфейса. Четыре опции выбираются по умолчанию:

- *Docking Toolbar (Стыковочная панель инструментов)* — Указывает AppWizard на создание стандартной панели инструментов, расположенной ниже меню вашего приложения. Расположение панели инструментов может изменяться. Это означает, что пользователи вашего приложения могут отсоединять ее, перемещать и затем опять помещать ее на любую границу экрана. Если поступает запрос к AppWizard о создании панели инструментов, то тем самым подразумевается поддержка стандартных кнопок для операций типа вырезания из буфера обмена, копирования, вставки, а также открытие файла и его сохранение.
- *Initial Status Bar (Начальная панель состояния)* — Размещает стандартную панель состояния Windows в нижней части вашего окна приложения. Если необходимо обратиться к панели состояния, то следует обратиться к опции меню вашего приложения, которая позволяет отображать или скрывать эту панель.
- *Printing And Print Preview (Печать и предварительный просмотр печати)* — Обеспечивает отображение стандартного окна предварительного просмотра печати Windows аналогично тому, как отображается диалоговое окно Printer при выборе пользователем команды File|Print (Файл|Печать). Кроме того, Visual C++ добавляет подпрограммы, позволяющие использовать один и тот же код для вывода как на экран, так и на принтер.
- *3D Controls (Трехмерные элементы управления)* — Добавляет код, который отвечает за трехмерное отображение элементов управления Windows, таких как флажки, текстовые поля и переключатели. Эта опция воздействует только на элементы управления, используемые вне диалоговых окон; элементы уп-

равления Windows, используемые внутри диалоговых окон, отображаются в трехмерном виде в любом случае. Активизируя эту опцию, можно быть уверенным в том, что элементы управления всего приложения будут иметь единый вид.

В дополнение к четырем выбранным пунктам AppWizard также позволяет добавлять контекстно-зависимую справку, поддержку MAPI и поддержку сокетов Windows. Поскольку эти возможности используются не так часто, по умолчанию они не выбраны. В отличие от четырех заданных по умолчанию пунктов, каждая из этих дополнительных возможностей требует выполнения значительной дополнительной работы.

Новая возможность в Visual C++ 6 позволяет выбирать панели инструментов стиля ReBar из Internet Explorer вместо традиционных панелей инструментов Windows. В заключение, используя нижнюю часть экрана, отображаемого на этапе 4, можно выбрать, какое количество файлов должно появляться в списке недавно используемых файлов (Most Recently Used, MRU). Значение, заданное по умолчанию и равное 4, слишком мало, так что авторы рекомендуют установить значение этой опции равным 16 (см. рис. 1.7).

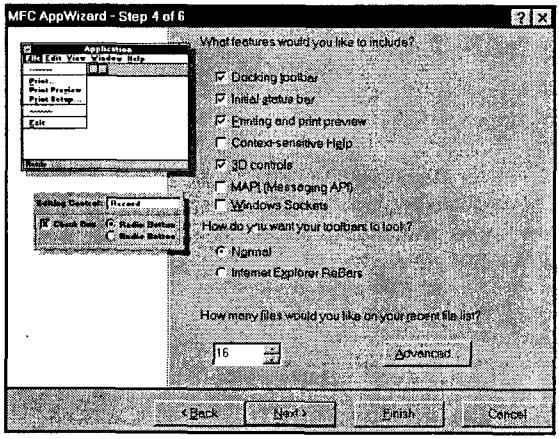


РИСУНОК 1.7
Окно MFC AppWizard Step 4: Выбор элементов оформления интерфейса.

Как только установлено количество недавно используемых файлов, можно двигаться дальше. Однако не щелкайте на кнопке Next — вместо этого щелкните на кнопке Advanced (Дополнительно) и будьте готовы окунуться в мир шаблонов документов.

Небольшое отступление от AppWizard: дайте имя расширению

Если щелкнуть на кнопке Advanced, расположенной на экране AppWizard Step 4, появится диалоговое окно, содержащее страницу свойств с двумя закладками. Одна закладка — Window Styles (Стили Windows) — позволяет выполнять тонкую настройку окон с документами. Эта опция используется редко, поэтому сейчас мы ее не касаемся.

Вторая опция — закладка Document Template Strings (Строки шаблона документа) — состоит из нескольких текстовых полей, и она намного более полезна. Вводя

соответствующие значения в каждое поле, можно зарегистрировать пользовательский тип документа в оболочке Windows.

В результате двойной щелчок на имени файла в Windows Explorer приведет к открытию вашего приложения. Пойдем далее и зарегистрируем тип документа для приложения NotePod. Ниже приведено описание двух шагов, которые следует предпринять:

1. В текстовом поле File Extension (Расширение файла) введите "pod". Это расширение файла для файлов NotePod. (Сами файлы будут содержать только простой текст).
2. При создании в Windows имени типа документа применяется строка, количество символов в которой ограничено шестью. Таким образом, тип документа называется "NotePo" вместо "NotePod". Пользователь не может изменить это. Однако можно изменять каждое из оставшихся текстовых полей для отображения "NotePod" вместо "NotePo". Visual C++ использует эти строки при открытии нового файла либо при поиске типа файла POD в Windows Explorer.

На рис. 1.8 показано завершенное диалоговое окно в том виде, какой оно должно получить. После заполнения каждого из полей щелкните на кнопке Close, далее на кнопке Next в Step 4 и приготовьтесь заполнить предпоследний экран AppWizard. Не останавливайтесь на половине пути: близок уж конец — завершенное приложение вот-вот будет готово!

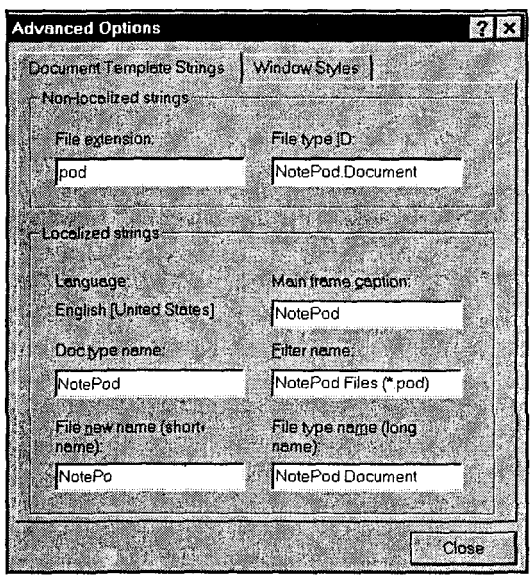


РИСУНОК 1.8 *Небольшое отступление от AppWizard: Выбор дополнительных опций документа.*

AppWizard 5: что осталось сделать

В предыдущей версии Visual C++ диалоговое окно Step 5 содержало только две опции: одна из них обеспечивала возможность автоматической генерации комментариев в исходном коде, а другая помогала определиться, какая связь будет устанавливаться с библиотекой MFC — статическая или динамическая. Обе опции сохранены и в этой версии. В Visual C++ 6 добавлена еще одна опция. С ее помощью обеспечивается возможность генерирования стандартного приложения MFC или программы стиля Windows Explorer, которая отображает окно документа, состоящее из двух панелей.

Для размещения комментариев AppWizard в коде потребуется выбрать соответствующую опцию. При выборе этой опции AppWizard вставит комментарии, указывающие на те области, где необходимо добавить собственный код. Кроме того, в этом случае вы получаете подсказку относительно вида создаваемого кода.

Если пользователь выбирает использование библиотеки MFC в качестве DLL общего доступа, ваши рабочие программы на языке Visual C++ будут намного

меньше по объему (поскольку они могут совместно использовать одну и ту же копию библиотеки). Однако, если планируется распространять приложение, связанное с одной и более общедоступных DLL, то следует учесть необходимость создания условий для распространения также и этих DLL. Напротив, если выбрать для использования статически скомпонованную версию MFC, то программы резко увеличатся в объеме, но зато каждая программа окажется полностью самодостаточной.

Для проекта NotePod создайте проект типа MFC Standard, позвольте AppWizard добавить комментарии и используйте MFC с DLL общего доступа. После активизации необходимых опций (см. рис. 1.9) щелкните на кнопке Next и приготовьтесь увидеть пестрый флажок.

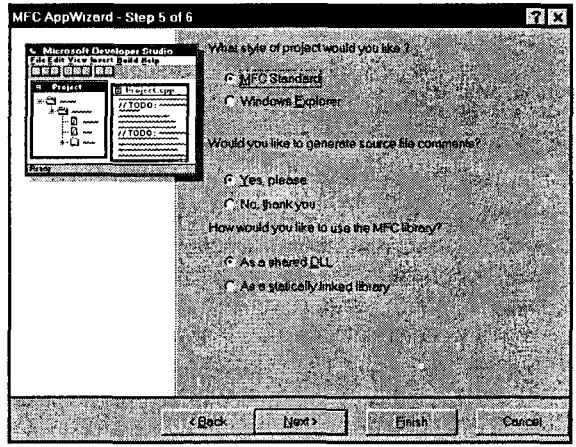


РИСУНОК 1.9
Окно MFC AppWizard Step 5: Стили окон, комментарии и библиотеки.

AppWizard 6: обзор

Если вам показалось, что наша забота о появлении пестрого флажка очень похожа на ребячество, то вы ошибаетесь! Как можно заметить из рис. 1.10, мы теперь находимся на последнем этапе.

Диалоговое окно AppWizard Step 6 отличается от предыдущих одним важным свойством: именно здесь пользователю предоставляется последняя возможность вернуться назад и изменить любой выбор, сделанный на предыдущих этапах. После этого вы будете иметь возможность исправить положение, однако не сможете вернуться к предыдущему состоянию.

В диалоговом окно Step 6 перечислены все классы C++, которые AppWizard собирается генерировать. Сюда также входят несколько текстовых полей, которые позволяют изменять вид кода, генерируемый AppWizard. При прокрутке списка имен класса каждое текстовое поле отображает имя класса (которое можно изменять по своему желанию), а также имя базового класса и файлов, используемых для хранения заголовка класса и его реализации.

Обычно в этом экране вносится лишь одно изменение. Выбирается класс представления вашего приложения (имя класса, завершающееся словом "view"). При этом раскрывающийся список заменяет текстовое поле, которое включало имя базового класса. В MFC класс представления отражает способ, в соответствие с которым ваше приложение взаимодействует с пользователями и информацией, присутствующей на экране. Простым выбором базового класса для представления приложения можно полностью изменять способ работы приложения.

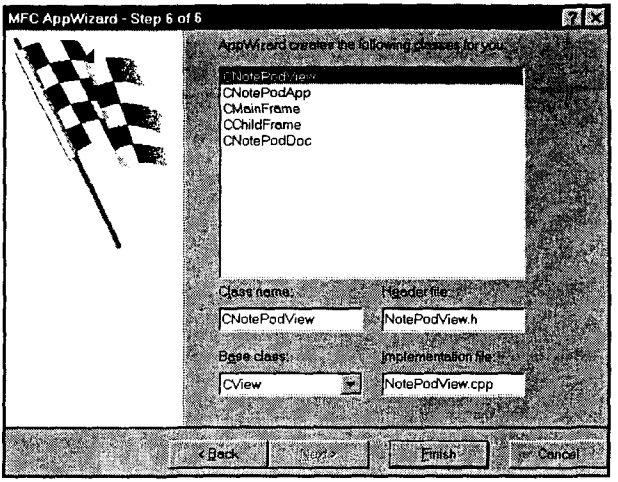


РИСУНОК 1.10
Окно MFC AppWizard Step 6: Изменение базовых классов.

Поскольку приложение NotePod — текстовый редактор, потребуется выбрать базовый класс, который поддерживает соответствующие функциональные возможности. К счастью, MFC включает **CEditView**. Выберите **CEditView** и щелкните на кнопке Finish (Готово).

Сейчас практически все завершено. Как видно из рис. 1.11, AppWizard предоставляет еще одну последнюю возможность вернуться назад. Однако обратите внимание на то, что диалоговое окно New Project Information (Информация о новом проекте) не содержит кнопку Back (Назад). Если необходимо вернуться к Step 6 AppWizard, щелкните на кнопке Cancel (Отменить).

Диалоговое окно New Project Information — это просто резюме всех выборов, которые были сделаны на предыдущих экранах. Прочитайте эту информацию, и если она не вызовет возражений, щелкните на кнопке OK. Мастер AppWizard создаст необходимые файлы и разместит их в указанном каталоге. Теперь можно попрощаться с AppWizard, поскольку его работа над проектом завершена.

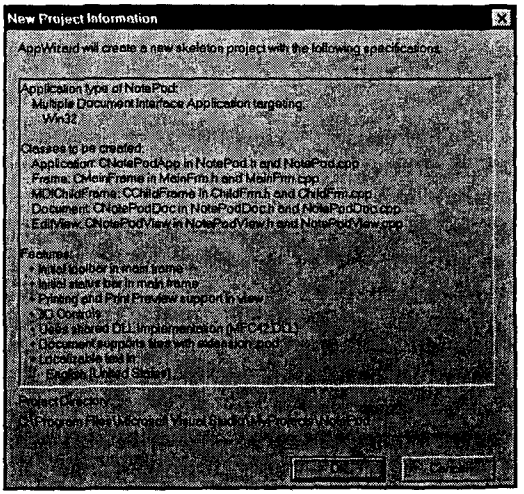


РИСУНОК 1.11
AppWizard: Последняя возможность для изменения вашего выбора.

Исследование вашего проекта

Несмотря на то что AppWizard сошел со сцены, он проделал достаточно большой объем работы, чтобы с благодарностью помнить о нем. Фактически, если для просмотра каталога, выбранного перед запуском AppWizard, используется Windows Explorer, то обнаруживается целый ворох новых файлов (22 штуки), а также пара новых подкаталогов. Как же разобраться в смысле создания всех этих файлов?

Изобретатель языка Pascal, Клаус Вирт (Nicklaus Wirth), как-то сказал: "Подробности — это джунгли, где скрывается Дьявол. Единственное спасение заключается в использовании структур". Visual C++ приходит на помощь, предлагая структурный подход для рассмотрения файлов. Сами по себе файлы не упорядочиваются — вместо этого пользователю предоставляются три различных типа "представлений" структуры проекта:

- FileView (файловое представление) показывает файлы, размещенные в соответствии с логической иерархией файлов исходного кода, заголовков и ресурсов.
- ClassView (представление по классам) отражает ваш проект в виде логической иерархии классов, функций и элементов данных.
- ResourceView (ресурсное представление) упорядочивает ресурсы в программе по типам ресурсов.

К каждому из этих типов представления можно получить доступ посредством окна Workspace, которое появляется по умолчанию в левой части экрана. Если это окно было закрыто, то его можно снова открыть, выбрав команду View | Workspace (Вид | Рабочая область) или нажимая комбинацию клавиш Ctrl+0. Если отображается окно Workspace, можно переключаться между тремя типами представления вашего проекта, выбирая соответствующую закладку в нижней части окна.

Давайте вкратце рассмотрим, как окно Workspace в Visual C++ упорядочивает файлы для проекта NotePod.

Множество файлов: представление FileView

При выборе закладки FileView изначально отображается единственная пиктограмма с маленьким символом + перед ней. Щелкните мышью на каждом символе + для разворачивания списка, пока подобных символов не останется на экране. Список, появившийся на экране, иллюстрируется на рис 1.12.

Файлы проекта находятся в трех папках: Source Files (Файлы исходного кода), Header Files (Файлы заголовков) и Resource Files (Файлы ресурсов). Кроме того, два файла — ReadMe.txt и NotePod.reg — находятся в "корне".

Если вы потратили некоторое время на то, чтобы просмотреть файлы, используя Windows Explorer, то наверняка заметили некоторые отличия. Представление FileView не показывает физическое размещение файлов, как это делает Windows Explorer, а предоставляет логическую структуру.

Щелчок на пиктограмме файла в окне FileView приводит к немедленному открытию файла в основном окне Developer Studio. Попробуйте сделать это непосредственно: щелкните дважды на файле ReadMe.txt и потратьте несколько минут на ознакомление с его содержимым. Из находящейся в нем информации можно понять, что AppWizard четко объясняет значение и цель использования каждого сгенерированного файла.

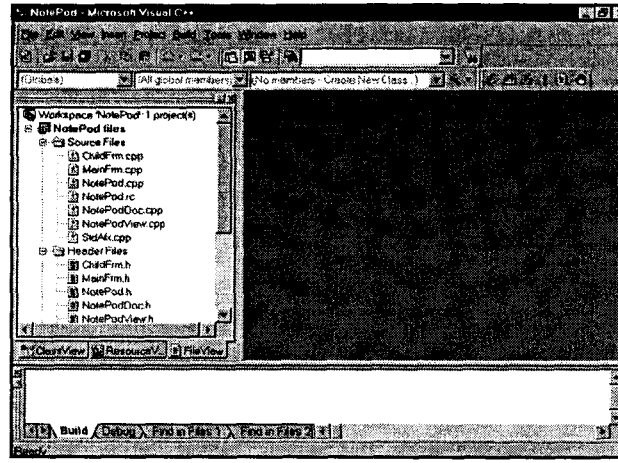


РИСУНОК 1.12

Исследование проекта NotePod: Внешний вид окна FileView.

Хотя FileView наряду с файлом ReadMe.txt AppWizard поддерживает небольшую структуру для вашего приложения, Visual C++ предлагает намного более запоминающуюся структуру: ClassView. Для использования FileView необходимо знать, какие файлы содержат определения и объявления классов. Окно ClassView позволяет просматривать проект в виде набора классов, независимо от физических файлов, содержащих эти классы.

Осознание класса: окно ClassView

После выполнения щелчка на закладке ClassView в окне Workspace появится дерево, похожее на то, которое отображается в окне FileView. Однако ClassView отображает проект скорее как список классов, чем набор папок. После раскрытия одного из классов на экране появится список элементов данных и методов.

Окно ClassView для проекта NotePod показано на рис. 1.13.

Из этого рисунка видно, что проект NotePod содержит шесть классов: **CAboutDIg, CChildFrame, CMainFrame, CNotePodApp, CNotePodDoc** и **CNotePodView**. Все эти классы будут рассмотрены в следующей главе. На данном этапе будет произведен только краткий обзор.

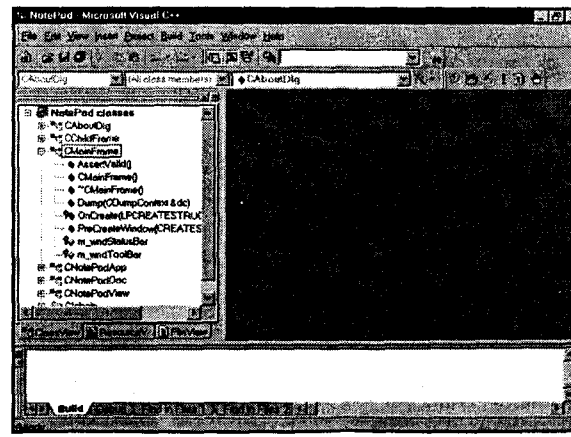


РИСУНОК 1.13

Исследование проекта NotePod: Окно ClassView.

Если щелкнуть на символе + рядом с классом CMainFrame, наименование класса распахнется и появится дерево, отображающее несколько различных пиктограмм. Visual C++ использует эти пиктограммы для упрощения ориентирования в различных частях классов. Например, маленький фиолетовый блок, появляющийся рядом с первой записью, **AssertValid()**, сообщает о том, что это метод. Маленькие цианновые (светлые сине-зеленые) блоки возле названий **m_wndStatusBar** и **m_wndToolBar** сообщают о том, что эти элементы являются элементами данных класса **CMainFrame**.

Закладка ClassView использует две других пиктограммы для отображения спецификатора доступа, связанного с элементом данных или функцией. Маленький ключ — подобный ключу, появляющемуся около элемента **m_wndStatusBar** — означает то, что этот элемент защищен. Если присутствует изображение замка, то элемент является приватным. Если ни одна из этих пиктограмм не появляется, элемент является общедоступным.

Как и в закладке FileView, двойной щелчок на элементе в окне Class View приводит к открытию окна редактирования, где можно выполнять изменения или просто просматривать определения элементов. При этом нет необходимости заниматься размещением связанного файла, поскольку эту функцию выполняет Visual C++. Однако тип открываемого файла зависит от типа элемента, на котором выполняется двойной щелчок. Если пользователь выбирает имя класса или элемент данных, то Visual C++ считает, что необходимо предоставить объявление класса. При этом появляется соответствующий файл заголовка. С другой стороны, если щелкнуть на методе, Visual C++ скорее вызовет файл реализации, который определяет функцию, а не объявление класса.

СОВЕТ

Программы — это нечто большее, чем код

Если ваш опыт программирования ограничивается командной строкой DOS или миром Unix, возникает вопрос, что представляют собой части программы, не включающие код, ведь процесс программирования заключается в создании кода, не так ли? Правильно, но не совсем точно. В программе Windows выполняется сохранение данных, определяющих "внешний вид" или пользовательский интерфейс приложения, отдельно из кода, который обрабатывает эти данные. Подобные элементы данных называются *ресурсами*, они автоматически связываются с выполняемой программой при создании приложения.

Краткий обзор ResourceView

Окно Workspace предоставляет еще один тип представления — окно ResourceView (Ресурсное представление). Подобно другим окнам, здесь происходит упорядочение компонентов программы в иерархическом порядке. Однако ResourceView не работает с исходным кодом проекта; вместо этого происходит организация частей программы, не включающих код.

Прежде чем говорить об общих вещах, давайте рассмотрим несколько специальных сообщений. Если посмотреть на рис. 1.14, то можно увидеть окно Resource View для проекта NotePod. Обратите внимание, что здесь имеется семь папок: Accelerator, Dialog, Icon, Menu, String Table, Toolbar и Version. Подобно папкам,

используемым в окне FileView, эти папки являются виртуальными и применяются просто для того, чтобы обеспечить структуру для приложения — это не физические папки, которые хранят файлы.

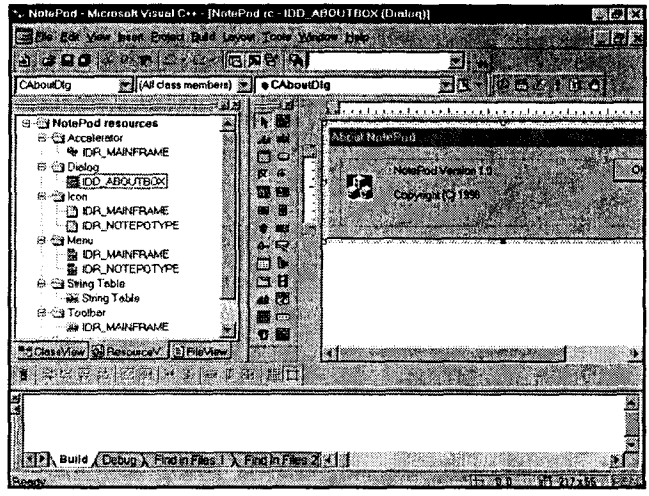


РИСУНОК 1.14
Исследование проекта NotePod: Окно ResourceView.

Если открыть одну из папок, например, Dialog, то можно увидеть, что она включает элемент под названием **IDD_ABOUTBOX**. Двойной щелчок на **IDD_ABOUTBOX** приводит к открытию окна редактирования точно как же, как это происходит в ClassView и FileView. Различие состоит в том, что ResourceView вызывает соответствующий вид редактора для типа ресурса, на котором выполняется двойной щелчок. Если открывается ресурс Dialog, ResourceView вызывает соответствующий редактор диалоговых окон; при открытии ресурса Toolbar вызывается редактор панели инструментов. На рис. 1.14 был активизирован редактор диалога.

В главе 3 будет рассмотрен каждый из редакторов ресурсов.

Активизация проекта NotePod

До сих пор проект NotePod рассматривался с разных точек зрения, но игнорировался один пункт: что происходит, когда NotePod запускается? Пришло время рассмотреть этот вопрос.

Для выполнения программы NotePod сначала необходимо создать выполняемую программу (невозможно запустить на выполнение исходный код). Этот процесс называется *building the application (построением, или сборкой, приложения)*. (Если вы имеете опыт программирования в Unix, то там этот процесс называется *making the application (созданием приложения)*.)

Построение приложения NotePod

Построение приложения не представляет особых трудностей: просто выберите команду Build | Build NotePod.exe из главного меню, щелкните на пиктограмме Build в панели инструментов или нажмите клавишу F7.

Как только запускается процесс формирования, Visual C++ предоставляет информацию пользователю о ходе этого процесса, выводя сообщения в окне Output (Вывод), которое обычно появляется в нижней части экрана. Если окно Output было закрыто, его вновь можно открыть, выбрав команду View | Output из главного меню. Если окно Output отображается, убедитесь, что выбрана закладка Build, иначе будет невозможно наблюдать за процессом компиляции приложения. Формирование проекта NotePod показано на рис. 1.15. Обратите внимание, что окно Output было расширено с тем, чтобы показать все сообщения, сгенерированные Visual C++.

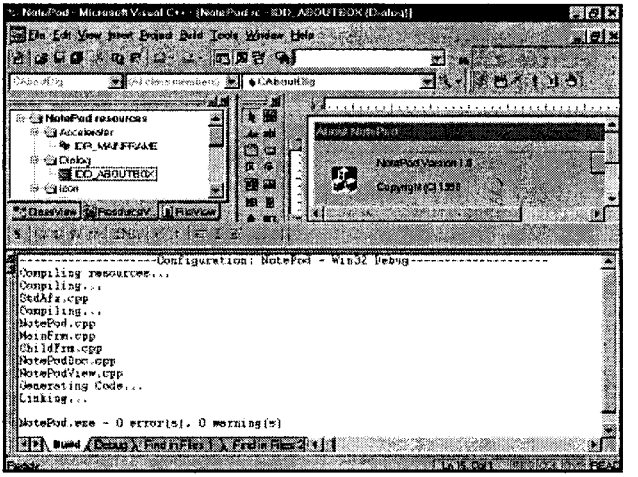


РИСУНОК 1.15
Сообщения компилятора в окне Output.

При первом построении программы NotePod понадобится терпение. Этот процесс может потребовать времени даже на быстром компьютере. Все дальнейшие попытки построения одного и того же проекта занимают существенно меньшее время.

Если компилятор Visual C++ находит какие-либо ошибки, сообщения об ошибках помещаются в окно Output. Однако в Visual C++ сообщения об ошибках специфичны. Если дважды щелкнуть на сообщении об ошибках в окне Output, происходит автоматический переход к месту нахождения ошибки, так что можно установить ее природу.

Тем не менее, на данном этапе вы можете не волноваться по поводу ошибок, поскольку не было написано ни одной строки кода. Вместо этого давайте протестируем программу NotePod.

Мир "шелухи"

Для выполнения приложения NotePod в среде Visual C++ необходимо просто выбрать команду Build | Execute NotePod.exe из главного меню, щелкнуть на пиктограмме Execute (!) в панели инструментов Build, либо нажать комбинацию клавиш Ctrl+F5. После этого вы увидите, что приложение NotePod успешно запускается. (Конечно, нет необходимости выполнять NotePod в среде IDE. Это — настоящая программа Windows, поэтому ее можно запустить двойным щелчком на пиктограмме файла в Windows Explorer или помещением ярлыка в меню Start.)

Первая вещь, на которую необходимо обратить внимание при работе с приложением NotePod — это то, что его можно перемещать по экрану, изменять раз-

меры рабочего окна, свертывать и развертывать. Поскольку, как было сказано, это — обычная программа Windows, нет необходимости создавать какой-то код для поддержки рабочих возможностей. Запущенная программа NotePod показана на рис. 1.16.

Наберите произвольный текст в окне документа точно как же, как в случае использования Notepad. Тем не менее, в отличие от Notepad, можно открывать другое окно документа и через меню Window упорядочивать окна. Можно выполнять копирование текста из одного окна в другое, распечатывать документы или использовать встроенный предварительный просмотр печати. Можно даже заполнить весь диск файлами с расширением .pod.

Приложение NotePod делает все это, не требуя написания ни одной строки кода. Конечно, имеется цена, которую приходится платить за это удобство: цена бессмысленной комфортности. В то время как AppWizard предоставляет пользователю множество опций, в конечном итоге создается фиксированная программа NotePod, работающая точно также, как любая другая программа NotePod, созданная другим "программистом", выбравшим те же самые опции AppWizard. NotePod является представителем мира клонов.

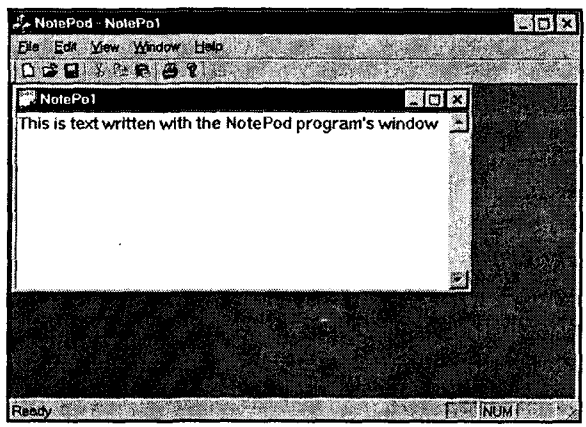


РИСУНОК 1.16
Программа NotePod: Реализована работа с буфером обмена и множеством документов.

Новый всемирный порядок

К счастью, это не конец нашей истории. (В конце концов, это только глава 1.) Хотя AppWizard — представляет собой удобное, экономящее трудозатраты устройство, которому уделяется определенное внимание со стороны Visual C++, на самом деле — это только верхушка айсберга: реальная мощь программирования достигается другим образом, в бесконечной, объектно-ориентированной борьбе классов, с привлечением "темной лошадки", именуемой ClassWizard.

Все программисты являются программистами только тогда, когда, в конечном счете, *они* указывают компьютеру, что ему делать, а не наоборот. При достижении этой цели ClassWizard является и вашим союзником, и вашим наиболее мощным оружием. AppWizard дает вам отправную точку для создания программы, но ClassWizard позволяет воплотить ваши мечты, не выполняя большой объем работы по программированию.

Встречайте меня в WizardBar

Давайте посмотрим, как ClassWizard может помочь настроить NotePod таким образом, чтобы это не был простой аналог любой другой программы NotePod, созданной AppWizard. Прежде чем использовать полнофункциональный ClassWizard, с которым вы столкнетесь в главе 3, остановимся на более доступном инструменте, WizardBar. WizardBar появляется в панели инструментов при запуске Visual C++.

Пользователей часто раздражает, когда в программе NotePod нажатие на клавишу табуляции приводит к пропусканию восьми пробелов. Программа Notepad работает аналогичным образом: Обе программы ведут себя таким образом, поскольку были созданы с использованием встроенного Windows элемента управления многострочным редактированием и это — его поведение, заданное по умолчанию. Однако элементы управления редактированием Windows могут отображать метки табуляции, использующие различный набор табуляторов. Для выполнения этого нужно просто послать сообщение **SetTabStops()**.

Ниже приведен порядок использования WizardBar для изменения проекта NotePod:

1. Выберите панель ClassView в окне Workspace. Дважды щелкните на классе **CNotePodView**, чтобы открыть окно определения класса в редакторе.
2. Щелкните на стрелке, направленной вниз, в панели инструментов для активизации раскрывающегося меню WizardBar (см. рис. 1.17). В меню выберите опцию Add Virtual Function.

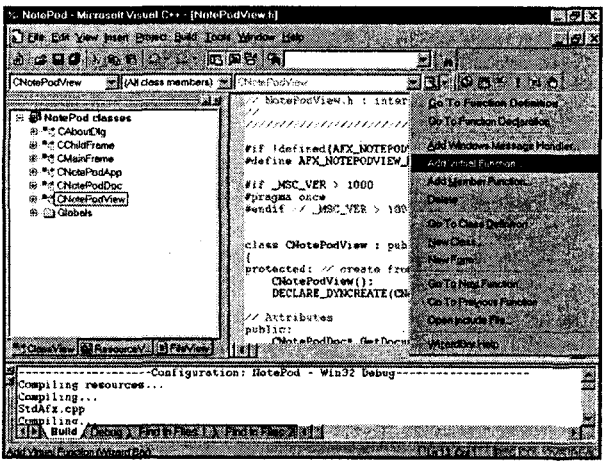


РИСУНОК 1.17
Использование меню опций WizardBar.

3. В появившемся диалоговом окне New Virtual Override выберите функцию **OnInitialUpdate()**. Затем щелкните на кнопке Add And Edit, как показано на рис. 1.18.
4. Visual C++ добавляет новую функцию в файл CNotePodView.cpp, добавляет комментарий и затем осуществляет переключение на редактор. Добавьте строку

```
SetTabStops (16);
```

внутри тела функции, как показано на рис. 1.19. После этого повторно постройте проект и попытайтесь поработать с новой версией NotePod.

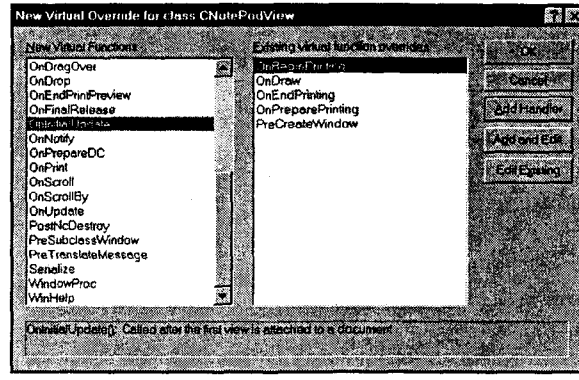


РИСУНОК 1.18
Перекрытие виртуальной функции, используемой WizardBar.

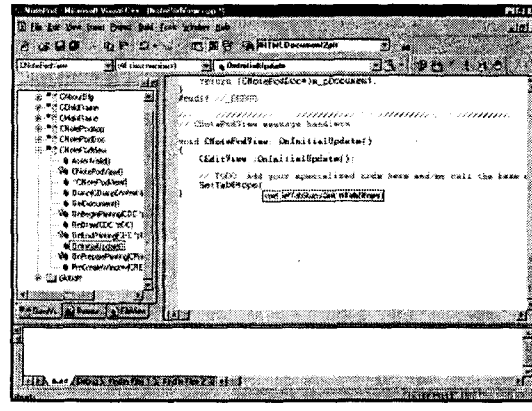


РИСУНОК 1.19
Добавление кода в редакторе Visual C++.

Неожиданная помощь

Если до сих пор пользователю приходилось указывать, щелкать и вводить текст на протяжении работы над проектом NotePod, то на этом этапе он, вероятно, будет очень удивлен в процессе добавлении функции **SetTabStops()** в последнем разделе. При этом, как только было напечатано имя функции, редактор Visual C++ помогает завершить определение функции, отображая типы требуемых параметров во вспомогательном окне, как показано на рис. 1.19.

Microsoft называет эту возможность AutoCompletion. Эта возможность реализована следующим образом:

- Когда пользователь вводит имя объекта. Visual C++ отображает список методов, применимых к этому типу объекта. Список можно просматривать и нажимать клавишу пробела, чтобы предоставить Visual C++ возможность вставить имя функции.
- Когда пользователь вводит имя функции, как это было в случае с **SetTabStops()**, Visual C++ отображает прототип для функции, поэтому можно определить тип передаваемых аргументов.

- Когда функция перегружается, вспомогательное поле, отображаемое Visual C++, включает набор стрелок. Щелкая на стрелках, можно переключаться между различными сигнатурами параметров, которые принимает данное имя функции.

AutoCompletion — это одна из тех привлекательных идей, который делает вашу работу в качестве программиста более производительной. Но что будет в том случае, если необходимо знать больше, чем только имя метода? Чтобы оказать помощь в этом случае, Visual C++ поставляется вместе с CD-ROM, содержащим специализированную версию Microsoft Developer Network (MSDN). Можно получить доступ к этому модулю, нажав клавишу F1. Наряду с документацией, включенной в CD-ROM, Visual C++ 6 включает новый HTML-основанный справочный механизм, который является усовершенствованием раскритикованного средства InfoViewer, используемого в Visual C++ 5.0.

Как видно на рис. 1.20, новый справочный механизм, основанный на использовании HTML, позволяет осуществлять поиск среди списка тем на CD-ROM и затем легко перемещаться по отображающимся в результате документам. Получение справки никогда не было столь простым.

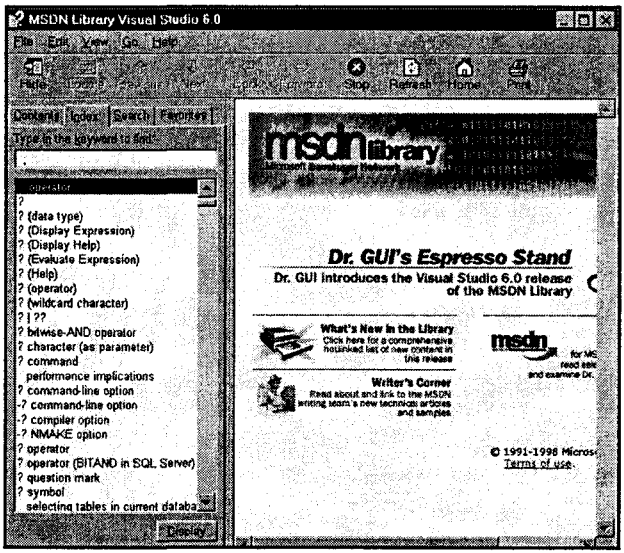


РИСУНОК 1.20
Использование интерактивной документации.

Фокусы с картами или фокусы со шляпами: что это означает?

Если вы плохо знакомы с программированием в среде Windows, то будете несколько озадачены — было рассмотрено много тем с очень небольшим объяснением. Не волнуйтесь, объяснения находятся в следующей главе.

С другой стороны, если вы даже имеете некоторый опыт программирования в среде Windows, то все равно будете несколько озадачены. Все эти указания и

щелчки мышью выглядят подобно карточным фокусам. *В конце концов*, — вы думаете про себя, — *программирование все еще сводится к написанию кода.*

Знаете что? Вы абсолютно правы.

Несмотря на похожее название, Visual C++ — это не среда программирования "указать и щелкнуть", подобно Visual Basic. Программы Visual C++ обычно не являются коллекциями форм, заполняемыми компонентами. Вместо этого написание программ Visual C++ включает модификацию и расширение общего каркаса приложения, основанного на MFC. Чтобы действительно эффективно использовать MFC, необходимо иметь четкое представление о принципах его работы.

Это — тема следующей главы.

Глава 2

Программирование в среде Windows

Подобно старомодному телеграфисту, программы Windows обрабатывают события путем посылки сообщений. В этой главе будет начато составление словаря событий.

"Программы Windows слишком велики по размерам и медлительны!"

Каким бы малым не был ваш опыт работы на персональных компьютерах, вы наверняка слышали это недовольное замечание. И действительно, такое замечание можно услышать неоднократно. И это нарекание имеет под собой основание. Текстовый процессор Windows занимает больше места на жестком диске, и скорость выполнения операций значительно ниже, чем при использовании предшествующей версии, работающей под DOS. Однако, несмотря на указанные недостатки текстового процессора Windows, число его приверженцев все время растет, и что-то не наблюдается "отток" пользователей в пользу его версии для DOS. Если ваш текстовый процессор DOS так хорош, то почему вы перестали его использовать?

Именно здесь и "собака зарыта". Каждый из нас хотел бы использовать в своей работе программное обеспечение, которое и занимает мало места, и отличается еще более высокой скоростью выполнения операций, чем это возможно при работе с Windows, но не за счет возможностей, предлагаемых Windows. При разработке Windows не ставилась задача создания версии DOS, отличающейся оригинала лишь быстротой и компактностью. Целью создания Windows была разработка такой операционной среды, которая свободна от некоторых ограничений, налагаемых DOS как на программистов, так и на пользователей. Поэтому необходимо учесть все обстоятельства и справедливо подойти к оценке Windows. Нарекания на размер Windows и ее быстродействие чем-то сродни критике Годзиллы за то, что он является неуклюжим конькобежцем — да не в этом его амплуа.

Данная глава посвящается рассмотрению тех проблем, которые решает Windows как для пользователей, так и для программистов. Для понимания последующего материала необходимо понять базовую внутреннюю архитектуру именно Windows. Затем будет исследована базовая структура программы Windows — архитектура приложения Windows. Несмотря на то что программы, которые будут созданы, используют библиотеки Visual C++ и MFC, в своей основе они имеют общее происхождение с самыми первыми программами, основанными на использовании API.

В заключение будет предпринята попытка глубокого и всестороннего рассмотрения непосредственно самого MFC. Точно так же, как все программы, основанные на API, совместно используют общую внутреннюю архитектуру, то же справедливо и относительно программ MFC. Сначала авторы предложат вниманию читателей небольшую программу MFC. После освоения этого материала предлагается рассмотреть архитектуру document/view (документ/представление) MFC и код, сгенерированный AppWizard.

Проблемы, возникающие при работе с DOS

Для полного понимания и всесторонней оценки Windows потребуется вернуться в 80-е годы, когда Windows только завоевывала своих почитателей. В то время программисты и пользователи работали с DOS и, естественно, у них были проблемы.

Пользователи находили, что программы командной строки, поддерживаемые DOS, довольно трудно использовать. Поэтому они нуждались в приложениях с графическими пользовательскими интерфейсами (GUI — Graphical User Interface). Программисты, работающие с DOS, теряют все больше времени на разработки всяческих "украшательств", похожих на GUI, и все меньше времени остается на разработку программных продуктов.

Пользователи были неприятно удивлены, когда их приложения DOS оказались не в состоянии поддерживать причудливый новый монитор или принтер, на приобретение которых уже затрачены средства. Тем временем достойный сожаления программист DOS тратит все больше времени на написание интерфейсного кода для новых аппаратных средств.

Пользователи не желали "брать в голову" затруднения компьютерщиков; они просто хотели получать свои заказы. Они хотели одновременно выполнять несколько программ, и были заинтересованы в том, чтобы их программы могли совместно использовать данные, а также работать совместно. При столкновении с такими амбициозными требованиями пользователя у измученного программиста DOS опускались руки.

Как вы помните, для информационных технологий 80-е были далеко не "старыми добрыми временами". Немного ниже речь пойдет о том, как Windows решает эти проблемы. Но сначала рассмотрим каждую проблему более подробно.

Проблемы, связанные с пользовательским интерфейсом

Как вы характеризуете Windows? Если вас можно отнести к подавляющему большинству, среди которого находятся как пользователи, так и программисты, то обычно поступает моментальный ответ, что Windows — это графический интерфейс пользователя в традиции Xerox Star и Apple Macintosh.

Хотя традиционная командная строка DOS (или Unix), возможно, и более эффективна, чем окна, пиктограммы, мыши и указатели GUI, однако большинство пользователей находит, что ввод команд в стиле запроса C:\> довольно затруднителен и непривлекателен. Итак, для большинства пользователей Windows GUI упростил работу за компьютером, сделав ее более понятной.

К сожалению, запросы пользователей по облегчению работы с приложениями GUI усложнили жизнь программистов: подобные приложения затруднительно создавать в среде DOS. Поскольку пользователи заинтересованы в простом внешнем виде программы, написанной в стиле "вижу и чувствую", то программисты DOS тратят большую часть времени, выделенного на разработку приложения, на программирование пользовательского интерфейса, и меньше времени уделяют проблемам, разрешению которых и призваны прежде всего служить их программы.

Проблемы, связанные с зависимостью от аппаратных средств

Традиционные приложения DOS способствовали появлению также и другой проблемы. Эти приложения были органически привязаны к тем аппаратным средствам, для которых они были первоначально разработаны. Возможно, вы помните то время, когда наибольшим спросом пользовалась программа обработки текстов, представляющая собой DOS-версию программы WordPerfect. Популярность, выпавшая на долю WordPerfect, частично объяснялась наличием стека драйверов, что позволило реализовать поддержку принтера фактически любого типа. Но, конечно, драйверы принтера, используемые для WordPerfect, не работали бы для программ Lotus 123 или WordStar.

Подобная аппаратная зависимость создала крупные затруднения также и для программистов. Если необходимо было разработать приложение DOS для рынка PC, то программисту следовало сделать выбор из трех вариантов. Вы могли бы:

- Создать приложение таким образом, чтобы оно работало на уровне "наименьшего общего знаменателя". Например, это означает, что возможно поддержание только CGA-графики в вашей программе. Конечно, это означало, что пользователи, заплатив кругленькую сумму за приобретение платы графического адаптера высокого класса, будут, вероятно, сильно разочарованы, обнаружив, что ваша программа не в состоянии поддерживать расширенные возможности "железа".
- Поддерживать расширенные возможности только выбранных аппаратных устройств, например таких, как принтеры. Этот подход, пожалуй, вполне бы удовлетворил тех пользователей, кто приобрел одно из поддерживаемых устройств, но он малоутешителен для тех, кто выбрал оборудование другого производителя или другой модели.
- Потратить неопределенно большое количество времени и денег, и получить взамен уверенность в том, что ваше приложение работает надлежащим образом на всех новых аппаратных средствах, которые существуют в обращении. Поступая таким образом, вы бы, несомненно, выиграли в глазах ваших заказчиков, но при этом большая часть вашей прибыли от разработки программного обеспечения ушла бы на то, чтобы "идти в ногу" с последними новинками в области разработки аппаратных средств.

Согласитесь, что каждый из этих вариантов чреват дополнительными затруднениями для всех, кто имеет дело с зависимостью от аппаратных средств. Затруднения могут возникать как для пользователя, так и для программиста, а частенько, и для обоих сразу.

Проблемы, связанные с одновременным выполнением

DOS — это *однозадачная операционная система.* Под ее управлением в каждый момент времени выполняется лишь одна программа. Только после того как завершено выполнение одной программы, может запускаться другая. К сожалению, жизнь непредставима в виде одного потока. Большинство из нас непрерывно отвлекается на телефонные разговоры, переписку по электронной почте, встречи и т.д. В такой ситуации операционная система, которая требует, чтобы программа закрывалась перед открытием другой программы, часто является помехой, а не подспорьем.

Дело не только в том, что DOS отказывается совместно выполнять программы, часто программы не в состоянии обмениваться даже самой простой информацией друг с другом. Пользователи желают иметь программы, которые работали бы совместно и без проблем совместно использовали бы информацию, но и DOS, и программы для DOS слишком часто оказываются не на высоте требований.

Windows была разработана, чтобы решить эти три главных проблемы. Предпримем небольшой экскурс и рассмотрим, каким образом (и насколько успешно) Windows достигла этой цели.

Решения, предлагаемые Windows

При создании Windows учитывались недостатки DOS и предполагалось, что они будут преодолены. Рассмотрим подробнее четыре характерных свойства Windows:

- Общий интерфейс пользователя.
- Общесистемный ввод, использующий очередь, и система передачи сообщений.
- Независящая от внешних устройств архитектура ввода-вывода.
- Многозадачность на уровне приложений и взаимодействие между процессами.

Общий интерфейс пользователя

Интерфейс пользователя Windows — это "видимая часть айсберга", которая наиболее подвержена критике. Это и естественно, поскольку это единственная видимая часть Windows! Работа с общим интерфейсом пользователя Windows значительно упрощает освоение компьютера пользователем, облегчая ему жизнь.

Однако для программистов эффект применения Windows более весом. Во-первых, Windows предлагает обширную библиотеку общих, встроенных подпрограмм пользовательского интерфейса. Причем эти подпрограммы будут гарантировано присутствовать в каждой системе, в которой выполняется Windows. Во-вторых, Windows поддерживает встроенный диспетчер, управляющий окнами и меню. Поэтому можно не уделять особого внимания таким деталям интерфейса пользователя, как изменение размеров окон или обработка меню. Появляется возможность больше времени посвятить функциональным аспектам программы.

Архитектура передачи сообщений в Windows

Кроме обеспечения пользователя встроенным диспетчером окон, Windows также реализует систему связи, известную как *queued input* (очередизованный ввод) или передача сообщений в Windows. Рассмотрим, что это означает.

Если создается традиционная программа DOS и необходимо осуществить ввод при помощи мыши или клавиатуры, то часто прибегают к методу, известному как *polling* (опрос). Программа, использующая функции опроса в виде бесконечного цикла, который непрерывно контролирует специфическое устройство (в данном случае, мышь или клавиатура), а затем предпринимает какое-либо действие в тот момент, когда пользователь перемещает мышь или нажимает клавишу.

При использовании опроса компьютер большую часть времени не выполняет никаких функций — "выполняет цикл", как говорят. Конечно, находчивые программисты DOS знают, как устранять циклы опроса, непосредственно перехватывая аппаратные прерывания, сгенерированные устройствами типа мыши, клавиатуры и системного таймера. К сожалению, подобная находчивость часто приводит к появлению своих проблем, поскольку программы конкурируют в процессе перехватывания тех же самых аппаратных прерываний. При состязании невероятной удачи и сверхчеловеческого предвидения возможны всякие неожиданности.

Windows отказывается от такой соревновательности, принимая на себя исключительные права по управлению всеми аппаратными устройствами. Windows непосредственно получает каждое событие, возникающее при системном вводе, которые

генерируются мышью, клавиатурой и таймером, и записывает события. Эти записи затем передаются приложениям, которые в них нуждаются.

Вследствие реализации этой процедуры отпадает необходимость в том, чтобы индивидуальные программы контролировали устройства ввода данных системы. В этом и нет необходимости. Вместо этого Windows направляет сообщения, поступающие от таймера, клавиатуры и мыши соответствующим программам. Все, что ваша программа должна теперь предпринять, — это ответить соответствующим образом.

Также в результате осуществления этого процесса Windows устраняет конкуренцию среди устройств ввода данных. При работе под управлением DOS, если несколько программ были установлены с подпрограммой обработчика прерываний мыши, часто возникали затруднения при определении того, какая программа должна обрабатывать какое-либо сообщение мыши. Однако когда Windows обрабатывает сообщения мыши, то она отслеживает расположения объектов на экране и проверяет корректность передачи событий мыши нужной программе. Таким образом, совместное использование аппаратного устройства программами становится тривиальным. Если в вашей программе нужно, чтобы курсор мыши отображался в виде руки, в то время как все другие программы хотят использовать курсор в виде стрелки, то Windows гарантирует отображение соответствующего вида курсора.

Архитектура ввода-вывода, независимая от внешних устройств

Во времена DOS при покупке высококлассного 21-дюймового монитора продавец должен был также снабдить вас набором программного обеспечения, необходимого для работы таких приложений, как Lotus или WordPerfect. В противном случае с работа с этими приложениями оказалась бы просто невозможной.

С другой стороны, если вы создаете программное обеспечение, то ситуация еще более усложняется. Для каждого поддерживаемого приложения необходимо было знать специфику аппаратных средств, используемых в компьютере с тем, чтобы создавать программу с учетом обеспечения прямого доступа к аппаратным средствам.

Конечно, это справедливо не для *каждого* аппаратного устройства. Например, DOS обеспечивает очень хорошую поддержку различных файловых систем. Нет необходимости знать что-либо относительно числа дорожек и секторов на вашем жестком диске для чтения и записи файлов. Помимо этого можно использовать одну и ту же программу для чтения и записи файлов на жестком или гибком диске (устройство которого отличается от устройства жесткого диска). Операционная система (DOS, в данном случае) поддерживает определенный уровень абстракции (файлы и каталоги), поэтому не нужно иметь дело с аппаратными средствами на физическом уровне (дорожки и секторы).

Таким образом, под управлением DOS запись файлов не зависит от внешних устройств, но вывод на дисплей или принтер таковым не является. Операционная система Windows намеревается устранить этот недостаток, предлагая Graphics Device Interface — GDI (Интерфейс графических устройств). Если программист желает осуществить вывод изображения на монитор, используя Windows, то он не будет иметь дело с монитором как с физическим устройством. Вместо этого нужно создать программу в терминах GDI; она должна работать с каждым новым монитором.

Связь и взаимодействие между процессами

В заключение Windows решила устранить недостаток DOS, заключающийся в невозможности одновременного выполнения различных программ и создания программ, взаимодействующих друг с другом. Более ранние версии Windows обеспечивали неполную многозадачность, например, особо "эгоистичное" приложение могло бы монополизировать время CPU. В 32-разрядных версиях Windows (Windows 95, Windows 98 и Windows NT) Microsoft реализовала более строгую форму многозадачности.

Под управлением Windows приложения могут взаимодействовать двумя способами. Во-первых, системный буфер обмена (Clipboard) позволяет совместно использовать простые данные разными приложениями, даже если эти приложения были созданы различными компаниями. Второй способ задействует технологию Component Object Model (COM), также известную под названием OLE (Object Linking and Embedding). Программы, поддерживающие COM, могут выступать в качестве компонентов других приложений или как вспомогательные программы, обеспечивающие взаимодействие других программ.

Оригинальные методы программирования в среде Windows: использование API

Возможно, вы слышали, что можно использовать Visual C++ для создания приложений Windows, которые организованы почти таким же самым способом, как и приложения DOS. Такие программы называются *консольными приложениями Win32*. К сожалению, консольные приложения не похожи на приложения Windows: они не включают ни одну из тех дружественных возможностей, которые пользователи ожидают от программ Windows.

Поэтому необходимо пройти обучение с тем, чтобы создавать управляемые событиями приложения, если вы хотите, чтобы они работали подобно другим программам Windows. К сожалению (по крайней мере для новичков), фундаментальная структура управляемой событиями программы сильно отличается от простой, иерархической структуры традиционного процедурного приложения. Это структурное изменение является причиной того, что требуется примерно 80 строк программы для написания простой, управляемой событиями программы, типа "Hello World". Программа сложна не только потому, что функционирует в графическом режиме, но и потому что организована совершенно другим образом.

Что означает термин *управляемый событиями*

Традиционные программы организованы иерархическим образом. В языке C, например, функция **main()** находится наверху пирамиды других функций. Она действует подобно генеральному директору, предоставляя работу нескольким подпрограммам верхнего уровня. Аналогично, каждая из этих подпрограмм управляет другими подпрограммами низшего уровня. Все упорядочено, и можно просмотреть всю программу, например, используя отладчик, от начала до конца. Подобный сорт программ большинство из нас изучает с целью их дальнейшего программирования.

Однако программа Windows не организована иерархическим способом в чистом виде. Программа все еще содержит функции, но эти функции были разработаны таким образом, чтобы генерировать отклик (немного в автономном режиме) в ответ на внешние события. Поэтому говорят, что программы Windows *управляются событиями*.

События могут быть самими разнообразными. Например, один вид события генерируется, когда пользователь щелкает мышью или нажимает клавишу. Другой сорт события может быть сгенерирован системным таймером вашего компьютера. Совершенно другие типы событий генерируются различными модулями собственного программного обеспечения внутри Windows. События могут даже быть сгенерированы функциями внутри вашей программы или других программ.

Может вызывать затруднения тот факт, что нет необходимости вызывать функции в ответ на эти события. Вместо этого посылаются сообщения объектам Windows, запрашивая у них необходимость ответить на события предопределенным образом. И даже, что более важно, функция доставки сообщений возлагается на Windows.

Способы генерации событий

На аппаратном уровне каждое устройство ввода Windows управляется прерываниями. Когда нажимается клавиша на клавиатуре, возникает аппаратное прерывание, обычное функционирование Windows приостанавливается и управление передается фрагменту кода, называемому Interrupt Service Routine — ISR (Программа обработки прерываний). ISR действует как драйвер устройства для мыши, клавиатуры и устройств, управляемых таймером.

При получении управления со стороны ISR происходит форматирование специфических данных с дальнейшим их сохранением в предопределенных регистрах. Затем вызывается специальная внутренняя подпрограмма Windows. Эта подпрограмма извлекает данные из регистров и помещает запись о событии в аппаратную очередь Windows. Обычно каждое событие создает новую запись события, но иногда несколько событий записываются в виде единственной записи. Например, подобное действие вызывает удержание клавиши до срабатывания автоповтора.

Аппаратные события ставятся в аппаратную очередь, форматируемую в виде обычных, преобразованных в последовательную форму сообщений Windows и помещенных во входную очередь приложения при помощи вторичного потока, выполняющегося внутри Windows. (Более ранние версии Windows выполняли подобную операцию только тогда, когда приложение запрашивало новое сообщение. Это позволяло приложению Windows, не поддерживающему взаимодействие, блокировать получение и обработку сообщений о событиях для других приложений.) Таким образом, даже при том, что аппаратный ввод является управляемым прерываниями, сообщения события обработано вашим приложением в порядке "первым пришел — первым вышел" (FIFO — first-in, first-out).

Назначение сообщений

Сообщения — это стандартный механизм связи внутри программ Windows, точно так же как вызовы функций — стандартный механизм связи в традиционно организованной программе DOS. Сообщения обычно генерируются одним из трех различных источников:

- События аппаратного ввода генерируются клавиатурой, мышью или таймером. Эти события иногда называются *событиями, поставленными в очередь (enqueued events)*, поскольку они проходят через очередь аппаратного ввода.
- События диспетчера окон генерируются Windows непосредственно в ответ на действия пользователя, выполняющего операции типа перемещения или изменения размеров окна либо выбора пункта меню. Наиболее часто возникающее сообщение подобно типа — **WM_PAINT**, которое посылается для обновления содержимого области окна.
- Индивидуальные окна могут посылать сообщения другим окнам. Например, для указания текстовому полю об удалении текста посылается сообщение **WM_SETTEXT**. Текстовое поле также использует этот механизм для сообщения об изменении текста путем посылки программисту сообщения **EN_CHANGE**.

Сообщения Windows идентифицируются набором мнемонических идентификаторов. Как можно было бы предположить из факта использования символов верхнего регистр, каждое имя замещает целочисленную константу, используемую для идентификации специфического сообщения. Сообщения, непосредственно сгенерированные Windows, обычно начинаются с набора символов "WM", который замещает *сообщение Windows (Windows Message)*. Сообщения, сгенерированные специфическими видами средств управления Windows, используют различные префиксы. Например, сообщения, сгенерированные средствами редактирования, имеют префикс "EN_", который замещает *(уведомление редактирования) (Edit Notification)*.

Изначально все эти константы были определены в главном файле заголовка Windows, windows.h. Однако, по мере разбухания Windows, этот файл заголовка стал слишком большим и был разделен на части. В настоящее время можно найти определения сообщений в файле заголовка winuser.h. Ниже приводятся первые несколько строк из этого файла:

```
# define  WM_NULL                    0x0000
# define  WM_CREATE                  0x0001
# define  WM_DESTROY                 0x0002
# define  WM_MOVE                    0x0003
# define  WM_SIZE                    0x0005
```

Хотя сообщения Windows идентифицируются одиночным целым числом, фактическое сообщение, используемое для связи — это структура C, именуемая **MSG** и включающая шесть различных полей. Эти поля таковы:

```
Typedef  struct  tag  MSG  {
    HWND      hwnd;
    UINT      message;
    WPARAM    wParam;
    LPARAM    lParam;
    DWORD     time;
    POINT     pt;
} MSG;
```

Каждое из полей внутри структуры **MSG**, в свою очередь, определено в терминах других типов данных, также определенных Windows. Использование этих типов данных облегчает экспорт программ Windows для выполнения на процессорах, где размеры фундаментальных данных могут отличаться. Ниже описывается функция каждого из этих шести полей:

- ***HWND hwnd*** — В Windows каждое окно имеет уникальный идентификатор, называемый дескриптором окна. Значение, хранящееся в этом поле — дескриптор окна, которому адресуется сообщение.
- ***UINT message*** — Целое число без знака, которое хранит идентификатор сообщения (например, **WM_PAINT**), обозначающий тип сообщения.
- ***WPARAM wParam*** — Дополнительная информация, которая может быть необходима для обработки сообщения. Например, это поле может содержать идентификатор выбранного пункта меню. Каждый тип сообщения использует это поле для собственной специфической цели. Полю WPARAM могут присваиваться 32-разрядные числа.
- ***LPARAM lParam*** — Используемое подобно **wParam** поле, содержащее дополнительную информацию, которая изменяется с типом сообщения. Поле **lParam** может включать 64-разрядные числа. Имеющийся тип сообщения использует один или оба поля WPARAM и LPARAM в соответствии с типами параметров.
- ***DWORD time*** — Значение системных часов во время генерации сообщения. Это значение используется для размещения сообщения в очереди сообщений.
- ***POINT pt*** — Расположение курсора мыши во время генерации сообщения.

Теперь, когда вы знаете о том, какие бывают события и сообщения, давайте рассмотрим организацию программ Windows, используя подход Software Developer's Kit, который включает непосредственное использование Windows API.

Архитектура приложения Windows

Не удивительно, что программа Windows является большей частью коллекцией окон. Каково же точное определение окна? Спасибо за вопрос. *Окно* — это прямоугольная область на экране, которая включает различные свойства и области. Рассмотрим некоторые термины, связанные с окнами, которых будут необходимы в дальнейшем:

- *Клиентские и неклиентские области окон (client and non-client areas)* — Каждое окно включает части, о которых заботится сама Windows, и части, которые должна обрабатывать ваша программа. Часть, которую обрабатывает ваша программа, называется *клиентской областью*. В этой области осуществляется пользовательский ввод и вывод. Оставшаяся часть окна — это неклиентская область, которую поддерживает Windows.
- *Стиль окна (Windows style)* — стиль окна определяется при его создании. Существует три основных стиля окна:
- ***WS_OVERLAPPEDWINDOW*** — используется для "основного" окна, окна верхнего уровня вашего приложения. Такие окна могут перемещаться и изменять размер, появляясь при этом в любом месте экрана. Стиль **WS_OVERLAPPEDWINDOW** используется для окон верхнего уровня; для таких окон нет включающих их *родительских (parent) окон*, которые могут ограничивать движение по экрану.
- ***WS_POPUPWINDOW*** — используется для диалоговых окон. Всплывающее окно имеет родительское окно, но может "плавать" в области перед родительским окном и может перемещаться вне экранной области родительского окна.

- ***WS_CHILDWINDOW*** — используется для оконных элементов управления. Командная кнопка (типа кнопки OK в типичном диалоговом окне) — дочернее окно, подобно текстовому полю или полосе прокрутки. Дочернее окно получает возможную область перемещения от родительского окна — невозможно выполнять его перемещение вне области перемещения родительского окна.

На рис. 2.1 показан каждый из основных стилей окна.

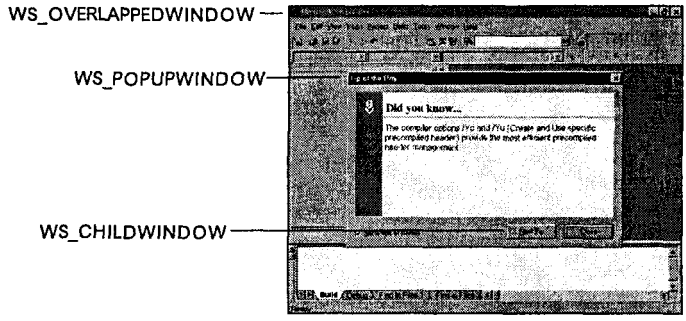


РИСУНОК 2.1 *Основные стили окна.*

Как и следовало ожидать, окна являются самым важным объектом в программах Windows. В программе Windows все коды нажатых клавиш, информация о перемещениях мыши и прерывания таймера посылаются в качестве сообщений некоему выделенному окну, отвечающему за обработку этого действия. Даже элементы пользовательского интерфейса, типа кнопок и текстовых полей, являются специализированными типами дочерних окон.

Порядок работы программы Windows

Как уже упоминалось выше, программы Windows по сравнению с программами DOS организованы по-другому. Программирование приложения Windows, основанного на использовании API, по существу сводится к созданию кода, выполняющего следующих четыре задачи:

- Инициализация
- Реализация
- Вызов цикла сообщений
- Ответ на сообщения

Первых три задачи всегда выполняются функцией под названием **WinMain()**. Функция **WinMain()** является точкой входа для каждой программы Windows, точно так же как функция **main()** является точкой входа для каждой традиционной программы, написанной на языке C. Четвертая задача выполняется функцией, традиционно именуемой **WndProc()** (хотя ее можно называть как угодно). На рис. 2.2 показана организация типовой программы Windows.

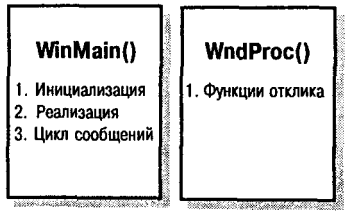


РИСУНОК 2.2 *Организация типовой программы Windows.*

Ваша программа может иметь дополнительные функции, если это необходимо, но каждая программа Windows должна включать, по крайней мере, эти две функции. Давайте рассмотрим простейшую программу, основанную на использовании API, и посмотрим, как эти две функции работают.

Внутри функции WinMain()

После того как Windows загружает вашу программу в память, она вызывает функцию **WinMain()**. Первая задача функции **WinMain()** заключается в определении специфичного для приложения оконного класса для основного окна вашей программы. Давайте посмотрим, как выполняется эта задача.

Инициализация

Для определения оконного класса сначала выбирается имя, например, "MyWindowClass". Затем имя связывается со стилем окна (например, **WS_OVERLAPPEDWINDOW**) и процедурой, которая обработает все сообщения для класса окна. После того как это выполнено, Windows использует ваше определение класса как шаблон при создании экземпляра вашего класса по запросу.

В некотором смысле классы окон подобны классам C++, по крайней мере, в теории, поскольку они действуют подобно шаблонам для создания объектов (окон) и поскольку необходимо определить атрибуты и поведение вашего нового класса. Но с другой стороны, они отличаются от классов C++, поскольку атрибуты, которые нужно определить, уже предварительно определены — необходимо только присвоить значения. В конечном счете классы окон и классы C++ являются все же совершенно различными объектами.

С целью определения класса необходимо сначала создать экземпляр типа Windows **WNDCLASS**. Тип **WNDCLASS** — это структура, содержащая различные поля, которые нужно заполнить. Наиболее важные поля — это имя класса и адрес функции (которая будет создана) для формирования ответов на сообщения. Ниже приводится укороченная версия описываемого кода:

```
static char MyClassName [] = "MyWindowClass";
WNDCLASS wc;
wc.lpfnWndProc = WndProc;
Wc.lpszClassName = MyClassName;
// Дополнительные поля пропущены
```

В этом случае **WndProc** — это адрес процедуры, которой нужно послать сообщение Windows. Также необходимо заполнить восемь других полей, определяя такие понятия, как пиктограмма приложения, кисть, применяемая при окраске фона, и курсор мыши, который используется, когда мышь перемещается поверх окна.

Как только были определены все атрибуты вашего оконного класса, управление передается Windows, которая выбирает и записывает информацию. Этот процесс, называемый *регистрацией класса*, реализуется при помощи функции Windows API **RegisterClass ()**, как показано ниже:

```
RegisterClass ($wc);
```

Реализация

Как только был определен и зарегистрирован класс основного окна вашего приложения, на следующем этапе необходимо создать и отобразить экземпляр основного окна приложения.

Для создания экземпляра окна используется функция API **CreateWindow()**, которая принимает 11 параметров. Если **CreateWindow()** выполняется успешно, то возвращается дескриптор недавно созданного окна. Этот дескриптор используется для выполнения операций с окном. Ниже приводится код, используемый для создания экземпляра оконного класса MyWindowClass:

```
HWND hwnd = CreateWindow ( MyClassName,              // Имя
                           " Hi Mom ",               // Заголовок
                           WS_OVERLAPPEDWINDOW,      // Стиль
                           CW_USEDEFAULT,            // Левый верхний
                                                        угол
                           CW_USEDEFAULT,            // Левый верхний
                                                        угол
                           CW_USEDEFAULT,            // Ширина
                           CW_USEDEFAULT,            // Высота
                           NULL,                     // Родитель
                           NULL,                     // Меню
                           hInstance,                // Дескриптор
                                                        приложения
                           NULL);                    // Дополнитель-
                                                        ные данные
```

Венгерская запись

Если вы работали с Windows, то наверное часто замечали странные имена, подобные *lpfnWndProc* и *lpszClassName*. Это примеры реализации *венгерской записью*, изобретенной одним из лучших программистом легендарной Microsoft Чарльзом Симонаи (Charles Simonyi). В венгерской записи для имен переменных используются префиксы, описывающие их тип и характер содержащейся информации. В рассматриваемом случае название *lpfnWndProc* означает, что *WndProc* является указателем на функцию типа long, а *lpszClassName* означает, что *ClassName* является указателем типа long на строку символов, которая оканчивается нулем. Таким образом, венгерская запись является реликтом тех времен, когда компиляторы не могли выполнять проверку ограничений по типам, которая выполняется компилятором современной версии C++. Однако данная технология используется до сих пор — например, в MFC, где все имена классов начинаются с "C", а все компоненты данных имеют префикс "m_"

Как только основное окно приложения создано, необходимо выполнить еще пару шагов прежде, чем сможет запуститься ваша программа. Сначала, когда создается окно, оно обычно невидимо. Можно легко сделать окно видимым, вызывая метод **ShowWindow()**, как указано ниже:

```
ShowWindow (hwnd, nCmdShow);
```

Первый параметр — это просто дескриптор, который был возвращен функцией **CreateWindow()**. Второй параметр, **nCmdShow**, является одним из параметров, полученных функцией **WinMain()** непосредственно. Значения этих параметров определяет, отображается ли окно в обычном режиме, свернуто или развернуто.

Теперь вы могли бы думать, что после вежливой просьбы Windows отобразить ваше новое окно, все будет в порядке. Однако это не совсем так. Сообщение **WM_PAINT**, которое указывает окну на повторную перерисовку, обладает одним из самых низких приоритетов, так что может пройти некоторое время, прежде чем ваше окно непосредственно отобразится. Можно значительно ускорить этот процесс, вызвав функцию **ShowWindow()** с одновременным обращением к функции

```
UpdateWindow (hwnd);
```

которая сообщает Windows следующее: "Делай это *сейчас*!"

К этому моменту был определен новый класс окна, создано и отображено фактическое окно, но **WinMain()** все еще лишена одной важной возможности — цикла сообщений.

Передача сообщений

Как вы уже слышали ранее, всякий раз когда происходит событие, т.е. пользователь перемещает мышь или нажимает клавишу, Windows создает сообщение и помещает его во входную очередь соответствующего приложения. Ваша программа выбирает сообщение из входной очереди, а затем обрабатывает его.

Эти задачи выполняются процедурой Windows API, именуемой **GetMessage()**. Когда вызывается **GetMessage()**, ей передается адрес структуры **MSG**. Если существует сообщение, ожидающее во входной очереди вашей программы, Windows заполняет все поля структуры **MSG** и возвращает значение **TRUE** функции **WinMain()**. Однако, если сообщение **WM_QUIT** ожидает в очереди сообщений, функция **GetMessage()** возвращает значение **FALSE** и программа завершается. В наиболее общих чертах цикл сообщений выглядит следующим образом:

```
MSG msg;
while(GetMesssage ($msg, NULL, 0,0,))
{
      // процесс обработки сообщений
}
// завершить выполнение, если функция GetMessage() получает сообщение
   WM_QUIT
```

Пока, как можно видеть, этот код не выглядит слишком необычным. Могут представлять интерес дополнительные параметры для **GetMessage()** (обычно используются параметры, приведенные в примере), или может возникнуть вопрос, каким образом генерируется сообщение **WM_QUIT** (когда основное окно приложения закрывается). Но в целом цикл ненамного отличается от ожидаемого — потому не видно ничего по-настоящему сверхъестественного.

"Сверхъестественное наполнение" начинается там, где комментарий сообщает о том, что *здесь обрабатываются сообщения*. Здесь находятся те вещи, которые вы, вероятно, ожидаете увидеть на этом этапе:

```
while(GetMesssage (&msg, NULL, 0,0,))
{
      switch(msg.message)
      {
            case WM_PAINT:
                // Обработать сообщения WM_PAINT
            case WM_SIZE:
                // Обработать сообщения WM_SIZE
                // И т.д.
    }
}
```

Однако вместо этого довольно нудного, но понятного кода, внутри цикла сообщений находятся две непонятных строки:

```
TranslateMessage ($msg);
DispatchMessage ($msg);
```

Вашей первой реакцией будет "Ну и ну!". Затем приходит спасительная мысль, что вы не одиноки, но это не проясняет сложившуюся ситуацию. Хорошо, не будем больше удивляться, попробуем разобраться в сложившейся ситуации.

Предположите, что пользователь вашей программы нажимает клавишу с заглавной буквой "B" на клавиатуре. Как рассматривалось ранее, Windows осуществляет переход внутрь действия и перехватывает событие, направляет его в сообщение и помещает сообщение в очередь. В этом случае Windows генерирует сообщение **WM_KEYDOWN**. (Когда пользователь отпускает клавишу, Windows генерирует другое сообщение: **WM_KEYUP**.) Теперь Windows не имеет ни малейшего представления о том, что будет делать программа при нажатии пользователем клавиши "B". Однако большую часть времени клавиатура используется для набора текста, поэтому, вероятно, что программа отыщет значение кода ASCII для символа "B". Пока это так, все будет хорошо. К сожалению, не всегда все завершается хорошо.

Сообщение **WM_KEYDOWN** передается вам, как только пользователь нажал на клавишу. Однако символами B и b обозначается одна и та же физическая клавиша. Таким образом, если необходимо выяснить код ASCII нажатой клавиши, необходимо также определить состояние клавиши Shift. И на этом проблемы не заканчиваются — необходимо быть осведомленным *и* о состоянии клавиши Caps-Lock. В этом случае, удерживая клавишу Shift, необходимо сгенерировать значение кода ASCII для нижнего регистра b. Итак, необходимо быть осведомленным о состоянии клавиши Shift *и* состоянии клавиши Caps-Lock.

Как вы уже поняли, довольно сложной является сама процедура, позволяющая узнать значение кода ASCII нажатой клавиши. Поскольку эта задача носит слишком общий характер, для данных случаев Windows поддерживает функцию **TranslateMessage()**. При передаче адреса вашего сообщения в **TranslateMessage()** происходит функциональная проверка относительно того, является ли это сообщение результатом нажатия клавиши. Если это так, то происходит вычисление соответствующего значения кода ASCII для нажатой клавиши (принимая во внимание состояние клавиши Shift, клавиши Caps-Lock и даже расположение символов для клавиатур, которые не являются английскими). После вычисления соответствующего значения кода ASCII, функция **TranslateMessage()** генерирует другое сообщение под названием **WM_CHAR**, которое содержит соответствующее значение кода ASCII. Затем новое сообщение пересылается вашей программе. На рис. 2.3 показан итог рассмотренного процесса.

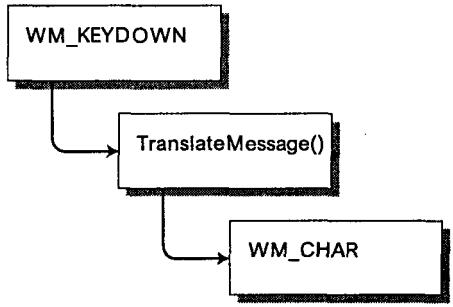


РИСУНОК 2.3 *Работа функции TranslateMessage().*

Применение этой процедуры имеет некоторые особенности. Если не обращаться к функции **TranslateMessage()**, то можно избежать непроизводительных затрат на трансляцию между нажатиями клавиш и их представлениями в виде кодов ASCII. Можно по собственному выбору обращаться либо не обращаться к этой функции. Конечно, трансляция сообщения на ваш собственный манер может привести к неожиданным последствиям.

Но что собой представляет вторая строка цикла сообщения, **DispatchMessage()**? И где фактически происходит обработка ваших сообщений?

Функция **DispatchMessage()** работает примерно следующим образом. Помните ли вы, когда была заполнена структура **WNDCLASS** и зарегистрировано основное окно для вашего приложения? Было заполнено поле с адресом **WNDPROC**, или

оконная *процедура*. Но странное дело, процедура окна не вызывается непосредственно; вместо этого необходимо обратиться к Windows с запросом о вызове этой процедуры. Функции подобного вида называются *процедурами обратного вызова (callback procedures). Оконная процедура (window procedure)* — это процедура обратного вызова, которая обрабатывает сообщения, посланные ей Windows.

Как и можно предположить, в процессе обращения к функции **DispatchMessage()** происходит простая передача сообщения, обнаруженного в очереди сообщений Windows, обратно в Windows вместе с указаниями о передаче его вашей оконной процедуре. Процедура фактически обрабатывает каждое такое сообщение. Рассмотрим эту процедуру подробнее.

Внутри функции WndProc()

После рассмотрения того, как работает функция **WinMain()**, приступим к изучению порядка работы второй важной функции, являющейся частью каждой программы Windows — **WndProc()**. Внутри процедуры **WndProc()** происходит обработка каждого сообщения, пересылаемого Windows. Опять же, функцию **WndProc()** нельзя вызывать непосредственно. Только Windows может вызвать процедуру **WndProc()**.

С точки зрения своего устройства **WndProc()** более проста по сравнению с **WinMain()**. При вызове Windows функции **WndProc()** сначала происходит распаковка и передача четырех полей структуры **MSG** функции **DispatchMessage()**. Этими четырьмя полями являются:

- ***hwnd*** — дескриптор экземпляра окна, пересылающего сообщение.
- ***w*Param** и **lParam** — информационные параметры для специфического сообщения.
- ***message*** — фактический идентификатор сообщения.

Внутри функции **WndProc()** код установлен в виде простого переключателя, содержащегося в параметре сообщения, подобно следующему коду:

```
Switch   (message)
{
   case  WM_PAINT:
         // обработка сообщения WM_PAINT
         return 0;
   case  WM_DESTROY:
         PostQuitMessage(0);
         return 0;
}
```

Как можно заметить, этот код чрезвычайно похож (по крайней мере, концептуально) на код, который авторы предложили вставить в цикл сообщения **WinMain()**.

При близком рассмотрении может возникнуть некоторое беспокойство: что происходит с сообщениями, для которых не найдется условие **case**? Не должны ли вы создавать **case** для каждого сообщения? Ответ отрицательный и это — хорошая новость, поскольку существует более 200 сообщений. Вместо этого в Windows определены заданные по умолчанию действия для всех сообщений, которые могут быть сгенерированы. Но эти действия не происходят автоматически: если не со-

здан соответствующий код для ответа на событие, то вы оказываетесь ответственными за вызов заданного по умолчанию кода для этого события. Создание кода происходит следующим образом: вызывается функция **DefWindowProc()** и ей передаются те же самые параметры, которые Windows передавал вашей функции **WndProc()**.

Обычно создание кода выполняется в виде последней строки функции **WndProc()**. Таким образом, любые сообщения, которые явно не обрабатываются, передаются обратно Windows для заданной по умолчанию обработки.

Резюме о программировании Windows API

Приведем резюме основных пунктов, посвященных программированию Windows API перед тем, как приступить к рассмотрению Visual C++ и Microsoft Foundation Classes:

- Программы Windows организованы существенно иным способом, чем иерархические, прямые, процедурные программы.
- Программа Windows — это коллекция окон, каждое из которых связано с оконной процедурой, реагирующей на события, сгенерированные Windows.
- Каждая программа Windows в обязательном порядке запускает функцию **WinMain()**, которая является ответственной за определение характеристик главного окна приложения. **WinMain()** сначала отображает главное окно, затем выполняет цикл для получения сообщений Windows и возвращает их обратно для последующей обработки.
- Каждое окно в программе Windows связано с процедурой, созданной для организации ответа на сообщения Windows. Эта процедура, по традиции именуемая **WndProc()**, всегда вызывается самой Windows и никогда не вызывается вашей программой.

Программирование в среде Windows: MFC

Хотя можно использовать Visual C++ для создания традиционных программ API подобно тому, как это выполнялось в предыдущем разделе, но реальную выгоду можно получить лишь при написании программ на базе Microsoft Foundation Classes (MFC). На практике, когда большинство программистов говорит о программировании на Visual C++, речь идет прежде всего о программировании при помощи MFC.

В этом разделе сначала проведен общий обзор MFC — что это такое, каковы его функции и почему столь привлекательно использование MFC. Затем исследуется простейшая программа, написанная с использованием MFC. Аналогично тому, как простые программы API обладают завершенной архитектурой, это справедливо и в отношении программ MFC; понимание того, каким образом работают MFC, дает возможность эффективного использования Visual C++. В заключение будет обсуждаться архитектура "документ-представление" MFC, а также указана важность и необходимость ее использования.

Что такое MFC?

Сердцевиной Windows API является C API. Windows был разработан значительно раньше того времени, когда C++ стал применяться достаточно широко. Однако, поскольку все больше разработчиков начало использовать C++ и обнаружило преимущества и мощь инкапсуляции, наследования и полиморфизма, то возрастали потребности формирования объектно-ориентированного подхода к разработке приложений Windows.

Одной из первых компаний, которые стали учитывать этот момент в своей работе, стала Whitewater Group. Сотрудники этой компании создали объектно-ориентированный язык программирования, известный как Actor. В то время как Actor так и не достиг большого успеха, используемый в нем набор библиотек классов был адаптирован компанией Borland для применения в Pascal и C++. Эти библиотеки стали базой для библиотек классов Object-Windows в этих языках.

Компания Microsoft выпустила MFC Version 1.0 одновременно с C/C++ компиляторами в качестве конкурентоспособного продукта. Эта первая версия MFC представляла собой лишь тонкую оболочку вокруг Windows API. Например, вместо использования **HWND** программисты могли создавать объекты **CFrameWnd**. Вместо вызова функций API и передачи **HWND** они могли вызывать функции из класса **CFrameWnd**. MFC 1.0 не пользовался громким успехом главным образом потому, что инструментальные средства, которые использовались им при формировании приложений, были относительно примитивны и ограничены.

Ситуация в корне изменилась с появлением Visual C++ 1.0. Для компании Microsoft создание языка Visual C++ позволило обрести новую почву для совершенствования технологии программирования. Однако более важно то, что инструментальные средства Visual C++ разработаны для совместной работы по созданию и поддержке приложений, основанных на MFC. И сам MFC был расширен, включив в себя каркас приложения или архитектуру.

Библиотека классов в противовес каркасу приложения

Каркасы приложения уже используются длительное время. Каркас приложения представляет собой некое подобие библиотеки наоборот, как показано на рис. 2.4. Библиотека классов содержит классы, которые могут создавать объекты для использования внутри вашего приложения, например, объекты **Button**, **Time** или **Socket**. Но библиотека классов не диктует полную форму и организацию вашей программы.

С другой стороны, каркас приложения является полным приложением, которое может

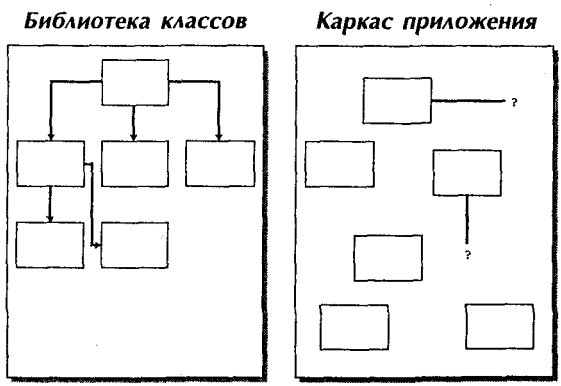


РИСУНОК 2.4 *Отличие библиотеки классов от каркаса приложения.*

настраиваться с целью удовлетворения ваших потребностей. Каркас непосредственно управляется потоком программирования — можно осуществлять настройку, изменяя (или, в объектно-ориентированном мире, перекрывая) способ, при помощи которого эти функции работают.

Большое различие между каркасом приложения и традиционной библиотекой классов заключается в объеме создаваемого кода и количестве повторно используемого кода. Вообще говоря, если каркас приложения выполняет то, что от него ожидается, то это влечет за собой создание намного меньшего объема кода, чем в случае, когда поддерживается каркас и используются относительно независимые объекты библиотеки классов.

Почему используется MFC?

В оставшейся части книги вы будете изучать классы и свойства, поддерживаемые MFC. Здесь же будет только краткое упоминание о них.

В главе 1 вы уже видели первоначальное преимущество, полученное при использовании каркаса приложения MFC. Какой объем кода нужно написать для добавления панели инструментов в ваше приложение NotePod? Ничего не нужно. Код для обработки панели инструментов был унаследован из каркаса приложения.

Какой объем кода нужно создать для поддержи отображения диалоговых окон File Open и File Save? Для поддержи чтения и записи ваших файлов на диске? Для добавления мозаичного или каскадного расположения окон документов в NotePod?

Снова, снова и снова, для этого не нужно создавать код. И это справедливо не только для программы NotePod — в общем случае это верно для всех программ MFC, созданных с использованием библиотек классов MFC и каркасов приложения.

Создание программ, использующих MFC, не является программированием "без создания кода", также этот процесс не является визуальным программированием типа "указать и щелкнуть". Вместо этого программирование в MFC включает *понимание модели программирования MFC и использование инструментальных* средств Visual C++ для изменения каркаса приложения с тем, чтобы достичь поставленных целей. Этот процесс можно назвать программированием при помощи исключений.

Простейшая программа MFC

Прежде чем перейти к следующей главе, давайте потратим немного времени на рассмотрение структуры простейшей программы MFC подобно тому, как это было сделано ранее для программы, основанной на использовании API.

Изначально следует помнить о том, что программа MFC — это программа Windows, точно так же как показанная ранее версия, основанная на использовании API. Это означает, что каждая программа MFC также включает функцию **WinMain()** и функцию **WndProc()**, обрабатывающую сообщения. Тем не менее, в программе MFC вы никогда не встретите ни одну из этих функций (при попытке их нахождения в исходном коде MFC). Эти функции скрыты внутри каркаса приложения.

Если не видно функций **WinMain()** и **WndProc()**, то что же можно обнаружить при рассмотрении программы MFC? Всегда обнаруживаются два объекта: объект главного окна и объект приложения. (На самом деле объект главного окна не является жизненно необходимым, хотя на практике он почти всегда присутствует. Однако каждая программа MFC должна включать объект приложения.)

Для создания простейшей программы MFC (без использования AppWizard) необходимо выбрать Win32 Application из списка New | Projects. Назовите новый проект BDSimpleMFC и щелкните на кнопке OK (см. рис. 2.5.)

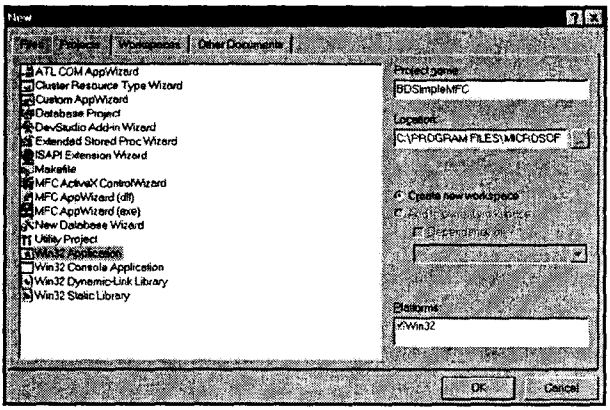


РИСУНОК 2.5
Создание простейшей программы MFC.

Как показано на рис. 2.6, Visual C++ выдает запрос относительно того, какое приложение Windows вы хотите создать. Выберите переключатель An Empty Project (Пустой проект) и щелкните на кнопке Finish (Готово).

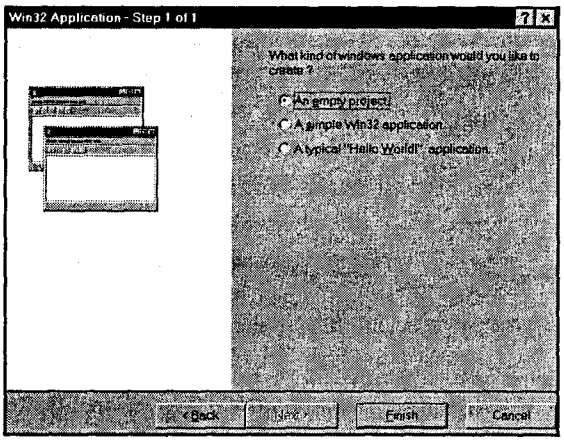


РИСУНОК 2.6
Выбор типа приложения.

Visual C++ отображает диалоговое окно New Project Information (Информация о новом проекте), как показано на рис. 2.7. Щелкните на кнопке OK для продолжения.

Ваша рабочая область окна будет пуста, как показано на рис. 2.8. Для добавления исходного кода C++ выберите команду Fill | New, после чего — опцию C++ Source. Внесите имя файла BDSimpleMFC, как показано на рис. 2.9, а затем щелкните на кнопке OK.

Теперь в окне редактора введите код, показанный на листинге 2.1. Лучше всего, если вы напечатаете код, поскольку при этом сможете рассмотреть все подробности. Однако, при желании можно воспользоваться сопровождающим книгу CD-ROM для копирования нужного файла.

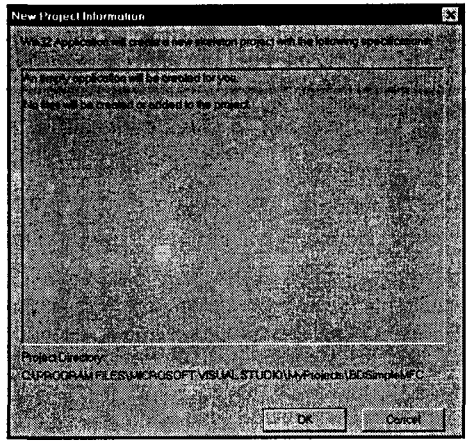


РИСУНОК 2.7
Диалоговое окно New Project Information.

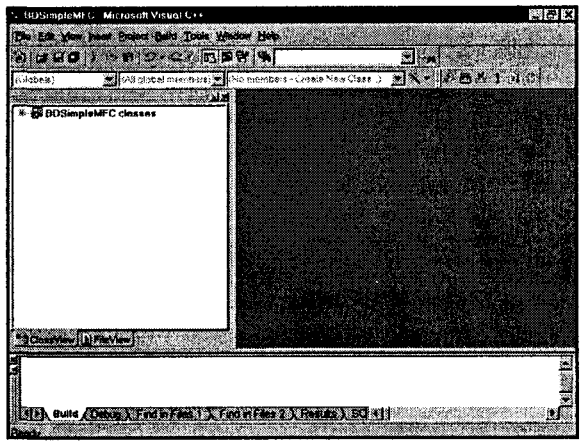


РИСУНОК 2.8
Пустая рабочая область.

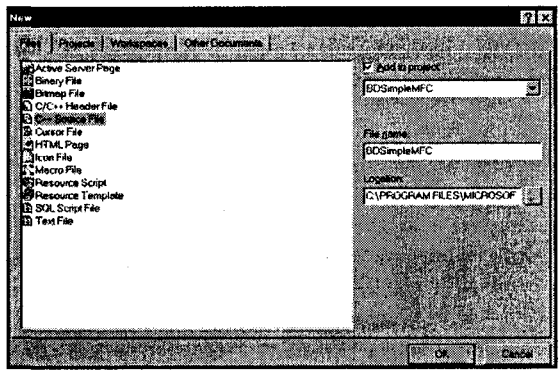


РИСУНОК 2.9
Добавление файла исходного кода C++.

Листинг 2.1 Простейшее приложение MFC.

```
#include <afxwin.h>

// Класс главного окна
class CBDWindow:public CFrameWnd
{
```

```
public:
   CBDWindow()
   {
      Create(0, "A Brain-Dead MFC Window");
   }
};

// Класс приложения
class CBDApp:public CWinApp
{
public:
   virtual BOOL InitInstance()
   {
      m_pMainWnd = new CBDWindow();
      m_pMainWnd->ShowWindow(SW_SHOWMAXIMIZED );
      return TRUE;
   }
};

CBDApp TheOneAndOnlyBrainDeadMFCApp;
```

Затем используйте пункт меню Project | Settings, чтобы обратиться к диалоговому окну Project Settings. Выберите опцию Use MFC In A Shared DLL (Использовать MFC в DLL общего доступа) из окна списка, как показано на рис. 2.10. Щелкните на кнопке OK.

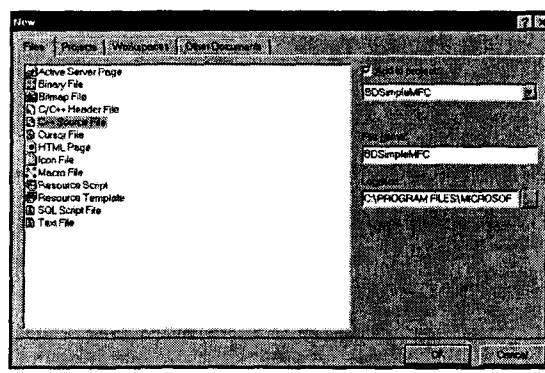


РИСУНОК 2.10
Проверка установок проекта.

Теперь можно скомпилировать и выполнять программу. Как и в главе 1, используйте команду Build | Build BDSimpleMFC для компилирования исходного кода. Проверьте и исправьте любые ошибки, возникшие на этапе компиляции, и при необходимости выполните повторную компиляцию. После успешного завершения компиляции используйте команду Build | Execute для запуска программы на выполнение. На экране появится ваше собственное окно простейшей программы MFC (рис. 2.11). (Изначально окно развернуто, размеры окна были изменены только на рисунке.)

Теперь, когда имеется работающая программа MFC, давайте исследуем, как она работает. Сначала обратите внимание на то, что в исходном коде определены два класса: **CBDWindow**, который представляет окно приложения верхнего уровня, и **CBDApp**, который представляет собственно приложение. Каждый из этих клас-

сов получен из базового класса, от которого были унаследованы значительные функциональные возможности.

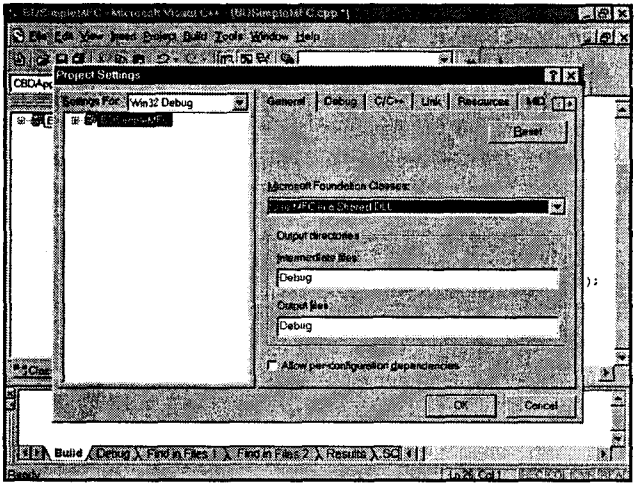


РИСУНОК 2.11
Окно простейшей программы MFC.

Класс **CBDWindow** порожден от класса MFC **CFrameWnd**, а класс **CBDApp** — от класса MFC **CWinApp**. (Объявления базовых классов **CFrameWnd** и **CWinApp** включены в файл заголовка каркаса приложения, afxwin.h.)

Поскольку каждый класс наследует столь много функциональных возможностей, каждый из классов является небольшим и простым. Класс **CBDWindow** только определяет общий конструктор, который вызывает метод **Create()**, унаследованный от класса-предка. **Create()** определяет имя для окна, которое появляется в заголовке окна (см. рис. 2.11).

Класс **CBDApp** просто перекрывает виртуальный метод **InitInstance()**. Внутри метода он создает экземпляр класса **CBDWindow** и присваивает возвращаемый указатель на унаследованный элемент данных **m_pMainWnd**. Затем этот указатель используется для вызова функции **ShowWindow()**, передавая параметр, который заставляет окно отображаться в развернутом виде. В заключение метод возвращает значение **TRUE**, которое сообщает об успешном завершении.

Остается одна деталь, которую часто забывают начинающие программисты MFC. Обратите внимание на то, что программа создает экземпляр приложения (объект **CBDApp**) с *глобальной областью видимости* (*global scope*). Создание этого глобального экземпляра запускает программу на выполнение.

Если вы находите это объяснение сложным, рассмотрите следующую объектно-ориентированную программу Hello World:

```
class HelloWorld
{ public:
   HelloWorld() { cout << "Hello World"; }
};
int void main() { }
HelloWorld h;
```

Обратите внимание, как конструирование глобального объекта **HelloWorld** влечет за собой печать сообщения *Hello World* даже при том, что внутри метода **main()** ничего нет. MFC работает аналогичным образом. Если невозможно будет создать

глобальный экземпляр вашего объекта приложения, то программа не сможет выполняться.

Об MFC всерьез

Мы надеемся, что это краткое введение в MFC разбудит ваш аппетит. Если положение дел кажется несколько туманным, то все в порядке. Вы будете видеть те же самые методы еще множество раз до конца этой книги. Цель этой главы, подобно путешествию по домам звезд Голливуда, состояла в том, чтобы ввести в курс дела и рассмотреть основные понятия лишь в общих чертах. Однако если вы не чувствуете себя уютно, то это вполне объяснимо.

В следующей главе будет показано, как создать FourUp, простую программу Windows на основе MFC, которая позволит вам сыграть в видеопокер. Хотите ли вы создать FourUp? Смелее в бой!

Глава 3

Создание простого приложения на основе диалоговых окон

Приложение FourUp напоминает автомат для игры в видеопокер за исключением одной важной детали: оно не влияет на сумму ваших сбережений.

Старая пословица гласит, что существуют две вещи: законы и сосиски, за изготовлением которых вы никогда не пожелаете понаблюдать. Но если задуматься над этим вопросом, то это высказывание применимо ко многим областям. Например, создание фильма для домашнего видео требует месяцев (или лет) упорных, кропотливых усилий, которые трудно характеризовать как особенно приятные или романтичные, даже если конечный результат и производит подобное впечатление. Законы, сосиски, фильмы, книги, программное обеспечение — в каждом случае многое происходит за сценой перед поднятием занавеса.

Подобные рассуждения справедливы и в отношении программы Windows. Эта программа — не только увенчанный славой код. Фактически значительная часть каждой программы Windows состоит из элементов, которые не являются программным кодом: растровые изображения, курсоры, пиктограммы, диалоговые окна и строки. Эти элементы Windows известны как *ресурсы*.

Ресурсы Windows можно представлять как чрезвычайно специальный вид инициализированных данных. Предположим, что имеется такая строка кода в вашей программе C++:

```
static int nCards[] = { 1, 2, 3, 4, 5 };
```

При компиляции программы данные, составляющие значения, которые хранятся в массиве **nCards**, будут сохранятся наряду с кодом программы в выполняемой программе на вашем диске. При загрузке программы в память значения будут читаться из файла .EXE в память таким образом, что будет возможно их прочтение пользователем.

Данные ресурса Windows организованы несколько иным образом. Они хранятся на жестком диске в файле с расширением .EXE, точно так же как инициализированные данные. Однако при загрузке программы данные о ресурсе *не* загружаются в память наряду с остальной частью программы. Вместо этого данные ресурса остаются на диске, пока они не станут необходимыми. Например, если программа отображает диалоговое окно, данные шаблона диалогового окна динамически восстанавливаются с диска и используются в процессе формирования диалогового окна.

Такая схема обладает двумя преимуществами. Во-первых, поскольку данные ресурса предназначены только для чтения, то они могут быть выгружены из памяти в том случае, если не используются — в случае возникновения необходимости данные автоматически загружаются с диска повторно. Это приводит к тому, что программа требует меньшего объема памяти. Во-вторых, отпадает необходимость в большом количестве малых по размерам файлов, занимающих место на жестком диске. Можете ли вы представить себе трудности, подстерегающие при попытке распространения программы Windows, если необходимо включать все файлы изображения, используемые для каждой инструментальной панели, кнопки и пиктограммы? Работа предстояла бы огромная. Вместо этого данные ресурса позволяют поставить одиночный пакет вашим пользователям. Они не должны знать все рабочие подробности — на самом деле, вероятно, так будет лучше.

Ресурсы и диалоговые окна

В главе 1 было показано, насколько просто использовать AppWizard для создания довольно сложного текстового редактора. Однако, хотя программа несложна для создания, но если просмотреть код, то обнаружится, что она не столь проста для понимания.

Подобная трудность не присуща MFC. Как показано в конце главы 2, минимальная программа MFC более проста и легка для понимания, чем эквивалентная ей программа Windows, основная на использовании API. Так в чем же причина? На первый взгляд, вина за сложность кода NotePod целиком лежит на AppWizard. Но фактически код сложен просто потому, что NotePod поддерживает множество сложных функций — конечно, намного больше, чем для случая минимальной программы MFC.

Никто из программистов, изучающих MFC, не должен преодолевать подобные трудности. Итак, в этой главе продолжится использование AppWizard, будет рассмотрен вопрос о генерировании при помощи AppWizard программы, основанной на диалоговых окнах. Подобная программа более проста по сравнению с NotePod, поскольку имеет только два главных класса: класс приложения (порожденный от **CWndApp**) и класс главного окна (в случае с приложением, основанным на диалоговых окнах, порожденный от **CDialog**). Приложения, основанные на диалоговых окнах, не нуждаются в дополнительных классах документов и представлений, которые необходимы проекту NotePod. Поскольку это значительно проще в реализации, то и приложения, основанные на диалоговых окнах, легче для понимания.

Приложения, основанные на диалоговых окнах, имеют еще одно преимущество по сравнению с большинством программ MFC: можно разрабатывать весь интерфейс пользователя, используя редакторы ресурсов Visual C++, а не программируя код. При создании приложения, основанного на диалоговых окнах, происходит визуальное размещение растровых изображений, кнопок, текстовых полей и других средств управления Windows, и затем визуально изменяются размеры этих элементов и их местоположение.

Приятно слышать, не так ли? Сделаем последнее замечание прежде, чем приступить к сути дела. Не обольщайтесь по поводу того, что только благодаря более простой структуре приложения, основанного на диалоговых окнах, по сравнению с программной, основанной на архитектуре "документ-представление", приложение должно в общем случае оказаться проще. Эти приложения столь же универсальны, как их более сложные собратья — в этом можно не сомневаться.

Начало работы над проектом FourUp

Теперь, продолжая разговор о сравнении различных программ, рассмотрим проект этой главы: FourUp. Программа FourUp подобна играм типа видеопокера, которым увлекаются операторы казино в Лас-Вегасе и Атлантик-Сити. FourUp немного проще, поэтому можно сконцентрироваться на программировании и изучении способов использования редакторов ресурсов Developer Studio.

FourUp не является в действительности интеллектуальной игрой. В начале игры игрок вкладывает в банк $100. На каждом этапе игры (который стоит $2) игрок получает четыре карты. Значения карт не столь важны — существенна лишь масть карты. Если игрок получает две пары (две черви и две трефы, например), то выигрывает 3 доллара. Если игрок получает три карты одной масти, то выигрывает 6 долларов. Если игрок располагает четырьмя картами, то размер выплаты составит 9 долларов.

Конечно, в игре не используются реальные деньги. Если хотите, можете создать программу, где в процессе игры происходит удаленное подключение к банку игрока для внесения выигрыша (или покрытия расходов).

В этой главе показано, как приступить к созданию программы FourUp; завершение работы над этой программой показано в главе 5. На начальном этапе в

процессе создания диалогового окна About, имеющего достаточно профессиональный вид, используются утилиты Visual C++ Dialog Editor и Bitmap Editor. Если же вам не терпится увидеть завершенную программу, обратитесь в главу 5. Или, что более приемлемо, можно воспользоваться текстом программы FourUp, находящимся на сопровождающем книгу CD-ROM.

Создание приложения, основанного на диалоговых окнах, в AppWizard

Запустите Visual C++ и используйте AppWizard для создания начальных файлов приложения FourUp. Процесс создания приложения, основанного на диалоговых окнах, аналогичен проекту NotePod в главе 1:

- Выберите команду File | New из главного меню.
- Выберите проект MFC AppWizard (Exe) из списка проектов, доступных на листе свойств Project.
- Назовите проект FourUp.

В первом окне AppWizard щелкните на переключателе, находящемся в нижней части при получении запроса о виде создаваемого приложения. Обратите внимание, что при выборе переключателя Dialog Based, флажок Document/View становится неактивным (окрашивается серым цветом). Так происходит потому, что архитектура "документ-представление" и архитектура приложения, основанного на диалоговых окнах, несовместимы. В приложении "документ-представление" можно всегда использовать ресурсы диалогового окна. Но в приложении, основанном на диалоговых окнах, нет никаких документов или представлений — имеется только диалоговое окно. На рис. 3.1 показано первое окно AppWizard, которое появляется при создании приложения, основанного на диалоговых окнах.

Опции проекта, основанного на диалоговых окнах

При создании приложения, основанного на диалоговых окнах, AppWizard не должен генерировать большие объемы кода — перед вами ставится лишь несколько вопросов. Вместо шести диалоговых окон приложения, основанные на диалоговых окнах, нуждаются в заполнении только четырех окон.

Во втором окне необходимо выбрать, будут ли добавлены несколько различных возможностей. Рассмотрим список свойств и возможностей, которым должна удовлетворять программа FourUp:

- Выберите флажок About Box для добавления экрана идентификации. Использование редакторов ресурсов Visual C++ начинается с придания профессионального вида диалоговому окну About программы FourUp.
- Вы не будете использовать контекстно-зависимую справку. Она не выбирается по умолчанию, так что можно оставить флажок в том положении, в котором он находится.
- Если нужно использовать трехмерные элементы управления, оставьте соответствующий флажок включенным.
- Если нет необходимости в использовании OLE Automation, ActiveX или сокетов (sockets) Windows, оставьте соответствующие флажки неустановленными.

- В нижней части диалогового окна можно вводить заголовок для диалогового окна. Если необходимо привлечь внимание игроков к игре, то введите "FourUp — Get Rich Now!!!!" (FourUP — Станьте богатым сейчас же!!!!). Убедитесь, что восклицательные знаки не опущены — исследования показали, что они являются эффективным инструментом маркетинга.

Заполненное диалоговое окно AppWizard показано на рис. 3.2.

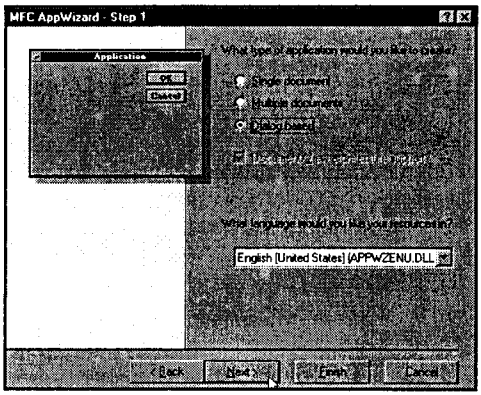


РИСУНОК 3.1 *Запуск проекта, основанного на диалоговых окнах.*

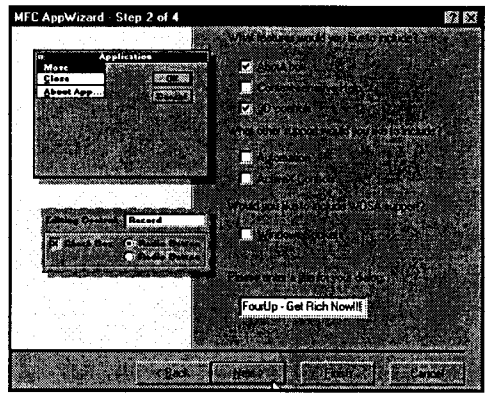


РИСУНОК 3.2 *Опции диалогового окна AppWizard.*

Этап 3 совпадает с этапом 5, который предпринимался при создании программы NotePod AppWizard, за исключением того, что теперь нет возможности создать проект в стиле Windows Explorer. Программный продукт типа Explorer несовместим с приложением, основанным на диалоговых окнах, поскольку в нем используется архитектура "документ-представление". Выберите Yes для добавления комментариев в исходный код и укажите, что необходимо использовать библиотеку MFC в виде общедоступной DLL. Затем щелкните на кнопке Next.

Обзор классов

На рис. 3.3 показано окно обзора класса AppWizard для приложения FourUp. Как и в случае с проектом NotePod, это окно с пестрым флажком позволяет выполнить обзор классов, используемых приложением, и отменить выборы, выполненные на предыдущих этапах. Это окно можно было бы сравнить с подобным окном, сгенерированным для NotePod. AppWizard выводил список из пяти классов для проекта NotePod, а для приложения FourUp выводятся только два класса. Это приложение действительно проще?

Окно обзора классов имеет еще одно отличие. В проекте NotePod можно было изменить базовый класс, используемый для получения класса представления приложения. Поскольку, как было упомянуто ранее, приложение, основанное на диалоговых окнах, не содержит никаких классов представлений, существует единственная настраиваемая опция, позволяющая изменить имя класса, который генерирует AppWizard. (Для класса **CFourUpDlg** можно также изменять имена файлов, используемых для хранения файлов реализации и заголовков. Можно изменять только имя класса **CFourUpApp**.)

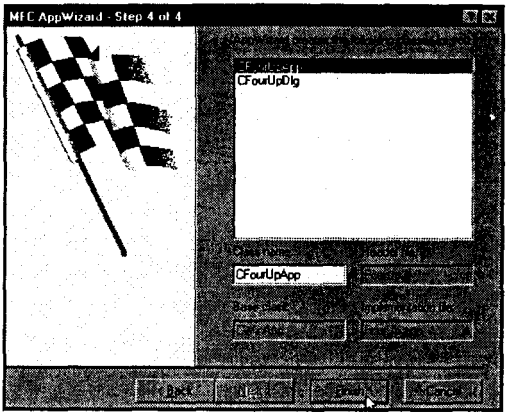


РИСУНОК 3.3
Файлы проекта MFC, основанного на использовании диалоговых окон.

Щелкните на кнопке Finish в окне обзора классов, а затем щелкните на кнопке OK в диалоговом окне New Project Information. AppWizard генерирует файлы, необходимые для вашей программы, и активизирует рабочую область окна проекта точно так же, как это делалось ранее. В рабочей области окна можно найти панели ClassView, ResourceView и FileView, как это было в случае с NotePod. Если щелкнуть на панели ClassView и развернуть все классы, можно увидеть, что приложение FourUp организовано более просто, чем NotePod — наличествуют только три класса по сравнению с шестью классами NotePod. (Ни NotePod, ни FourUp не перечисляют класс, используемый для создания диалогового окна приложения About в окне обзора классов.) Начальная конфигурация Visual C++ для проекта FourUp показана на рис. 3.4.

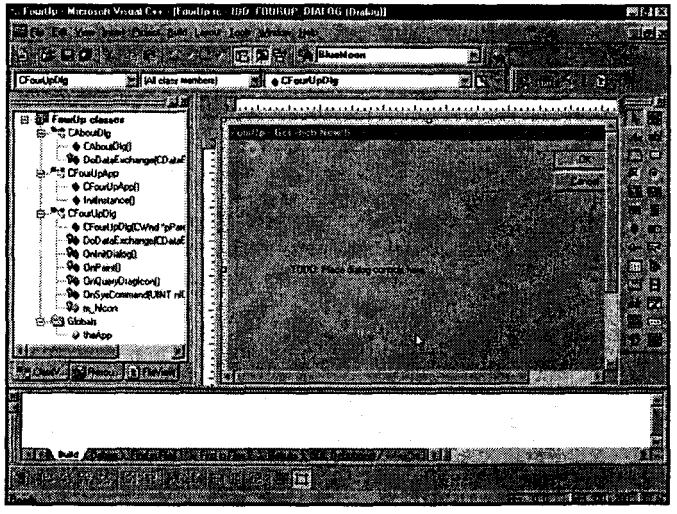


РИСУНОК 3.4
Visual C++ открывает конфигурацию приложения, основанного на диалоговых окнах.

При рассмотрении рис. 3.4 бросается в глаза одно обстоятельство: вместо пустой рабочей области справа от окна рабочей области проекта находится форма с кнопками OK и Cancel. Метка на форме гласит следующее:

TODO: Place dialog controls here (Что сделать: поместите здесь элементы управления диалоговых окон)

Рядом с формой находится панель инструментов, включающая кнопки, которые похожи на элементы управления в Visual Basic или Delphi.

Поскольку FourUp — это приложение, основанное на использовании диалоговых окон, то Visual C++ предоставляет возможность воспользоваться утилитой Dialog Editor (Редактор диалоговых окон). Эта утилита — один из нескольких специализированных редакторов, которые можно использовать для создания и поддержки ресурсов внутри программ Windows. Давайте перейдем к более близкому знакомству с этим приложением.

Начало работы с Dialog Editor

Перед началом работы над проектом игрового автомата FourUp давайте вкратце рассмотрим работу утилиты Dialog Editor. Затем будут использованы инструментальные средства для изменения диалогового окна приложения About прежде, чем перейти к более сложному главному экрану в главе 5.

Встреча с панелью управления

Панель, напичканная кнопками в Dialog Editor, называется *Control Toolbar*. Подобно большинству инструментальных панелей в Visual C++, эта панель является закрепляемой, т.е. когда она помещается у края экрана, то она выглядит как часть рамки.

Но посмотрите на верхнюю часть панели инструментов. Увидите ли вы маленькую кнопку Close и набор зубцов? Возьмите мышь, и "захватите" панель инструментов зубцами — ее можно перемещать и использовать в качестве перемещаемого набора инструментов. Если панель инструментов перемещается обратно к одной из сторон, то она закрепляется снова.

Каждая кнопка из Control Toolbar — это компонент (или *элемент управления*, согласно общей терминологии Visual C++), который может помещаться в диалоговом окне. Можно использовать два метода для размещения компонентов, подобных командным кнопкам или текстовым меткам, на поверхности диалогового окна:

- Щелкните на элементе управления, который будет использоваться. Кнопка, соответствующая элементу управления, останется в нажатом положении, а курсор мыши превратится в перекрестие при перемещении мыши поверх диалогового окна. Установите указатель мыши поверх диалогового окна и щелкните мышью для "помещения" элемента управления в требуемую позицию.
- Перетащите элемент управления с Control Toolbar. При перетаскивании элемента управления и перемещения курсора мыши поверх диалогового окна появится контур элемента управления. При отпускании кнопки мыши элемент управления размещается в текущем местонахождении.

СОВЕТ

Найдите свой инструмент

Панель Control Toolbar включает настолько много элементов управления, что может показаться затруднительным нахождение нужных элементов управления, особенно,

если пиктограммы, обозначающие элементы управления, являются незнакомыми. Однако Visual C++ может оказать вам помощь в подобной ситуации. При перемещении курсора мыши над Control Panel остановите курсор над интересующей кнопкой и Visual C++ отобразит окно подсказки, в котором будет содержаться наименование элемента управления.

Панель Control Toolbar включает 25 элементов управления. В этой главе будут рассмотрены только 4 из этих 25 элементов управления, остальные же элементы управления будут рассмотрены в других главах. На рис. 3.5 показана панель Control Toolbar и описывается каждый элемент управления.

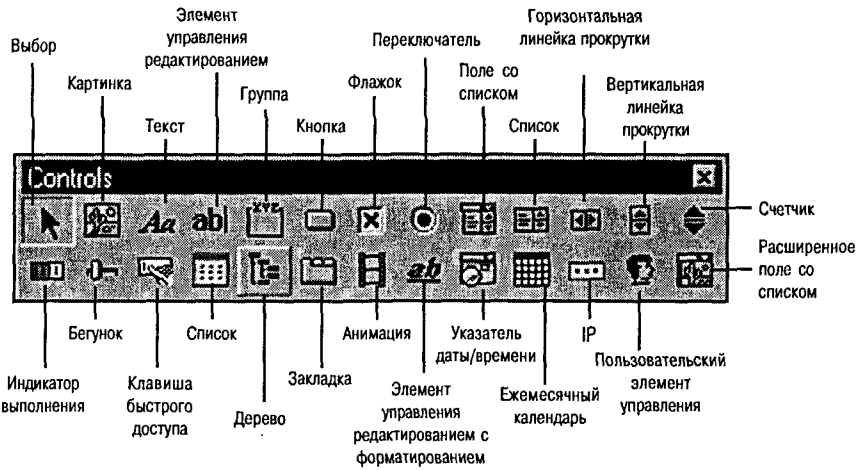


РИСУНОК 3.5 *Панель Control Toolbar из Developer Studio.*

Начнем с окна About

Хорошо, достаточно ходить вокруг да около. Давайте начнем с диалогового окна About. Если вы еще это не сделали, то убедитесь, что в рабочей области проекта выбрана закладка ResourceView. Раскройте папки таким образом, чтобы было видно то, что они содержат. Внутри папки Dialog можно найти две небольших пиктограммы, которые похожи на экранные формы. Каждая пиктограмма идентифицируется двумя необычными именами: **IDD_ABOUTBOX** и **IDD_FOURUP_DIALOG**.

Идентификаторы ресурса

Эти необычные имена — *идентификаторы ресурса (resource identifiers)*. Поскольку стиль написания идентификаторов ресурса предполагает использование заглавных букв, соединенных символами подчеркивания, они не являются именами переменных, но явно определенными константами препроцессора: константами, созданными при использовании директивы препроцессора C++ **#define**. Практически все ресурсы получают имена при помощи подобных идентификаторов.

Первая часть идентификатора ресурса, в данном случае **"IDD_"**, обозначает термин *ID*entifier for a *D*ialog (идентификатор для диалогового окна). Если ресурс

является пиктограммой, имя обычно начинается с **"IDI_"** и т.п. Как и большинство констант, определенных при помощи директивы **#define**, фактическое значение **IDD_ABOUTBOX** является целочисленной константой, хранимой в файле заголовка — в данном случае используется файл заголовка resource.h. Однако с точки зрения пользователя нет никаких различий в поддержке со стороны Visual C++ определения константы и содержимого файла resource.h, так что не следует задумываться об этом.

Дважды щелкните на **IDD_ABOUTBOX** и начните редактировать диалоговое окно About, разработанное при помощи AppWizard. Ваше окно должно походить на окно, изображенное на рис. 3.6.

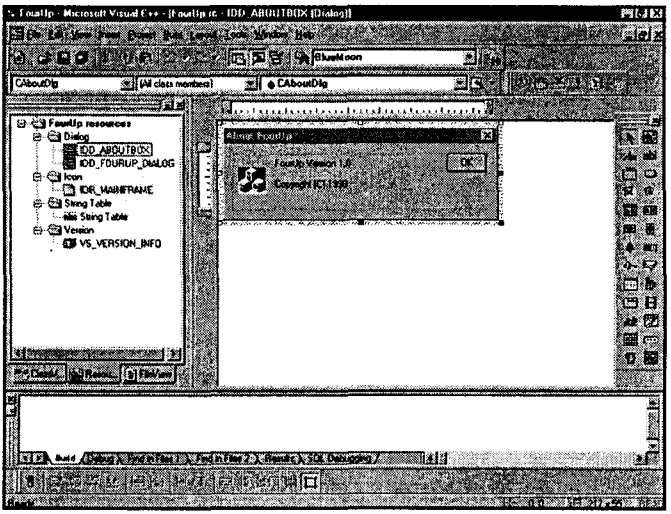


РИСУНОК 3.6
Открытие Dialog Editor для IDD_ABOUTBOX.

Идентификатор IDR_MAINFRAME

Вы, вероятно, изначально заметили пиктограмму MFC в диалоговом окне About. Все проекты MFC используют эту стандартную пиктограмму и как пиктограмму программы, и как пиктограмму, появляющуюся в заданном по умолчанию диалоговом окне About, но пользователь практически всегда заменяет эту пиктограмму на свою собственную.

Если снова обратить внимание на окно ProjectView, можно отметить, что пиктограмма именуется при помощи идентификатора **IDR_MAINFRAME**. Учитывая предыдущее обсуждение идентификаторов, можно было бы ожидать, что пиктограмма получит имя **IDI_MAINFRAME**. Однако рассматриваемый случай является исключением для стандарта обозначений имен: **IDR** означает *Id*entifier for a *R*esource, поскольку один и тот же идентификатор используется для пиктограмм, курсоров, таблиц акселераторов и ряда других ресурсов. Как вы думаете, почему MFC имеет несколько ресурсов с тем же именем? Если рассматривать оставшуюся часть имени, то это могло бы иметь смысл. Любой ресурс, который использует идентификатор **IDR_MAINFRAME** — это заданный по умолчанию ресурс для главного окна вашего приложения. Пиктограммы, курсоры и другие ресурсы могут совместно использовать это имя.

Добавление новых ресурсов

Для замены встроенной по умолчанию пиктограммы MFC необходимо создать свою собственную пиктограмму. При установке Visual C++ в вашем распоряжении имеется возможность установки набора ресурсов для использования в собственных программах. Эти ресурсы устанавливаются в подкаталог Common\Graphics каталога Visual Studio. (Если установка ресурсов не осуществлялась, этот каталог можно найти на CD-ROM и просто скопировать необходимые файлы на жесткий диск.) В этом каталоге также находятся другие подкаталоги для пиктограмм, растровых изображений и других графических ресурсов. Для создания новой пиктограммы (и генерации изображений, выводимых платой видеоадаптера), используются четыре изображения, находящиеся в подкаталоге Icon\Misc: Misc34.ico, Misc35.ico, Misc36.ico и Misc37.ico. Эти файлы включают изображения, соответственно, мастей черви, трефы, бубны и пики. Конечно, можно использовать любые изображения, которые вам по душе, если они сохранены в виде файлов ICO.

Для добавления этих пиктограмм в проект выберите пункт меню Insert | Resource, а затем выберите Icon из списка ресурсов, отображаемых в диалоговом окне Insert Resource. Затем щелкните на кнопке Import и используйте стандартное диалоговое окно открытия файлов для нахождения требуемых пиктограмм. Можно выбрать несколько файлов, удерживая нажатой клавишу Ctrl. Диалоговое окно Insert Resource показано на рис. 3.7.

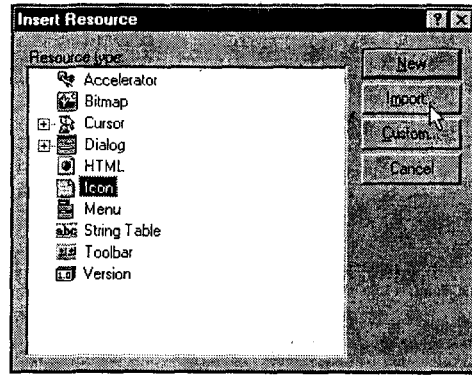


РИСУНОК 3.7
Диалоговое окно Insert Resource.

Встреча с Bitmap Editor

Как только загружается пиктограмма, можно заметить еще одно изменение во внешнем виде Visual C++. Dialog Editor и Control Toolbar просто исчезают, а на их месте появляется палитра красок и комплект инструментов, включающий инструменты рисования.

Перед вами Bitmap Editor (Редактор растровых изображений) из Visual C++. Утилита Bitmap Editor используется для создания или управления курсорами, пиктограммами и (как можно ожидать) растровыми изображениями. Вместо поддержки отдельных инструментов для каждого типа ресурса поведение Bitmap Editor является более тонким, обеспечивая переключение типа ресурса при переходе от рисунка к рисунку.

В дополнение к палитрам Drawing и Color, внимание на себя обращает то обстоятельство, что область рисования — место, где размещается рисунок — раз-

бита на две панели. В одной панели находится пиктограмма, увеличенная приблизительно в шесть раз по сравнению с нормальной величиной, в то время как другая панель показывает ресурс в нормальную величину. Можно выполнять операции по окрашиванию в любой панели, но большую часть времени вы будете рисовать на большей панели и наблюдать результаты своей работы в меньшей панели.

Начать работу с Bitmap Editor можно, выполнив двойной щелчок на любом ресурсе растрового типа (пиктограмма, курсор и пр.) в окне ResourceView. Выберите при необходимости окно ResourceView и щелкните дважды на пиктограмме с именем **IDR_MAINFRAME**. Появится экран, аналогичный показанному на рис. 3.8.

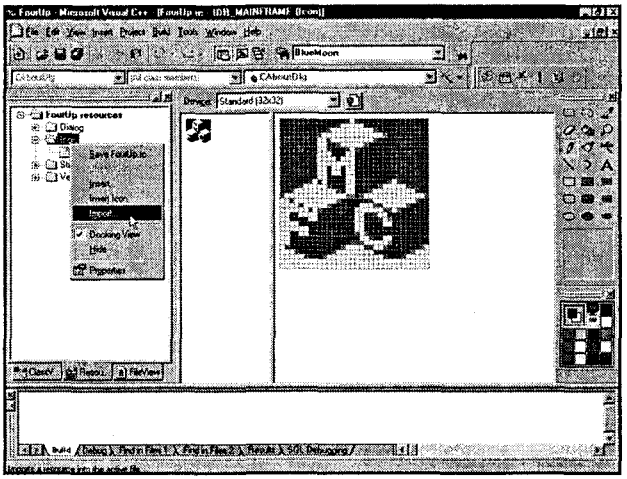


РИСУНОК 3.8
Утилита Visual C++ Bitmap Editor.

Добавление новых пиктограмм

Как показано на рис. 3.8, добавлять новый ресурс в проект можно также при помощи щелчка правой кнопкой мыши на папке Icon в окне ResourceView. Затем можно выбрать команду Import из появившегося контекстного меню и сохранить результаты своей работы за счет выбора типа ресурса из меню Insert Resource.

При добавлении ресурсов к проекту, каждому из них присваивается имя по умолчанию. Первая пиктограмма, добавляемая в проект, получает идентификатор **IDI_ICON1**. По общему признанию, это имя не является мнемоническим. Для некоторых ресурсов имя не имеет никакого значения — оно никогда не будет использоваться. Однако иногда нужно будет работать с ресурсами и управлять ими во время выполнения программы. Это утверждение может оказаться справедливым для каждой из четырех новых пиктограмм, в которых нуждается проект FourUp, поэтому мнемоническое имя значительно облегчает работу с ресурсами.

Изменение идентификатора ресурса

Изначально может быть неочевидно, каким образом следует изменять имя пиктограммы. Если дважды щелкнуть на имени пиктограммы в окне ResourceView, запускается Bitmap Editor. Самый простой метод, как обычно в Windows, состоит в использовании щелчка правой кнопкой мыши:

- В окне ResourceView выберите имя пиктограммы, подлежащей переименованию.
- Щелкните правой кнопкой мыши для открытия контекстного меню для пиктограммы.
- Выберите команду Properties из списка.
- В появившемся диалоговом окне Properties введите новое имя в раскрывающемся списке ID.

На рис. 3.9 показана первая из пиктограмм, **IDI_ICON1**, переименованная в **IDI_HEART**. После этого загрузите других три пиктограммы (если вы не загрузили их ранее, используя возможность выделения нескольких пиктограмм) и переименуйте их в **IDI_CLUB**, **IDI_DIAMOND** и **IDI_SPADE**.

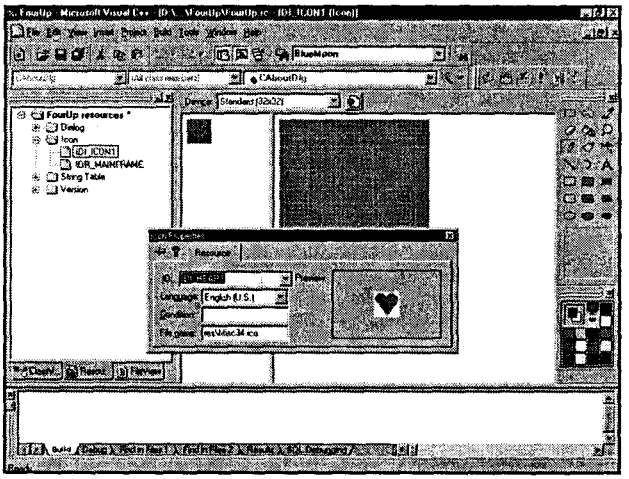


РИСУНОК 3.9
Выбор идентификатора ресурса.

Изменение пиктограммы приложения

Теперь, когда были найдены и поименованы четыре пиктограммы, обратим наши взоры на изменение пиктограммы, представляющей вашу программу. Эта пиктограмма отображается в нескольких местах:

- Возле имени программы в Windows Explorer.
- Возле имени программы в Windows 95 Taskbar при выполнении программы.
- В названии окна приложения.
- На рабочем столе, если был создан ярлык для приложения.
- В диалоговом окне About.

Пиктограммы бывают разных размеров. Обычный размер — 32 пиксела в ширину и 32 пиксела в высоту. Такого рода пиктограмма появляется на рабочем столе, либо же когда пользователь выбирает пункт Large Icons (Большие пиктограммы) из меню View приложения Windows Explorer. Каждая сторона меньшей пиктограммы (половина ранее указанной величины, т.е. 16 пикселов) появляются, когда пользователь выбирает пункт Small Icons или List в меню View Windows Explorer, а также в названии окна приложения и в Windows 95 Taskbar.

Файл изображения пиктограммы может содержать оба типа изображений, хотя в окне ResourceView будет виден только один идентификатор. Для переключения между размерами изображения пиктограммы потребуется выбрать пункт меню из раскрывающегося списка, который появляется над изображением при редактировании пиктограммы. При выполнении такого переключения AppWizard генерирует пиктограммы размером 16 × 16 и 32 × 32 пиксела.

Поскольку нет необходимости создавать пользовательскую пиктограмму для обоих размеров, нужно удалить маленькую пиктограмму из файла. После этого, когда Windows выполняет программу FourUp и не находит маленькую пиктограмму, она просто соответствующим образом изменяет размеры большой пиктограммы. Сначала убедимся в том, что загружена пиктограмма **IDR_MAINFRAME**. Затем можно удалить маленькую пиктограмму, выбрав пункт Small (16 × 16) из раскрывающегося списка, а затем выбрав команду Delete Device Image (Удалить изображение устройства) из меню Image. (Убедитесь, что не выбран пункт Standard (32 × 32), иначе придется создавать пиктограмму заново.)

Пользовательская пиктограмма FourUp может быть создана за счет выполнения следующих трех этапов:

1. Возьмите существующую пиктограмму MFC и воспользуйтесь некоторыми инструментами Bitmap Editor для создания заказного фона.
2. Объедините фон с импортированной ранее пиктограммой масти черви для создания новой составной пиктограммы.
3. Добавьте некоторые специальные эффекты, тем самым придав пиктограмме FourUp запоминающийся вид.

Готовы? Пошли дальше.

Этап 1: Создание заказного фона

Следуйте этим инструкциям для изменения пиктограммы MFC таким образом, что она будет выступать в роли заказного фона для пиктограммы FourUp:

1. Дважды щелкните на пункте **IDR_MAINFRAME** в папке Icons окна ProjectView. При этом откроется окно Bitmap Editor Visual C++ (см. рис. 3.8).
2. Выберите инструмент Rectangle Selection (Прямоугольное выделение) из палитры Drawing. Этот инструмент похож на прямоугольник, нарисованный пунктирной линией, и появляется по умолчанию в левом верхнем углу палитры. (В случае необходимости используйте окно подсказки для идентификации инструмента.)
3. Перемещайте курсор мыши над изображением пиктограммы; при этом курсор приобретает вид большого перекрестия. Поместите курсор в левом верхнем углу пиктограммы **IDR_MAINFRAME**, используемой по умолчанию, щелкните левой кнопкой мыши. Удерживая левую кнопку мыши, выполняйте буксировку вниз и вправо, пока не будет максимально выделен темно-синий фон, без выделения любого из блоков MFC. Выделенная область должна расшириться примерно на одну четверть в левом направлении и почти на половину размера пиктограммы в нижнем направлении. Отпустите кнопку мыши.
4. Скопируйте выбранный блок в буфер обмена, используя либо команду Edit | Copy, либо комбинацию клавиш Ctrl + C.

5. Вставьте блок обратно в ваше изображение, используя либо комбинацию Edit | Paste, либо комбинацию клавиш Ctrl + V. После выполнения этой операции изображение появится в позиции, в которой оно было ранее. Обратите внимание на прямоугольник выделения, окружающий блок — перемещайте ваш курсор внутри блока, при этом ориентиром служит выделяющий контур. Переместите вставляемый блок вправо, покрывая блок в максимально возможной степени, но сохраняя при этом выделение в верхней части пиктограммы.
6. Продолжайте вставлять и перемещать ту же самую выделенную область, пока не покроется верхняя половина пиктограммы. На этом этапе повторите тот же самый процесс для нижней половины пиктограммы. Не волнуйтесь относительно заполнения средней части — это можно выполнить весьма быстро. Убедитесь, что заполнена синяя область с края нижней части пиктограммы.
7. Используя инструмент выделения, выберите нижнюю половину пиктограммы. При этом проявляйте осторожность, выбирая только пикселы, окрашенные в синий и черный цвета, которые были вставлены из буфера обмена — не выбирайте пикселы, окрашенные в белый цвет и находящиеся в середине пиктограммы. Ваше выделение должно распространяться поперек ширины пиктограммы.
8. Выберите команду Image | Flip Vertical (Изображение | Зеркальное отражение по вертикали) из главного меню. Теперь появляются два зеркальных фона изображения, цвет которых постепенно изменяется от черного на верхней и нижней границах к синему посредине. Чтобы закрыть остающийся "мусор" в середине пиктограммы, выберите несколько синих пикселов, затем скопируйте и вставьте их в середину, пока ваше изображение не станет походить на изображение, показанное на рис. 3.10. Сейчас самое время для сохранения рабочей области окна проекта с тем, чтобы непредвиденный аварийный отказ системы оказался менее болезненным.

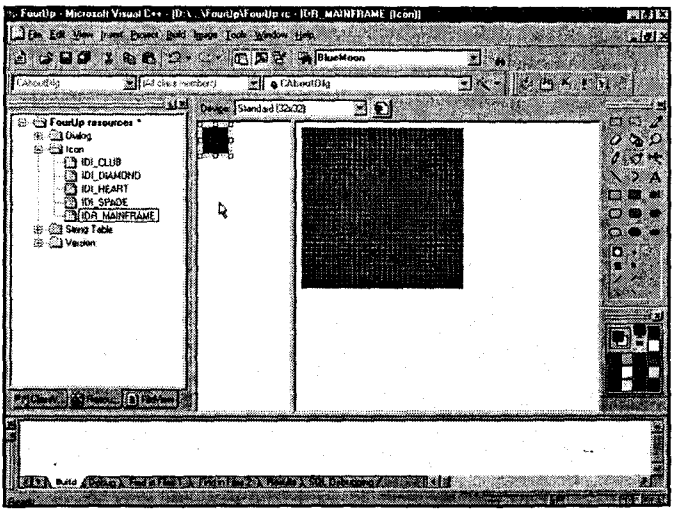


РИСУНОК 3.10
Изменение пиктограммы IDR_MAINFRAME в Bitmap Editor.

Этап 2: Объединение двух изображений пиктограммы

Теперь можно взять пиктограмму **IDI_HEART**, добавленную к вашему проекту ранее, и объединить ее с только что созданным новым фоном. Выполните это, поступая в соответствии со следующими советами:

1. Щелкните дважды на имени пиктограммы **IDI_HEART** в окне ResourceView. (Если вы уже забыли ранее изложенный материал, то это имя первоначально выглядит как **IDI_ICON1**.)
2. Убедитесь в том, что изображение выделено в Bitmap Editor. (Должно быть выделено либо большое, либо маленькое изображение — это не имеет никакого значения.)
3. Скопируйте изображение в буфер обмена, используя команду Edit | Copy или комбинацию клавиш Ctrl + C.
4. Выберите пиктограмму **IDR_MAINFRAME**, дважды щелкнув на ее имени в окне ResourceView.
5. Выберите изменяемое изображение (и снова не имеет значения, выбирается большое или маленькое изображение) и вставьте изображение **IDI_HEART** из буфера обмена.

Непрозрачный и прозрачный фоны

Мы выполнили практически все, что нужно на этом этапе, но существует одна потенциальная сложность. Для пиктограмм и курсоров (но не для обычных растровых изображений) Bitmap Editor может обрабатывать фон как непрозрачную либо прозрачную область. Для новой заказной пиктограммы нам необходим синий градиент фона с тем, чтобы высветить изображение масти черви, так что следует убедиться в том, что фон изображения является прозрачным.

Если выбирается изображение при помощи инструмента Rectangle Selection (Прямоугольное выделение), то палитра Drawing немного изменяется в зависимости от используемых инструментов, инструментов, вычерчивающих линии, или инструментов, позволяющих выполнить рисование вручную. Если выбирается инструмент Pencil (Карандаш) или инструмент Brush (Кисть), то область, ограниченная палитрами Drawing и Color изменяется, что позволяет выбирать тип карандаша или ширину красящей кисти. Однако когда используется инструмент Rectangle Selection, то в этой области появляется *индикатор прозрачного выделения* (transparency selection indicator). Если верхняя часть изображения, находящегося в этой области, подсвечена, то вставляемое изображение использует непрозрачный фон. Если изображение, находящееся в нижней части области индикатора подсвечивается, то вставляемое изображение использует прозрачный фон. Результат перетаскивания пиктограммы **IDI_HEART** поверх пиктограммы, заполненной градиентом синего цвета, с прозрачным фоном можно видеть на рис. 3.11.

Остальные инструменты из палитры Drawing работают так же, как и аналогичные им инструменты в программе рисования, подобной Windows Paint. Могут возникнуть трудности с небольшими пиктограммами — необходимо обладать особыми талантами, чтобы реализовать хороший вид маленького изображения. Потратьте некоторое время на экспериментирование с каждым из инструментов, что поможет определить, владеете ли вы подобным талантом.

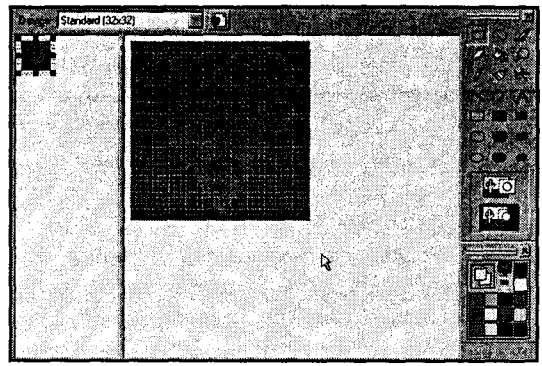


РИСУНОК 3.11
Объединение двух пиктограмм.

Этап 3: Использование инструмента Text

Если вы находите ваш рисунок недостаточно совершенным, его внешний вид можно улучшить при помощи инструмента Text. При использовании этого инструмента на экране появляется маленькое плавающее диалоговое окно, расположенное поверх главного окна Bitmap Editor. Диалоговое окно включает кнопку Font, которая позволяет выбирать любой тип шрифта, установленного в данной системе. После выбора шрифта можно вводить любые желаемые символы. По мере ввода символов они появляются на пиктограмме, окруженные прямоугольником выделения. Текст можно располагать в любом требуемом месте, позиция текста не фиксируется до тех пор, пока не закрывается диалоговое окно инструмента Text.

Давайте добавим небольшой текст в пиктограмму прежде, чем перейдем к другому материалу. Вот как это делается:

1. Выберите инструмент Text из палитры Drawing.
2. Используя палитру Color, установите черный цвет символа, выполнив щелчок левой кнопкой мыши на образце черного цвета. (Если щелкнуть правой кнопкой мыши, то изменится цвет фона.)
3. Используя кнопку Font в диалоговом окне инструмента Text, выберите на свое усмотрение любой шрифт величиной 20 пунктов. Затем введите и позиционируйте символ "4" в центре изображения масти черви.
4. Если черная цифра "4" расположена в желаемом месте, щелкните на кнопке инструмента Text снова для фиксации символа.
5. Теперь измените цвет фона на желтый при помощи палитры Color.
6. Напечатайте символ "4" тем же шрифтом снова, на этот раз левее черной цифры "4". Первое изображение будет выглядеть как падающая тень, расположенная позади желтой цифры "4". Если цифра разместилась в желаемом месте, закройте диалоговое окно инструмента Text за счет выбора другого средства управления (для этого пригоден любой инструмент). Работа завершена. На рис. 3.12 показана размещенная по-новому желтая цифра "4".

И снова Dialog Editor

Примите поздравления! Ваше приложение снабжено новой пиктограммой. Теперь давайте завершим оформление диалогового окна About с тем, чтобы можно было перейти к работе над остальной частью программы. Начнем с добавления новой пиктограммы к диалоговому окну About. Для этого потребуется сделать следующее:

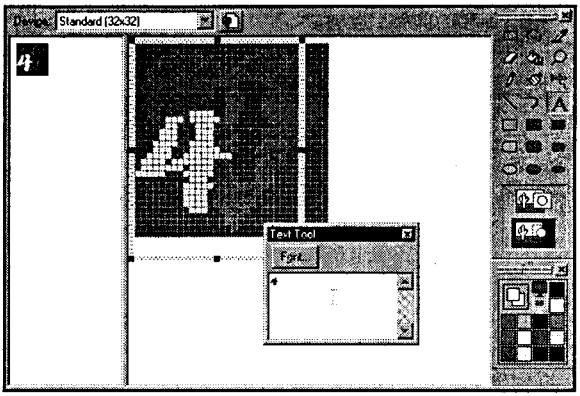


РИСУНОК 3.12
Добавление текста к пиктограмме.

1. В окне ResourceView дважды щелкните на пункте **IDD_ABOUTBOX** для открытия окна Dialog Editor.
2. Выберите пиктограмму, отображаемую в диалоговом окне About. Затем щелкните правой кнопкой мыши для открытия контекстного меню. Выберите пункт Properties для открытия диалогового окна Picture Properties (см. рис. 3.13).

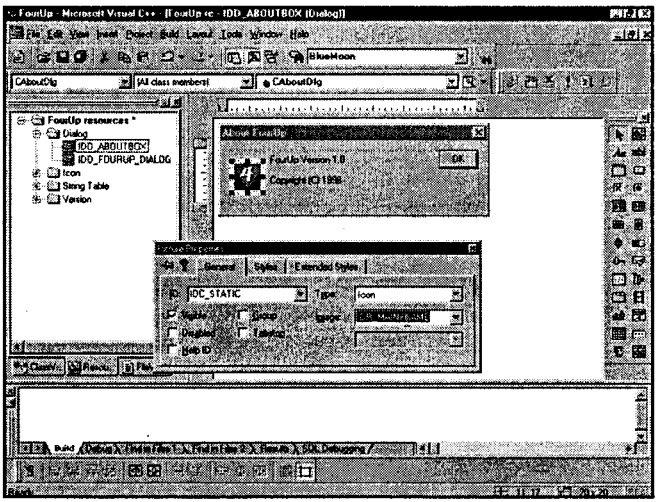


РИСУНОК 3.13
Изменение пиктограммы в диалоговом окне About.

3. В диалоговом окне Picture Properties убедитесь, что раскрывающийся список Type отображает значение Icon, затем из списка Image выберите любое значение, *за исключением* **IDR_MAINFRAME**. Обратите внимание, что пиктограмма, отображаемая в диалоговом окне About, изменяется. (Вам, вероятно, придется вывести курсор из поля прежде, чем изменение воздействует.) Теперь выберите повторно пункт **IDR_MAINFRAME** из раскрывающегося списка Image. Закройте диалоговое окно и посмотрите на новую пиктограмму, находящуюся в диалоговом окне About.

Выбор, изменение размеров и перемещение компонентов

Теперь, когда вы умеете работать с растровыми изображениями, наступило время узнать, как работать с элементами управления. Во всей этой главе даются указания относительно выбора пунктов меню, однако еще не сообщалось подробно о том, что появляется на экране при выборе указанного пункта меню.

При выборе элемента управления, растрового изображения или диалогового окна в любом из редакторов ресурсов следует обратить внимание на два обстоятельства. Во-первых, выбранный пункт меню окружен довольно толстым "размытым" белым *прямоугольником выделения*. Во-вторых, маленький прямоугольник появляется в каждом углу выделяющего контура, равно как и в средних точках верхней части и сторон, — эти прямоугольники являются *органами изменения размеров*.

Органы изменения размеров, расположенные в средних точках, позволяют изменять размеры прямоугольника выделения в указанном направлении — горизонтальном или вертикальном. Органы установки размеров, расположенные по углам, управляют размером формы по диагонали, увеличивая или уменьшая ее, а также изменяют размеры в горизонтальном и вертикальном направлениях.

Если курсор размещается поверх одного из органов изменения размеров, то вид курсора изменится, сообщая направление, в котором можно перемещать данную линию. Если курсор перемещается поверх органа изменения размера по вертикали, появится стрелка компаса, показывающая направление "север-юг". (Курсор примет вид двунаправленной стрелки, указывающей в верхнем и нижнем направлениях.) Если курсор перемещается поверх одного из органов изменения размеров по горизонтали, он примет вид стрелки компаса, показывающей направление "восток-запад"; в зависимости от выбираемого угла, курсор может также принимать вид стрелки, указывающей направление "северо-запад — юго-восток" или "северо-восток — юго-запад".

Помимо направления, органы изменения размеров также имеют *статус активности*. Активный орган появляется в виде заполненного прямоугольника, а неактивный — в виде пустого прямоугольника. Неактивный прямоугольник изменения размеров в действительности не делает чего-либо за исключением того, что сообщает о своем местонахождении.

Давайте посмотрим, как эти органы работают в диалоговом окне About. Обратите внимание на рис. 3.14 и отметьте для себя следующие факты:

- Прямоугольник выделения окружает диалоговое окно About.
- Активные органы изменения размеров находятся снизу и справа, а также в нижнем правом углу.
- Пять неактивных органов изменения размеров находятся на всех остальных сторонах и углах.
- Курсор, имеющий вид стрелки компаса "север-юг", используется для увеличения диалогового окна.

Измените размеры диалогового окна About так, чтобы увеличить его высоту, обеспечив пространство для размещения новых элементов управления. Не беспокойтесь относительно точной установки размера: эти размеры легко изменяются.

Для перемещения элементов управления без изменения размеров необходимо сначала удостовериться в том, что эти элементы выбраны. Затем переместите ваш

курсор таким образом, чтобы установить его на прямоугольник выделения, а не на один из органов изменения размеров. Можно также перемещать некоторые элементы управления, размещая курсор внутри прямоугольника выделения, т.е. непосредственно на элементе управления.

СОВЕТ

Секрет правильной установки курсора

Не задавали ли вы себе вопрос о том, как узнать о месте нахождения курсора? Очень просто. Достаточно взглянуть на вид курсора мыши. Если элемент управления готов для перемещения, курсор приобретает вид стрелки компаса (четыре стрелки, указывающие направление "север-восток-запад-юг"). Если курсор примет вид стрелки компаса, то можно выполнять буксировку элемента управления в желаемом направлении.

Введение в Dialog Toolbar

Довольно сложно точно установить и изменить размеры элементов управления. Но как и можно было ожидать, Visual C++ предлагает несколько инструментов, компенсирующих допускаемые погрешности. Возможно, самый удобный из них — это Dialog Toolbar.

Dialog Toolbar обычно закрепляется в нижней части экрана. Большинство элементов управления предназначены для оказания помощи в выравнивании элементов управления в индивидуальном порядке и совместно. Dialog Toolbar используется в оставшейся части этой главы, что придает диалоговым окнам аккуратный и профессиональный вид.

Начнем с перемещения кнопки OK из позиции в левом верхнем углу в нижнюю часть диалогового окна. Это перемещение осуществляется достаточно просто, поскольку рамка диалогового окна четко выделяется. Проблема проявляется, когда нужно центрировать кнопку, поскольку если наблюдается смещение даже на один пиксел в одном из направлений, экраны будут выглядеть незавершенными. Решение заключается в использовании инструмента Center Horizontal, находящегося в Dialog Toolbar (см. рис. 3.15). При этом нет нужды заниматься скучными подсчетами пикселов вручную.

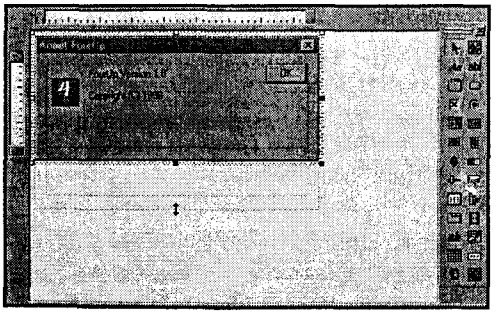


РИСУНОК 3.14 *Изменение размера диалогового окна About в Dialog Editor.*

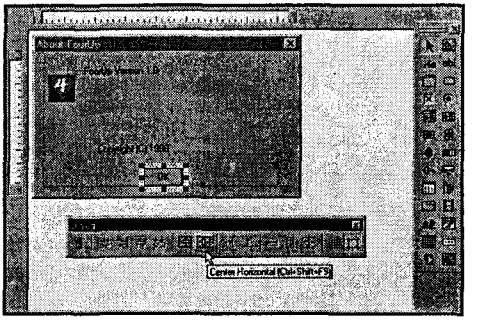


РИСУНОК 3.15 *Буксировка и выравнивание элементов управления в Dialog Editor.*

Работа со статическим текстом

Компонент *статического текста* используется в диалоговых окнах достаточно часто. Статический текст, как и следует из самого наименования, — это просто метка, которая содержит неизменное текстовое значение. В диалоговом окне About образцами статического текста могут служить строки, содержащие номер версии FourUp и информацию об авторском праве.

В Control Toolbar пиктограмма статического текста состоит из букв *Aa*, написанных курсивом. (Две других пиктограммы также выглядят похоже: элемент управление текстовым полем, также известный как элемент управления редактированием, состоит из символов *ab*, за которыми находится вертикальная черта, а элемент управления форматированным текстом состоит из символов *ab*, написанных подчеркнутым курсивом.) Для добавления нового элемента управления статическим текстом просто выберите пиктограмму статического текста в Control Toolbar, а затем перетащите его на ваше диалоговое окно.

Подобно большинству других элементов управления, используемых в Dialog Editor, можно изменять свойства статического текста за счет выбора элемента управления и последующего щелчка правой кнопкой мыши и выбора пункта Properties из контекстного меню. Диалоговое окно Text Properties состоит из трех страниц. Первая страница, General, позволяет изменять идентификатор и заголовок, отображаемый на экране. Давайте попробуем ею воспользоваться и посмотрим, как она работает.

1. Выберите текст, содержащий информацию об авторском праве, и переместите его в нижнюю часть экрана. Поместите его выше кнопки OK. (Не волнуйтесь относительно точного выравнивания на этом этапе.)
2. Откройте диалоговое окно Text Properties, выберите команду Properties из контекстного меню или команду View | Properties.
3. Обратите внимание на то, что идентификатор элемента управления **IDC_STATIC.IDC_STATIC** определен как -1, что является стенографическим способом записи высказывания: "Я хочу, чтобы этот элемент управления находился здесь и хорошо выглядел. Я никогда не собираюсь связываться с ним снова". Это связано с тем, что вы не имеете ни малейшего намерения изменять информацию об авторском праве во время выполнения; его значение остается неизменным.
4. Если изменяется заголовок, можно использовать любые символы. Для создания нескольких строк текста можно вставлять в текст управляющую последовательность C++ **\n**. Можно также использовать специальные символы ANSI, если вам это нравится, применяя соответствующую восьмеричную управляющую последовательность — вводится символ ESC, сопровождаемый восьмеричным значением с тремя значащими цифрами для символа, который требуется отобразить. Список полезных управляющих последовательностей можно найти в оперативной справке Visual C++.

На рис. 3.16 показаны изменения строк, содержащих описание авторского права. Обратите внимание, что поскольку производится ввод текста в диалоговом окне Text Properties, изменения отражаются в фактическом тексте, отображаемом в диалоговом окне. На рисунке используется восьмеричная управляющая последовательность **\251** для того, чтобы строка начиналась с символа авторского права, т.е ©.

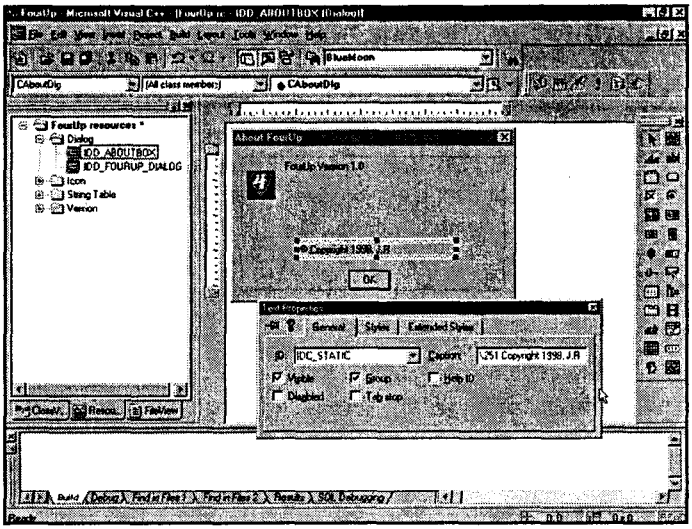


РИСУНОК 3.16
Статический текст. Изменение заголовка.

Стили статического текста

В диалоговом окне Text Properties страница свойств Styles позволяет выбирать опции выравнивания текста. Наиболее часто тип выравнивания текста выбирается с использованием раскрывающегося комбинированного списка Align Text, особенно если необходимо центрировать текст.

Как можно видеть на рис. 3.17, эта страница также позволяет выбирать ряд других опций. Ниже рассматриваются их значения:

- *Center Vertically* (Вертикальное выравнивание) указывает Windows об автоматическом центрировании текста внутри области, определенной прямоугольником выделения.

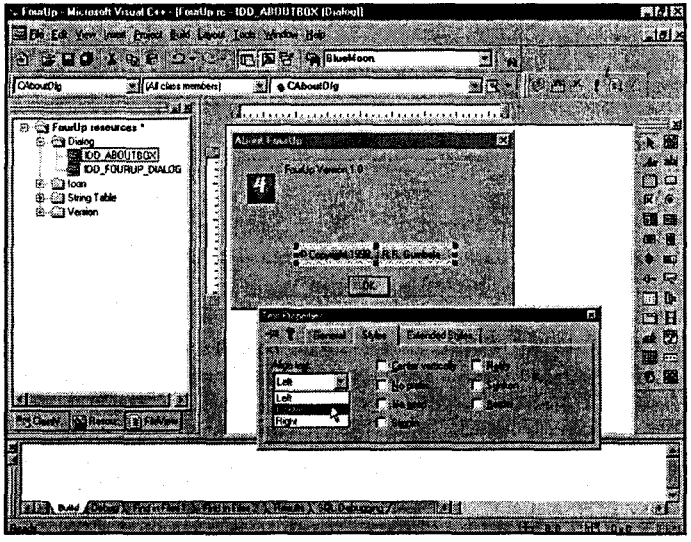


РИСУНОК 3.17
Изменение стиля выравнивания статического текста.

- *No Prefix* (Отсутствует префикс) позволяет вводить текст, содержащий символы амперсанда (&). Обычно если символ амперсанда помещается в текст, Windows интерпретирует его как указание подчеркивать следующий символ и сделать этот символ клавишей быстрого доступа. Эта опция иногда полезна в случае статического текста, как это будет видно далее в книге, но она чаще используется вместе кнопками и другими элементами управления, которые могут выполнять некоторое действие.
- *No Wrap* (Нет перехода на новую строку) предотвращает автоматический переход на новую строку. Обычно если текст в области заголовка слишком большой для размещения на одной строке, Windows разбивает текст между строками. Однако обратите внимание на то, что Windows не изменяет размеры прямоугольника выделения с целью размещения дополнительных строк, — это необходимо делать самостоятельно. Можно увидеть работу этой опции, взяв любой фрагмент статического текста и изменив размеры прямоугольника выделения для того, чтобы заставить Windows переносить строки по словам. Если установить флажок No Wrap, то это поведение отменяется.
- *Simple* (Простой) создает элемент управления текстом, который отображается Windows быстрее, чем другие типы элементов управления текстом. Однако простой элемент управления текстом не может выполнять некоторые из приемов для знатоков, которые доступны его менее простым собратьям.
- *Notify* (Уведомить) создает элемент управления текстом, которое может посылать сообщение родителю, когда на нем выполняется одиночный или двойной щелчок. Обычно элементы управления статическим текстом никаких сообщений не генерируют.
- *Sunken* (Снизить) добавляет трехмерную вогнутую рамку, которая следует за прямоугольником выделения статического текста.
- *Border* (Рамка) функционирует подобно Sunken, но при этом создается рамка из единственной линии, ограничивающая область выделения.

Для строки авторского права в FourUp установим выравнивание по центру и оставим все другие значения по умолчанию. Результат показан на рис. 3.17.

Теперь аналогичным образом используйте свойство Align Text для выравнивания по центру фразы "FourUp Version 1.0" внутри элемента управления статическим текстом, который ее содержит.

Выравнивание и выбор нескольких элементов управления

До сих пор работа происходила только с отдельными элементами управления. Однако часто необходимо будет работать с несколькими элементами управления как с группой. К счастью, Dialog Toolbar содержит набор инструментов, который позволит легко выполнять выравнивание и работу с несколькими компонентами.

Обычно когда выбирается компонент, подобный элементу управления статическим текстом, отменяется выбор любого ранее выбранного элемента управления. Для работы с несколькими элементами управления необходим некий способ выбора нескольких элементов управления. Можно использовать любой из следующих двух методов:

- Удерживайте клавишу Shift или клавишу Ctrl во время выбора второго и последующих элементов управления. Обратите внимание на то, что когда

выбирается несколько элементов управления, органы изменения размеров элемента управления, выбранного в последний раз, будут активными, в то время как органы изменения размеров всех других элементов управления станут неактивными. Элемент управления с активными органами изменения размеров называется *доминирующим элементом управлением* (*dominant control*).

- Перетащите курсор мыши таким образом, чтобы создавался прямоугольник выделения вокруг любых элементов управления, которые требуется выбрать. (Если прямоугольник выделения только частично окружает элемент управления, последний выбран не будет.) Когда отпускается кнопка мыши, элемент управления, наиболее близкий к начальной точке прямоугольника выделения, становится доминирующим элементом управлением.

Если только было выбрано несколько элементов, их можно перемещать вокруг вашего диалогового окна в качестве группы, используя либо мышь, либо клавиши управления курсором. Можно также копировать, вырезать и вставлять группу элементов, используя буфер обмена. И что более интересно, можно использовать Dialog Toolbar для выравнивания элементов управления относительно друг друга. Давайте посмотрим, как это процесс происходит с нашим диалоговым окном About. Обратите внимание на рис. 3.18.

1. Выберите текст, содержащий заголовок приложения: FourUp Version 1.0.
2. Перетащите элемент управления так, чтобы он установился в позиции, показанной на рис. 3.18.

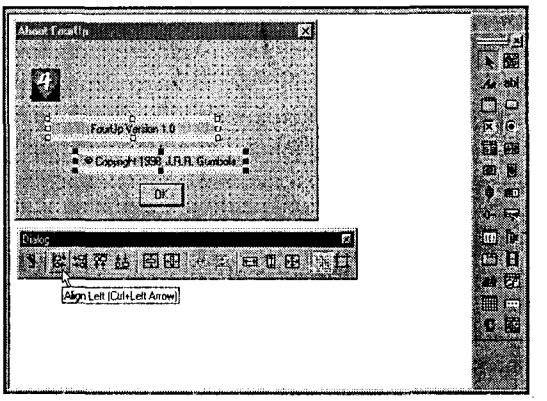


РИСУНОК 3.18
Выравнивание элементов управления при помощи Dialog Toolbar.

3. Удерживая клавишу Shift или Ctrl, выберите элемент управления статическим текстом, включающий знак авторского права, таким образом, чтобы он стал доминирующим элементом управления. Убедитесь в том, что элемент управления текстом заголовка выбран.
4. В Dialog Toolbar щелкните на кнопке Align Left. Можно добиться того же результата, выбрав команду Layout | Align | Left из главного меню, или используя клавишу быстрого доступа Ctrl + стрелка влево.

СОВЕТ

Избегайте неправильного выравнивания

Не угодите в одну возможную ловушку при использовании элементов управления выравниванием — необходимо убедиться, что кнопка, на которой производится

щелчок, соответствует требуемому типу выравнивания. Например, если щелкнуть на кнопке Align Top, используя выделенную метку на рис. 3.18, то вы будете удивлены, когда метка текста заголовка закроет метку авторского права. Конечно, можно использовать команду Edit | Undo во избежание последствий вашего "преступления". Вообще говоря, избегайте использовать команды Align Top или Align Bottom с элементами управления, которые размещаются вертикально, или команды Align Left или Align Right с элементами управления, размещенными горизонтально.

Когда это выполнено, упорядочивайте текст заголовка самостоятельно таким образом, чтобы левый край выравнивался в соответствии с левым краем текста авторского права. Текст заголовка перемещается скорее, чем текст авторского права, поскольку текст авторского права является доминирующим элементом управления.

Как можно видеть, другие элементы управления из Dialog Toolbar работают аналогично: Align Right, Align Top и Align Bottom. (В главе 5 будет рассмотрен набор связанных кнопок, которые образуют постоянную группу выделенных средств управления.)

Dialog Toolbar предлагает, наконец, опцию выравнивания, которая, наверняка, окажется удобной для пользователя: кнопка Grid (возможно, вы уже заметили ее на рис. 3.18). Кнопка Grid позволяет создавать сетку на форме. При перемещении элементов управления и совмещении их с активной сеткой они должны разместиться вдоль линий сетки. Кроме того, можно изменять интервал между линиями сетки при помощи диалогового окна Guide Settings, получить доступ к которому осуществляется через меню Layout.

О свойствах шрифтов

Как показано на рис. 3.18, диалоговое окно About приобретает привлекательный вид. Осталось только расположить по центру пиктограмму FourUp, размещенную над именем приложения, и выбрать красивый и крупный шрифт для заголовка. Выберите текст заголовка и откройте диалоговое окно Text Properties. Теперь просто выберите из списка шрифтов новый шрифт и размер, который необходимо использовать.

Однако никакого списка шрифтов обнаружить не удается. Фактически, нет какого-либо простого способа выбора шрифта для индивидуальных компонентов в диалоговом окне. Если выбрать диалоговое окно About и открыть его свойства, можно заметить, что устанавливаются свойства шрифта для диалогового окна в целом. Однако упомянутым путем идти нежелательно, поскольку это дает два неприятных побочных эффекта:

- Весь текст в вашем диалоговом окне появляется с использованием нового выбранного шрифта. Выбор шрифта DinoBots для заголовка может создать иллюзию оригинальности, но скорее всего, не приведет к удовлетворительному результату при использовании этого шрифта для всех кнопок и текстовых полей.
- Размер диалогового окна вычисляется в единицах, определяемых на основе шрифта. Если шрифт изменяется, например, с 8-пунктного MS Sans Serif на 12-пунктный, то диалоговое окно серьезно увеличится в размерах, чтобы вместить необходимую информацию, выраженную в новых единицах измерения. Это, вероятно, не приведет к удовлетворительному результату.

Необходимо выяснить, каким образом можно добавлять красивый и крупный шрифт для вашего заголовка диалогового окна About. Ответ довольно прост и делает честь распорядителю растрового изображения — воспользуйтесь Visual C++ Bitmap Editor.

Создание ресурсов растровых изображений

Поскольку вам уже известны основные функции Bitmap Editor, не будем вдаваться в особые подробности. Вместо этого выделим те места, где Bitmap Editor работает с ресурсами растровых изображений иначе, чем в случае пиктограмм.

Для начала перетащите новое изображение элемента управления с Control Toolbar. Элемент управления Picture украшен изображением солнца и кактуса на переднем плане, что можно трактовать как символы жизни. Изображение элемента управления служит сразу нескольким целям. При первоначальном размещении этого изображения в диалоговом окне можно заметить, что оно представляет собой лишь пустой черный фрейм. Однако при открытии диалогового окна Properties можно заметить, что существует возможность выбора при отображении различных видов изображений. Выберите опцию Bitmap, как показано на рис. 3.19.

При активизации опции Bitmap вместо опции Frame в окне списка Type обратите внимание на то, что окно списка Image становится активным. К сожалению, список при просмотре оказывается пустым. Как это уже было проделано для загружаемых ранее пиктограмм, необходимо создать ресурс растрового изображения прежде, чем использовать его в качестве одного из элементов управления.

Для добавления нового растрового изображения выберите команду Insert | Resource из главного меню, и далее, выберите опцию Bitmap в диалоговом окне Insert Resource. Поскольку растровое изображение будет создаваться с нуля, щелкните на кнопке New вместо Import (см. рис. 3.20).

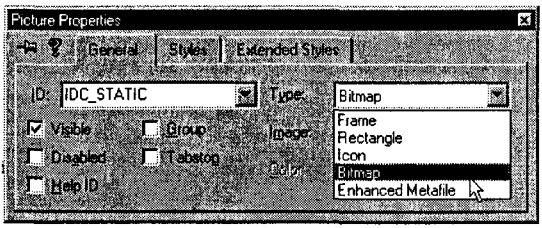


РИСУНОК 3.19 *Изменение опций для сохранения растрового изображения.*

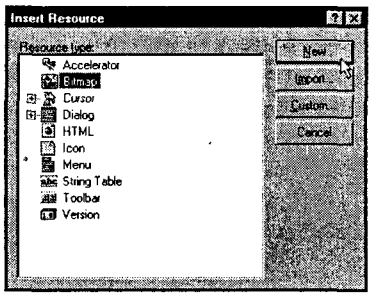


РИСУНОК 3.20 *Вставка пункта Bitmap в диалоговом окне Insert Resource.*

Поскольку проект FourUp ранее не содержал никаких ресурсов растровых изображений, то в окне ResourceView будет показана новая папка Bitmap. Откройте ее, и вы найдете ваше новое растровое изображение с идентификатором **IDB_BITMAP1**. (На настоящий момент можно лишь предполагать, что именно заменяет **IDB**.) На практике можно не беспокоится по поводу имени, поскольку оно не будет использоваться в вашей программе, а двойной щелчок на нем просто приводит к открытию утилиты Bitmap Editor, если она еще не открыта.

При первоначальном открытии ресурса растрового изображения — в противоположность ресурсу пиктограммы — можно заметить два отличия в характере поведения Bitmap Editor:

- Область, выделенная для рисования (или *холст*), может изменяться в размерах. Так не происходило с пиктограммами, которые имели фиксированный и определенный размер.
- Область, выделенная для рисования, имеет цвет фона. Пиктограммы используют заданный по умолчанию прозрачный цвет фона, но растровые изображения имеют непрозрачный фон.

Воспользуйтесь следующими шагами для создания нового логотипа приложения FourUp:

1. Растяните область, выделенную для рисунка, приблизительно на 100 единиц в ширину и 35 единиц в высоту, как указано в полях размеров в нижнем правом углу окна. При необходимости можно увеличивать изображение, а позже вырезать его в соответствии с размерами. Если нельзя увидеть всю текстовую область в большей панели для рисования, выберите пиктограмму Magnify (она похожа на лупу) из палитры Drawing и выберите из списка более низкое значение для разрешающей способности.
2. Выберите цвет фона, выполнив щелчок правой кнопкой мыши на светло-сером цвете образца цвета. Аналогичный цвет появляется в качестве фона в диалоговом окне, так что границы вашего растрового изображения отображаться не будут.
3. Используйте инструмент Fill (это похоже на распыление краски) для окрашивания фона. Для выполнения такой процедуры щелкните на инструмент в палитре Drawing и затем выполните щелчок правой кнопкой мыши где-либо в области рисунка. (Если выполнить щелчок левой кнопкой мыши, окраска производится с использованием цвета переднего плана.)
4. Замените цвет переднего плана на черный, выполняя щелчок левой кнопкой мыши на черном образце цвета.
5. Выберите инструмент Text, а затем соответствующее начертание шрифта и его размер (приблизительно 14 пунктов). Напечатайте слово "FourUp" в диалоговом окне инструмента Text. Щелкните на инструменте Text для установки нового содержимого.
6. Разместите текст ближе к нижнему правому полю вашей области рисунка. Если текст слишком большой, уменьшите размер шрифта. Если это не приводит к почти полному заполнению области, выделенной для рисования, выберите больший шрифт.
7. Измените снова цвет переднего плана, на сей раз — на красный.
8. Выберите инструмент Text и повторно напечатайте "FourUp" при помощи того же самого шрифта и размера. Теперь разместите красный текст поверх черного текста, так что черный текст выглядит подобно тени. Результаты показаны на рис. 3.21.
9. По завершении вашего растрового изображения (и после его сохранения при помощи команды File | Save) возвращайтесь к Dialog Editor и вторично выберите элемент управления Picture. В диалоговом окне Picture Properties в

раскрывающемся списке изображений теперь можно найти **IDB_BITMAP1**. Выберите его, и ваш новый привлекательный заголовок появится в диалоговом окне.

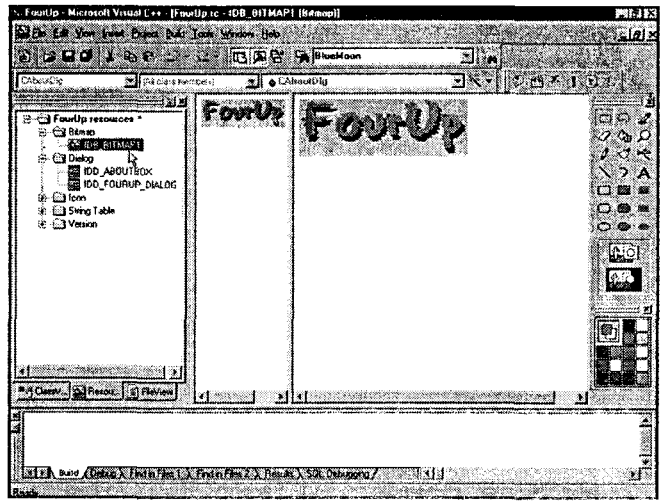


РИСУНОК 3.21
Использование Bitmap Editor в процессе создания заголовка для главного диалогового окна.

Последний элемент управления: группа

На данный момент диалоговое окно About выглядит довольно привлекательно. Присутствуют пиктограмма и пользовательский растровый заголовок, ваш статический текст и даже кнопка. О чем еще можно мечтать?

Прежде чем мы оставим в покое диалоговое окно About, давайте рассмотрим еще один элемент управления, расположенный на Control Toolbar — группу. Пиктограмма этого элемента управления походит на прямоугольник с символами xyz, расположенными в верхней части по центру (см. рис. 3.5).

Элемент управления группой подобен элементам управления статическим текстом. И пиктограммы, и растровые изображения в данном случае действительно *не выполняют* никаких функций — они существуют лишь для создания симпатичного внешнего вида. Разместите элементы управления в диалоговом окне About приложения FourUp в виде группы. Для этого необходимо выполнить следующее:

1. Перетащите элемент управления группой с Control Toolbar в диалоговое окно About.
2. Используйте Dialog Toolbar для включения сетки выравнивания. Это поможет равномерно разместить элемент управления группой.
3. Выберите и расширьте элементы управления таким образом, чтобы они находились на равных расстояниях от всех четырех сторон (приблизительно две единицы от каждого края, где каждая точка сетки представляет одну единицу). Необходимо обратить внимание на две вещи. Во-первых, пусть не волнует строка, проходящая вправо через кнопку OK, — этот недостаток будет сейчас устранен. Во-вторых, необходимо будет расположить прямоугольник выделения ближе к верхней части, чем к другим трем сторонам, поскольку выводимый прямоугольник дополняет текст, который появляется в названии окна.

4. Откройте лист свойств группы и удалите заголовок. По умолчанию каждая группа содержит в заголовке слово *Static* (см. рис. 3.22).

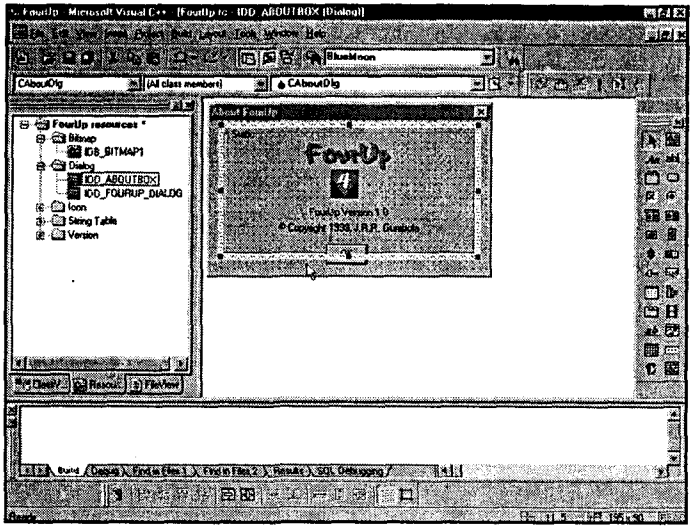


РИСУНОК 3.22 *Добавление элементов управления группой.*

5. Расширьте диалоговое окно в вертикальном направлении, чтобы появилось пустое пространство между верхней частью кнопки OK и нижней частью контура группы, подобно пространству, имеющемуся между нижней частью кнопки OK и нижней частью диалогового окна.

Заключительные штрихи

Хорошо, вы завершили формирование диалогового окна About и исследовали при этом множество новых инструментов. В следующей главе придется потратить некоторое время, разбираясь в коде, созданном AppWizard, что поможет вам лучше понять то, что фактически делают классы MFC **CWinApp**, **CWnd** и **CDialog**.

Прежде чем завершить рассмотрение этой темы, давайте последний раз посмотрим на FourUp и диалоговое окно About. Если вы скомпилируете и запустите на выполнение FourUp, то увидите, что используется ваша новая пиктограмма, однако меню отсутствует. Как же увидеть диалоговое окно About?

Все очень просто. Когда AppWizard создает приложение, основанное на использовании диалоговых окон, он добавляет диалоговое окно About в *системное* меню приложения. Просто щелкните на пиктограмме, которая появляется в названии окна, и в появившемся меню выберите About FourUp.

Итак, имеется завершенное диалоговое окно, которое можно видеть на рис. 3.23. Все что потребуется еще сделать — это реализовать оставшуюся часть приложения. Упомянутые действия будут предприняты в следующей главе.

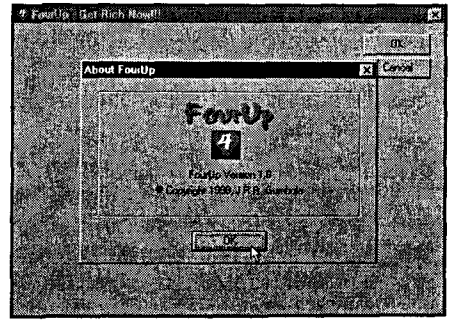


РИСУНОК 3.23 *Отображение завершенного диалогового окна About.*

Глава 4

Диалоговые окна

Диалоговые окна дают возможность пользователям осуществлять обратную связь с прикладной программой. В этой главе вы узнаете, как сделать это общение взаимовыгодным.

Возможно, и авторы согласны с тем, что о вкусах не спорят, самым крупным достижением кинематографа является кинофильм Луиса Малле (Louis Malle) *My Dinner with Andre* с участием таких звезд как Уоллис Шаун (Wallace Shawn) и Андре Грегори (Andre Gregory). Актер Шаун выступает в роли драматурга и актера, занимающего активную жизненную позицию, который принимает приглашение на обед от Грегори, старого друга, недавно возвратившегося из дальних многолетних странствий. Грегори в беседе со Шауном припоминает обстоятельства своего внезапного отъезда и многих странных происшествий, происшедших за это время — по существу, фильм представляет собой хаотичный диалог, поскольку Грегори проводит свои экстравагантные фантастические эксперименты, а Шаун стремится понять побуждения и поведение друга. Зрительская аудитория до самых последних минут фильма ожидает действия, но в фильме присутствуют только диалоги. Оказывается, вся соль именно в них. И лишь немногие зрители испытывает истинное наслаждение от оригинального суховатого стиля юмора, провозглашая фильм настоящей вершиной творческого гения.

Приложение FourUp, изучение которого начато в предыдущей главе, очень напоминает это явление. Оно также представляет собой сплошной диалог. Но не заблуждайтесь подобно Малле; фильм, представляющий собой сплошной диалог, а также приложение, являющее собой лишь только диалог, может быть весьма интересно, даже если ваши вкусы близки к обычным.

Структура приложения FourUp

По этому вопросу нет расхождений: MFC огромен, содержит более 200 классов, ряд из которых включают в себя сотни элементов данных и функций. Учитывая этот момент, MFC лишь незначительно уменьшает сложность неизменяющихся Windows API. Но, фактически, MFC значительно уменьшает сложность программ Windows, поскольку предлагает *архитектуру приложения*. Может присутствовать большое количество компонентов, однако все компоненты находятся в соответствии друг с другом. Практически каждая создаваемая программа MFC — от самой простой до наиболее сложной — может быть сведена к базовой структуре, показанной на рис. 4.1.

РИСУНОК 4.1 *Основная архитектура приложения MFC.*

Необходимые части

В левой части иллюстрации можно видеть две необходимых части — объект приложения и объект окна. Каждая программа MFC *должна* включать объект приложения, который отвечает за запуск и останов приложения.

Приложение обычно создает другой необходимый объект, коим является главное окно программы. (С технической точки зрения объект приложения представляет собой единственно необходимую часть приложения MFC, но вы, наверное, никогда не встретитесь с уединенным объектом приложения. Если программа с графическим пользовательским интерфейсом не содержит окон, то, вероятно, лучше всего использовать неграфический интерфейс пользователя.)

После того как главное окно создано и отображено, оба объекта продолжают работать вместе. Объект приложения выполняет доставку сообщений, восстанавливая сообщения из Windows. Объект окна отвечает на сообщения или передает их другим частям приложения.

В программах MFC можно реализовать эти два необходимых элемента в виде подклассов классов **CWinApp** и классы **CWnd**, как видно (в несколько сокращенном виде) на рис. 4.2. Эти классы, а также их предки, **CCmdTarget** и **CObject**, рассматриваются в текущей главе.

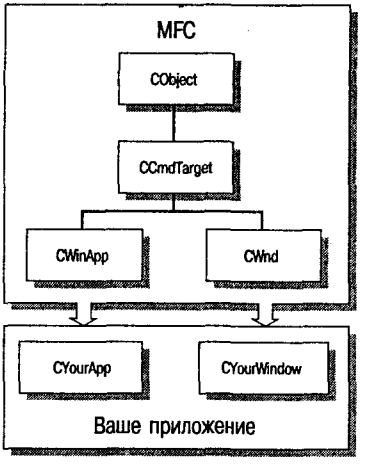


РИСУНОК 4.2 *Иерархия классов приложения MFC.*

Естественные сомнения

Большинство программ содержат дополнительные элементы сверх тех, которые необходимы для минимального приложения. Проект NotePod из главы 1, например, также включал класс документа и класс представления, которые были ответственны за управление и отображение данных приложения. Классы документа-представления обычно не отделяются, т.е. если используется один из них, то почти всегда применяется и второй.

Архитектура на основе документов и представлений идеально подходит для сложных приложений, потому что позволяет разделять работу, выполняемую программой на меньшие, самостоятельные части. Однако архитектура вносит некоторые дополнительные сложности, которые возникают вследствие необходимости понимания основ MFC. По этой причине мы не будем возвращаться к рассмотрению архитектуры "документ-представление" вплоть до главы 11.

Не беспокойтесь преждевременно по поводу такого разделения. Как будет видно, две других части — компоненты и ресурсы — в приложениях MFC даже более распространены, чем классы документов и представлений. Теперь сосредоточим свое внимание на их исследовании.

Компоненты или элементы управления?

Компоненты являются специализированными, отдельными оконными объектами, которые чаще всего формируют часть пользовательского интерфейса. Другими словами, компоненты обычно обеспечивают взаимодействие с пользователем, который привлекает их для целей управления программой. Поэтому компоненты обычно известны как *элементы управления (controls)*. Несмотря на то что элементы управления являются самостоятельными элементами, они не могут существовать сами по себе — они должны содержаться внутри другого окна, которое выполняет функции *родительского окна (parent window)*. По этой причине элементы управления иногда называют *элементами управления дочернего окна (child-window controls)*. MFC поддерживает шесть встроенных семейств элементов управления дочернего окна:

- **CStatic** — элементы управления, отображающие текст или пиктограммы, наподобие элементов управления изображением в панели Control Toolbar.
- **CButton** — кнопки, флажки, переключатели и групповые рамки.

- **CListbox** — элементы управления, которые отображают прокручиваемый список пунктов.
- **CComboBox** — элементы управления, которые отображают список пунктов переменной длины.
- **CScrollBar** — горизонтальные и вертикальные полосы прокрутки.
- **CEdit** — элементы управления одно- и многострочным редактированием текста.

Все версии Windows поддерживают упомянутые основные элементы управления. С разработкой Windows 95, а также с появлением Windows NT 4 и Windows 98, список доступных элементов управления продолжает расширяться. Действительно, используя элементы управления ActiveX (доступные из обширного круга источников), можно находить верный компонент практически для любой ситуации.

На рис. 4.3 показано шесть общих классов управления дочернего окна. Здесь также показано два других обычно используемых класса, унаследованных от **CWnd**, — **CFrameWnd** и **CDialog**.

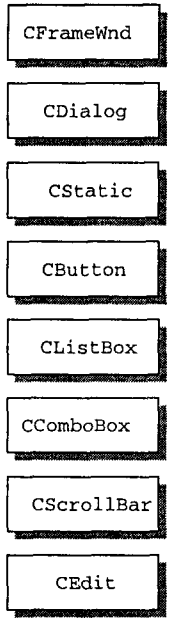


РИСУНОК 4.3 *Классы Windows, унаследованные от CWnd.*

Ресурсы

В дополнение к элементам управления, большинство приложений использует *ресурсы*. В программе Windows ресурсы представляют собой специальную форму данных, предназначенных только для чтения, и они связаны в выполняемый файл программой, называемой *компилятором ресурсов*. Ресурсы входят в два базовых множества:

- *Двоичные* — графические ресурсы, включающие пиктограммы, курсоры и растровые изображения.
- *Текстовые* — структурные ресурсы, включающие диалоговые окна, меню, таблицы строк и таблицы акселераторов.

Двоичные ресурсы хранятся в отдельных файлах, в то время как текстовые ресурсы хранятся в простом текстовом файле ASCII, который описывает структуру каждого элемента. Этот файл ASCII называется *описанием ресурсов* и обычно имеет то же самое имя, что и проект, но только с расширением **.rc**.

Каждая программа, использующая ресурсы, включает описание ресурсов. Текстовые ресурсы полностью содержатся внутри описания ресурсов; двоичные ресурсы просто вызываются через имена соответствующих файлов. При компиляции программы компилятор ресурсов объединяет текстовые и двоичные ресурсы в единое двоичное изображение (которое хранится в файле **res**). На этапе компоновки оно присоединяется к вашей выполняемой программе. На рис. 4.4 этот процесс подробно иллюстрируется.

Семейство FourUp

Прежде чем энергично взяться за дело и подробно исследовать эти части приложения, давайте окинем быстрым взглядом классы программы FourUp и выясним, какую функцию выполняет каждый из них.

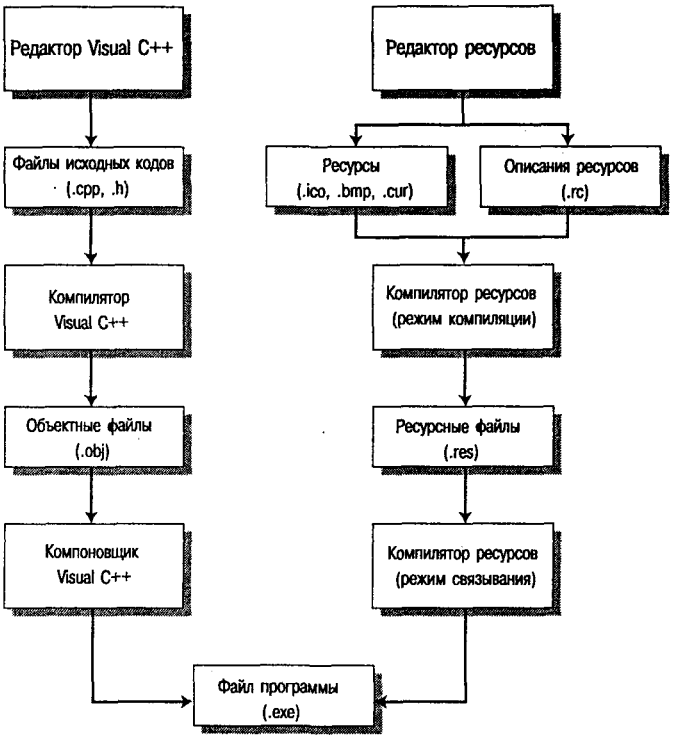


РИСУНОК 4.4
Исходный код, ресурсы и программы.

Ниже приводятся их краткие характеристики:

- *Объект приложения* — подобно почти любому приложению, приложение FourUp использует класс, унаследованный от **CwinApp**, как объект приложения. Класс называется **CFourUpApp** и его определение и реализация находятся в файлах FourUp.h и FourUp.cpp.
- *Объект окна* — FourUp использует класс, унаследованный от **CDialog**, как главное окно. **CDialog** — подкласс класса **CWnd**. Главный класс окна FourUp называется **CFourUpDlg**; определение класса и его реализация содержатся в файлах FourUpDlg.cpp и FourUpDlg.h.
- *Компоненты* — FourUp использует несколько элементов управления дочернего окна в дополнение ко второму классу, полученному из **CDialog** (**CAboutDlg**), который реализует окно About программы. Классы включают **CStatic** (растровое изображение, пиктограмма и разнообразные простые тексты) и **CButton** (групповая рамка и различные кнопки). Эти компоненты распределены между двумя файлами: схемы размещения для **CAboutDlg** и **CFourUpDlg** в сценарии ресурса и определения и реализации класса в — FourUpDlg.cpp.
- *Ресурсы* — FourUp использует как двоичные, так и текстовые ресурсы. Двоичные ресурсы включают растровое изображение для пользовательского заголовка и пиктограммы, представляющие каждый набор играющих карт, равно как и приложение. Текстовые ресурсы включают размещение для

главного экрана и диалогового окна About, а также некоторые ресурсы, сгенерированные Visual C++ (типа информации о версии). Текстовые ресурсы хранятся в файле с именем FourUp.rc, а двоичные ресурсы размещаются в подкаталоге ресурсов проекта.

Теперь, когда программа FourUp вкратце рассмотрена (как говорится, с высоты птичьего полета), остановимся на деталях. Начнем с класса приложения проекта, т.е. с **CFourUp**.

Знакомьтесь: объект приложения

Даже если вы — опытный программист на языках C/C++, вы, вероятно, будете удивлены, когда впервые откроете файл, сгенерированный при помощи AppWizard. Может сложиться впечатление, что код чрезвычайно странный и, кроме того, *что* означают эти очень уж непонятные комментарии:

```
//{{AFX_MSG(CAboutDlg)
//}}AFX_MSG
```

А что вы будете делать с такого рода строкой?

```
#define
AFX_FOURUP_H__2965C9C5_FBA4_11D1_837C_8CAD2F77F2B1
    __INCLUDED_
```

Что бы вы ни думали об этом, код AppWizard имеет вполне логическое объяснение. Он создан для улучшения работы инструментальных средств автоматизации типа ClassWizard, а не для того, чтобы его расшифровывал пользователь.

Поскольку код, предназначенный для ClassWizard, столь необычен, то маловероятно как спутать его с написанным вами кодом, так и поддаться соблазну изменять его. Также сомнительно, чтобы один из ваших идентификаторов случайно вошел в противоречие с идентификатором, выбранным AppWizard (даже если ваш вкус в выборе имен идентификаторов исключительно оригинален). Чтение программы, созданной AppWizard, сходно с чтением письма, написанного на двух различных языках. Если вы не понимаете один из языков, то просто перескакиваете далее, отыскивая части, имеющие смысл.

Рассмотрение объекта приложения FourUp начнем с изучения исходного кода для класса **CFourUpApp**.

Взгляд на CFourUp.h

Проект FourUp в качестве своего класса приложения использует **CFourUpApp**. Проекты AppWizard следуют практике, общепринятой при работе с C++ и состоящей в сохранении определения класса в файле заголовков, который использует расширение **.h**. Также принято сохранять реализацию (или фактические функциональные определения) для класса в отдельном файле с расширением **.cpp**. Файлы заголовков могут содержать встроенные функции, но они не создают переменные и не распределяют память.

Приложение FourUp следует этому соглашению, записывая файл заголовков для класса CFourUpApp под именем FourUp.h. Файл заголовков показан в листинге 4.1 — давайте рассмотрим его построчно.

Листинг 4.1 FourUp.h: главный файл заголовков для приложения FourUp.

```
// FourUp.h : главный файл заголовков для приложения FourUp
//
```

```
#if \
!defined(AFX_FOURUP_H__2965C9C5_FBA4_11D1_837C_8CAD2F77F2B1__INCLUDED_)
#define \
AFX_FOURUP_H__2965C9C5_FBA4_11D1_837C_8CAD2F77F2B1__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#ifndef __AFXWIN_H__
     #error include 'stdafx.h' before this file for PCH
#endif

# include "resource.h"          // основные символы

/////////////////////////////////////////////////////////////
// CFourUpApp:
// За реализацией класса обращайтесь в файл FourUp.cpp
//
class CFourUpApp : public CwinApp
{
public:
           CFourUpApp();

// Перекрытия
   // Перекрытия виртуальных функций, сгенерированные ClassWizard
   // {{AFX_VIRTUAL(CFourUpApp)
 public:
    virtual BOOL InitInstance();
    //}}AFX_VIRTUAL

// Реализация
// {{AFX_MSG(CFourUpApp)
            // ПРИМЕЧАНИЕ - Здесь ClassWizard будет добавлять и
            // удалять методы.
            // НЕ ИЗМЕНЯЙТЕ то, что видите в этих блоках
            // сгенерированного кода !
//}} AFX_MSG
     DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////////////////////////
   ////////////
//{{ AFX_INSERT_LOCATION}}
// Дополнительные объявления
// Microsoft Visual C++ будет вставлять непосредственно перед
// предыдущей строкой.
#endif //
!defined(AFX_FOURUP_H__2965C9C5_FBA4_11D1_837C_8CAD2F77F2B1__INCLUDED_)
```

Защита заголовка

Давайте начнем с предварительных замечаний. Обращает на себя внимание странное начало и конец файла. В C++ это не является чем-то необычным для файла заголовков, который в течение одной компиляции будет включаться не единожды. Обычно этого добиваются косвенным путем, включая два файла заголовков, каждый из которых требует общего заголовка.

Во избежание повторной обработки файла заголовков, могущей привести к появлению ошибок компиляции, следует *предохранять* заголовки C++, используя условную компиляцию. Рассмотрим, каким образом работает такая методика. Пред-

положим, что имеется файл заголовков **junk.h**, содержащий приведенные ниже строки:

```
#if !defined(JUNK)
// Здесь находятся условные строки заголовков
#endif
```

Если включается файл заголовков из другого файла, подобного такому:

```
#include "junk.h"
#define JUNK
#include "junk.h"
```

тогда компилятор обработает строки между первым **#if** и последним **#endif** во время первого включения, но не будет обрабатывать их во время второго включения. Препроцессор "пропустит" их. Итак, чем дальше, тем лучше.

Конечно, при определении, когда файл включается вторично, эта схема полагается на *пользователя*. В конце концов, в рассмотренном случае можно создавать код гораздо проще, не включая **junk.h** второй раз. Практически вы заинтересованы в том, чтобы файл заголовков автоматически следил за тем, был ли он уже включен. Как и следовало ожидать, это действительно предельно просто. Потребуется лишь изменить первоначальные строки подобно тому, как это сделано ниже:

```
#if !defined(JUNK)
#define JUNK
// Здесь находятся строки условных заголовков
#endif
```

Теперь, если заголовок включен впервые (во время единственной компиляции), идентификатор **JUNK** не будет определен, поэтому код, находящийся между **#if** и **#endif**, будет выполняться. Этот код немедленно определяет идентификатор **JUNK** в соответствие с **#define JUNK**. Итак, при следующем прочтении файла, его содержимое пропускается.

Однако такая методика не исключает возможность конфликта имен. Предположим, что идентификатор **JUNK** уже определен в другом файле заголовков, который не встречался вам уже несколько лет, и упомянутый файл заголовков включен ранее **junk.h**. Неожиданно компилятор перестает читать ваш файл заголовков. Имеется несколько способов исключить подобную ситуацию. AppWizard генерирует длинную непонятную строку и использует ее в качестве идентификатора-предохранителя. Ниже показан несколько измененный способ защиты файла заголовков для FourUp.h:

```
#if\
!defined(AFX_FOURUP_H_2965C9C5_FBA4_11D1_837C_8CAD2F77F2B1_INCLUDED_)
#define\
AFX_FOURUP_H_2965C9C5_FBA4_11D1_837C_8CAD2F77F2B1_INCLUDED_
//....
// Здесь находится тело файла заголовков
//....
#endif //\
!defined(AFX_FOURUP_H_2965C9C5_FBA4_11D1_837C_8CAD2F77F2B1_INCLUDED_)
```

Считается маловероятным, чтобы идентификатор, идентичный указанному в **#defined**, встретился где-либо еще в вашей программе.

Resource.h

Теперь рассмотрим включение файла заголовков resource.h при помощи строки:

```
#include "resource.h"
```

СОВЕТ

Не изменяйте файл заголовков Resource.h

Resource.h — это файл заголовков, поддерживаемый компьютером. Он содержит идентификаторы всех ресурсов, работа с которыми происходит при помощи редакторов ресурсов. Не изменяйте этот файл самостоятельно.

В листинге 4.2 показан файл **resource.h** для FourUp в том виде, в котором он преподносился в конце главы 3. (Он увеличится, поскольку добавлено большое количество элементов управления.)

Листинг 4.2 Файл resource.h приложения FourUp.

```
// Файл включения, сгенерированный Microsoft Developer Studio.
// Используется FourUp.rc
//
#define IDM_ABOUTBOX                    0x0010
#define IDD_ABOUTBOX                    100
#define IDS_ABOUTBOX                    101
#define IDD_FOURUP_DIALOG               102
#define IDR_MAINFRAME                   128
#define IDI_HEART                       129
#define IDI_CLUB                        130
#define IDI_DIAMOND                     131
#define IDI_SPADE                       132
#define IDB_BITMAP1                     133

// Далее следуют значения по умолчанию для новых объектов
//
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE        134
#define _APS_NEXT_COMMAND_VALUE         32771
#define _APS_NEXT_CONTROL_VALUE         1006
#define _APS_NEXT_SYMED_VALUE           101
#endif
#endif
```

Определение класса

Определение для класса **CFourUpApp** занимает большую часть CFourUp.h. Вместо того чтобы повторять каждый фрагмент кода, рассмотрим лишь некоторые моменты:

- Обратите внимание на то, что из самого заголовка следует, что класс публично унаследован от **CWinApp**. Это означает возможность вызова любого из методов, унаследованных от **CWinApp** или его суперклассов.
- Для упрощения поиска файлы заголовков, которые создает AppWizard, делятся на разделы. Заголовок начинается с раздела конструктора. Затем следует раздел, отмеченный комментарием:

```
// Overrides (перекрытия)
```

Этот раздел содержит виртуальные функции, которые вы можете расширять. Рассмотрим раздел, отмеченный комментарием:

```
// Implementation (реализация)
```

Этот раздел содержит код AppWizard, который может изменяться от версии к версии. Не следует полагаться на код в разделе реализации — вообще не изменяйте его.

- Наконец, обратите внимание на *AFX*-комментарии. В этом месте ClassWizard добавляет или удаляет функции и тому подобное. Как указывает один из комментариев, не следует редактировать что-либо, находящееся между маркерами комментария, которые выглядят следующим образом:

```
//{{
//}}
```

- Последняя строка файла заголовка содержит загадочную строку

```
DECLARE_MESSAGE_MAP()
```

Эта макрокоманда сообщает Visual C++, что класс будет отвечать на сообщения Windows. Эта запись и соответствующие записи отображаемых сообщений в файле реализации являются частью сложной макросистемы для формирования *таблиц ответных сообщений (message response tables)*. Таблица ответных сообщений гарантирует, что сообщения Windows направляются соответствующему методу. Нет необходимости в дополнительных знаниях для практического использования системы отображения сообщений (однако упомянутая тема будет рассмотрена более подробно при обсуждении возможностей клавиатуры и сообщений мыши). Как правило, используется ClassWizard для связи функции с соответствующими сообщениями — ClassWizard реализует поддержку таблиц ответных сообщений.

Исследование FourUp.cpp

Реализация класса **CFourUpApp** находится в файле FourUp.cpp. Вместо рассмотрения всего листинга целиком, обратим внимание на каждый раздел, объясняя назначение каждого фрагмента. Начнем с предварительных замечаний.

Включения и определения

В первом разделе FourUp.cpp включает файлы и определяет специальные константы. Рассмотрим три строки **#include**. Первая строка

```
#include "stdafx.h"
```

представляет собой стандартный файл заголовков каркаса приложения, используемый для всех программ MFC. Он вводит определения для стандартных компонент MFC, наиболее часто используемых расширений и стандартных элементов управления Internet Explorer 4.

В следующих двух строках вводятся файлы заголовка для двух файлов, которые составляют наш проект:

```
#include "FourUp.h"
#include "FourUpDlg.h"
```

Можно задаться вопросом, почему необходимо включать FourUpDlg.h, если он представляет собой файл реализации для класса приложения, а не класса окна. Причина этого очевидна. Достаточно отметить, что прежде чем завершится работа с этим файлом, могут понадобиться определения, сохраненные в FourUpDlg.h.

При формировании проекта C/C++ имеется возможность определить *построения окончательной версии (release build)* или *отладочное построение (debug build)*. По умолчанию в начале выполняется отладочное построение. Для случая отладочной версии MFC генерирует дополнительный код для проверки общих ошибок программирования. При переходе к этапу окончательного построения этот код удаляется. (Переключение между упомянутыми двумя возможностями осуществляется в диалоговом окне Build | Set Active Configuration (Построить | Активная конфигурация).)

Процесс распределения и освобождения памяти можно отследить только на этапе отладки. Если объект создается динамически при помощи оператора **new**, а затем вы забываете освободить используемую под него память, отладочная версия MFC оператора **new** сообщит, где и когда произошла ошибка. Эти возможности для класса **CFourUpApp** активизируются при помощи следующих программных строк:

```
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[ ] = __FILE__;
#endif
```

Отображения сообщений класса

После файлов заголовков и отладочных **#define** находится раздел отображения сообщений класса, как показано в листинге 4.3.

Листинг 4.3 Таблица ответных сообщений CFourUpApp.

```
/////////////////////////////////////////////////////////////
// CFourUpApp

BEGIN_MESSAGE_MAP(CFourUpApp, CWinApp)
   //{{AFX_MSG_MAP(CFourUpApp)
         // ПРИМЕЧАНИЕ - в этом месте ClassWizard добавляет и
         // удаляет макрокоманды отображения.
         // НЕ ИЗМЕНЯТЬ то, что находится в этих блоках
         // сгенерированного кода!
   //}}AFX_MSG
   ON_COMMAND(ID_HELP, CWinApp::OnHelp)
END_MESSAGE_MAP()
```

Таблица ответных сообщений начинается с макрокоманды **BEGIN_MESSAGE_MAP** и завершается макрокомандой **END_MESSAGE_MAP**. В промежуточных строках находятся макрокоманды, которые *отображают* индивидуальные сообщения на указанные функции.

Например, в отображении сообщения **CFourUpApp** макрокоманда **ON_COMMAND** (выделенная в листинге) связывает сообщение **ID_HELP** с функцией **CWinApp::OnHelp()**. Если во время выполнения программы объект **CFourUpApp** получает от Windows сообщение **ID_HELP**, генерируемое нажатием клавиши F1, выполняется вызов метода **CWinApp::OnHelp()**.

Несмотря на то что собственные записи отображения сообщений (разумеется, пока они не попадают в область "руки прочь" ClassWizard), это требуется редко. ClassWizard может выполнить работу намного лучше.

Конструктор CFourUpApp и объект приложения

Следующий раздел CFourUp.cpp включает конструктор класса и единственный глобальный экземпляр **CFourUpApp** с именем **theApp**. Код показан в листинге 4.4.

Листинг 4.4 Конструктор CFourUpApp и объект приложения.

```
////////////////////////////////////////////////////////
// Конструктор CFourUpApp

CFourUpApp::CFourUpApp()
{
        // ЧТО СДЕЛАТЬ: добавьте здесь код конструктора.
        // Поместите весь значащий код в
        // функцию инициализации экземпляра (InitInstance())
}

////////////////////////////////////////////////////////////////////////
   /////
//////////////////////
// Один и только один объект CFourUpApp
CFourUpApp theApp;
```

Поскольку класс **CFourUpApp** представляет именно ваше приложение, вас наверняка удивит столь небольшой код в конструкторе класса. Фактически *небольшой* — это еще мягко сказано; он отсутствует вовсе. Если попытаться совершить умозрительный переход от API функции **WinMain()** к конструктору **CFourUpApp**, то, скорее всего, возникнет путаница.

Путаница возникает вследствие того, что нет точного понимания, когда именно вызывается конструктор **CFourUpApp**. Вернувшись к классу C++, можно узнать ответ: конструктор вызывается при создании объекта **CFourUpApp**. Но *когда* точно создается объект **CFourUpApp**? Ответ находится в последней строке листинга 4.4, с которой мы вскоре познакомимся.

Во-первых, не забудьте, что каждая программа может иметь только один объект приложения. В конце концов, объект приложения фактически представляет собой программу. Во-вторых, обратите внимание на то, что объект **theApp** из **CFourUpApp** создается как глобальный. В результате его конструктор вызывается при загрузке программы — перед вызовом **WinMain()** и до того, как многие подсистемы MFC получат возможность завершить свою инициализацию. Это происходит лишь *после* конструирования основного объекта приложения, который вообще вызывается **WinMain()**. Таким образом, если необходимо совершить некоторую инициализацию внутри вашего класса приложения, то конструктор здесь — не помощник. Вместо этого класс **CFourUpApp** перекрывает виртуальную функцию **InitInstance()**. При помощи **InitInstance()** выполняется инициализация приложения, которую программа API осуществляет в рамках **WinMain()**.

К счастью, **InitInstance()** — единственная функция, которую осталось исследовать в классе **CFourUpApp**. Приступим к ее рассмотрению.

Функция CFourUpApp::InitInstance()

Каждый раз при загрузке программы Windows происходит повторный вызов функции **WinMain()**. Программы MFC в этом смысле ничем не отличаются — они также включают функцию **WinMain()**. Но в отличие от программы Windows, основанной на API, функция **WinMain()** программы MFC глубоко спрятана внутри самой библиотеки MFC.

Если возникает впечатление, что программы MFC являются в какой-то степени менее универсальными или не столь гибкими по сравнению с программами API, то это не так. Почти в каждой программе API функции **WinMain()**, по существу, идентичны. MFC скрывает код шаблона, затем дает вам "привязки" к частям **WinMain()**, которые, вероятно, потребуется изменить. Эти привязки имеют вид виртуальных функций, которые вызываются из собственной функции **WinMain()** библиотеки MFC. При первоначальном рассмотрении класса **CWinApp** можно заметить еще несколько виртуальных функций. Однако наиболее важной является функция **InitInstance()**.

При запуске функция **WinMain()** версии MFC сначала получает указатель на объект приложения. Поскольку объект приложения является глобальной переменной, то он уже присутствует на этапе запуска **WinMain()**. Затем **WinMain()** использует этот указатель для вызова метода **InitInstance()** объекта приложения. Поскольку **InitInstance()** — виртуальная функция, то **WinMain()** фактически вызывает версию, созданную в вашем перекрытом приложении.

В листинге 4.5 показана версия **InitInstance()** из **CFourUpApp**. Чуть позже будут рассмотрены некоторые ключевые моменты.

Листинг 4.5 Функция CFourUpApp::InitInstance().

```
//////////////////////////////////////////////////////////
// Инициализация CFourUpApp
BOOL CFourUpApp::InitInstance()
{
// Стандартная инициализация
// Если эти возможности не используются и необходимо
// уменьшить размер заключительной выполняемой
// программы, потребуется удалить из следующего текста
// специфические подпрограммы
// инициализации, потребность в которых отсутствует.

#ifdef _AFXDLL
   Enable3dControls(); // Вызов в случае использования версии MFC
   общего доступа
#else
   Enable3dControlsStatic(); // Вызов для случая статической привязки
#endif

   CFourUpDlg dlg;
   m_pMainWnd = &dlg;
   int nResponse = dlg.DoModal();
   if (nResponse == IDOK)
   {
        // ЧТО СДЕЛАТЬ: Поместите здесь код,
        // обрабатывающий закрытие диалога по нажатию кнопки OK
   }
   else if (nResponse == IDCANCEL)
   {
```

```
        // ЧТО СДЕЛАТЬ: Поместите здесь код,
        // обрабатывающий закрытие диалога по нажатию кнопки Cancel
    }
    // Так как диалог прерван, то возвращается FALSE,
    // поэтому мы выходим из приложения, прежде чем
    // начать процесс накопления сообщений приложения.
    Return FALSE;
}
```

Работа функции InitInstance()

В приложении API функция **WinMain()** решает три задачи, а именно:

1. Регистрация нового класса главного окна.
2. Создание экземпляра главного окна и его отображение.
3. Запуск процесса накопления сообщений.

Функция **InitInstance()** в полной мере соответствует второму из этих требований для MFC-версии функции **WinMain()**. Рассмотрим, каким образом функция **CFourUpApp::InitInstance()** выполняет упомянутую задачу.

Во-первых, создается новое главное окно:

```
CFourUpDlg dlg;
```

Во-вторых, происходит инициализация элемента данных **m_pMainWnd** так, что он указывает на новое окно:

```
m_pMainWnd = &dlg;
```

В-третьих, при помощи метода **DoModal()** отображается главное окно:

```
int response = dlg.DoModal();
```

Поскольку главное окно приложения FourUp основано на классе **CDialog**, то здесь представлен весь требуемый код. Функция **DoModal()** отображает главное окно, затем обрабатывает все направляемые в него сообщения до тех пор, пока пользователь не щелкнет на кнопках OK или Cancel, расположенных в главном окне. Если это случается, то главное окно закрывается и **DoModal()** возвращает идентификатор кнопки, на которой был выполнен щелчок. Метод **InitInstance()** сохраняет значение локальной переменной с именем **response**, и AppWizard записывает каркасы обработчиков для обоих кнопок OK и Cancel, основываясь на возвращенном значении. Разумеется, что можно все реализовать и по-своему, либо оставить все как есть. Все, что потребуется предпринять — это изменить код, сгенерированный AppWizard.

Последняя строка в **InitInstance()** возвращает **FALSE**. Это пригодно только для приложений, основанных на диалоговых окнах. Приложения, отличные от основанных на диалоговых окнах, возвращают **TRUE**, чтобы запустить процесс накопления сообщений **WinMain()**.

Все о CWinApp

Как можно заметить на рис. 4.1 в начале главы, **CFourUpApp** наследуется от **CWinApp**. В документации по MFC рассматриваются несколько "могущих быть перекрытыми" виртуальных функций, определенных в классе **CWinApp**. Ниже приведены четыре наиболее интересных из них:

- ***InitInstance()*** — всегда будет перекрываться.
- ***Run()*** — просматривает цикл сообщений и обрабатывает сообщения.
- ***OnIdle()*** — вызывается **Run()**, если не обнаружено никаких сообщений. **OnIdle()** дает возможность выполнять фоновые задачи, которые в противном случае могли бы замедлить ответную реакцию системы.
- ***ExitInstance()*** — вызывается **Run()** при завершении приложения.

В **CFourUpApp** перекрывается только **InitInstance()**. В большинстве программ оставшиеся функции перекрываться не должны, поскольку они устроены так, чтобы библиотека MFC смогла отвечать даже в самых необычных ситуациях.

Что такое CCmdTarget?

Вновь посмотрите на рис. 4.2. Несложно заметить, что класс **WinApp** является производным от класса **CCmdTarget**. **CCmdTarget** — это базовый класс для объектов, получающих сообщения и отвечающих на них. Он поддерживает элементы данных и методы, которыми должны обладать все подобные объекты. Этот класс является достаточно плодовитым родительским классом — среди его подклассов: **CWinApp**, **CWnd**, **CFrameWnd**, **CView** и **CDocument**. Кроме того, **CCmdTarget** имеет множество классов-"внуков" (т.е. дочерних от дочерних) — ввиду того, что элементы управления вообще являются производными от **CWnd**, они также входят и в число потомков **CCmdTarget**.

Куда попадает объект CObject?

Опять-таки, при просмотре рис. 4.2 можно заметить, что **CCmdTarget** является производным от **CObject**. **CObject** представляет собой корневой класс всех объектов MFC; каждый объект MFC является потомком **CObject**, и **CObject** не имеет какого-либо родительского класса. Если необходимо создать объект MFC нового вида, и ни один из существующих объектов MFC не подходит в качестве хорошей отправной точки, то новый класс можно получить из **CObject**. Таким образом он будет мирно сосуществовать с другими классами MFC.

Обзор окон

Приложение FourUp использует два типа окон: главное и диалоговое. Главные окна ведут свое происхождение от плодовитого класса **CWnd**, в то время как диалоговые окна наследуются от **CDialog**. Давайте рассмотрим более подробно эти два важных общих класса.

Все о CWnd

CWnd является наиболее примечательным дочерним классом для **CCmdTarget**, а также базовым классом для главных окон. Как уже упоминалось, **CWnd** служит также родительским классом для многих других важных классов MFC, включая **CFrameWnd**, **CMDIFrameWnd**, **CView**, **CDialog**, **CButton**, **CControlBar**, **CToolbar** и т.д. Взгляните еще раз на рис. 4.3.

Приложение FourUp использует диалоговое окно в качестве главного, так что объект окна является экземпляром подкласса **CWnd** — **CDialog**. Рассмотрим универсальный класс **CDialog**.

Познакомьтесь: класс CDialog

Диалоговые окна имеют два ассоциированных программных объекта: ресурс и объект, который является экземпляром класса, полученного из **CDialog**. Ресурс хранит свойства диалогового окна, подобные элементам управления, находящимся в его распоряжении, вместе с их экранными позициями. Объект наследует данные и методы от **CDialog** и его предков: **CWnd**, **CCmdTarget** и **CObject**.

Приложение FourUp содержит два диалоговых окна: одно служит главным окном приложения (**CFourUpDlg**), а другое — окном About приложения (**CAboutDlg**). Рассмотрим сперва **CAboutDlg**, а затем вернемся к **CFourUpDlg**.

Краткое рассмотрение CAboutDlg

Исходный код для класса **CAboutDlg** показан в листинге 4.6. Уделите некоторое время на его исследование, а затем мы подробно рассмотрим код.

Листинг 4.6 Класс CAboutDlg.

```
////////////////////////////////////////////////////////
// Диалоговое окно CAboutDlg

class CAboutDlg : public CDialog
{
public:
     CAboutDlg();

// Данные диалогового окна
      //{{AFX_DATA(CAboutDlg)
      enum { IDD = IDD_ABOUTBOX };
      //}}AFX_DATA

      // Перекрытия виртуальных функций, сгенерированные ClassWizard
      //{{AFX_VIRTUAL(CAboutDlg)
      protected:
      virtual void DoDataExchange(CDataExchange* pDX);
      //}}AFX_VIRTUAL

// Реализация
protected:
      //{{AFX_MSG(CAboutDlg)
      //}}AFX_MSG
      DECLARE_MESSAGE_MAP()
};

CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
      //{{AFX_DATA_INIT(CAboutDlg)
      //}}AFX_DATA_INIT
}

void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
      CDialog::DoDataExchange(pDX);
      //{{AFX_DATA_MAP(CAboutDlg)
      //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
      //{{AFX_MSG_MAP(CAboutDlg)
```

```
        // Обработчики сообщений отсутствуют
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

Первым делом вы, вероятно, отметили, что **CAboutDlg** не содержит каких-либо отображений сообщений:

```
BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
    //{{AFX_MSG_MAP(CAboutDlg)
        // Обработчики сообщений отсутствуют
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

Как и в **CFourApp**, здесь присутствуют строки **BEGIN_MESSAGE_MAP** и **END_MESSAGE_MAP**. Однако **CAboutDlg** *не должен обрабатывать сообщения* Windows, поэтому обработчиков сообщения нет, о чем, собственно, и говорит комментарий.

Другое новшество содержится в строках

```
protected:
    virtual void DoDataExchange(CDataExchange* pDX);
```

Показанные строки обеспечивают перекрытие унаследованной функции **DoDataExchange()**. Эта функция поддерживает обмен данными диалога (DDX — dialog data exchange) и проверку данных диалога (DDV — dialog data verification), о чем подробно рассказывается в главе 18. DDX и DDV упрощают работу с диалоговыми окнами, содержащими элементы управления, которые организуют ввод и отображение данных. DDX позволяет связывать элемент данных с элементом управления, типа текстового поля, таким образом, чтобы значение элемента данных всегда отражало содержимое элемента управления. DDV позволяет проверять вводимые пользователем данные на предмет соответствия набору правил. Подобного рода помощь гарантирует, что в обработку вовлекаются только допустимые данные. Поскольку диалоговое окно About не содержит динамических элементов управления, поддержка DDX и DDV не требуется. Тем не менее, во избежание различных неприятностей упомянутую поддержку лучше оставить.

Путешествие по CFourUpDlg

Листинг 4.7 показывает файл заголовков FourUpDlg.h. Если быть более точным, здесь показана сокращенная версия файла, который содержит буквально на несколько строк больше, опущенных здесь в связи с их неуместностью. Внимательно исследуйте листинг. Если ваше любопытство не удовлетворено, воспользуйтесь редактором Visual C++ откройте файл из сопровождающего книгу CD-ROM и просмотрите строки, отсутствующие в листинге.

Листинг 4.7 Файл заголовков FourUpDlg.h (сокращенный вариант).

```
//////////////////////////////////////////////////////////
// Диалоговое окно CFourUpDlg

class CFourUpDlg : public CDialog
{
// Конструктор
public:
    CFourUpDlg(CWnd* pParent = NULL); // стандартный
```

```
        // Данные диалогового окна
        // {{AFX_DATA(CFourUpDlg)
        enum { IDD = IDD_FOURUP_DIALOG };
        //}} AFX_DATA

        // Перекрытия виртуальных функций, сгенерированные ClassWizard
        //{{AFX_VIRTUAL(CFourUpDlg)
        protected:
        virtual void DoDataExchange(CDataExchange* pDX);
        //}}AFX_VIRTUAL

// Реализация
protected:
        HICON m_HICON;

// Сгенерированные функции отображения сообщений
        //{{AFX_MSG(CFourUpDlg)
        virtual BOOL OnInitDialog();
        Afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
        Afx_msg void OnPaint();
        Afx_msg HCURSOR OnQueryDragIcon();
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
};
```

Файл содержит пять главных разделов, представляющих интерес:

- *Конструктор (Construction)* — содержит конструктор **CFourUpDlg**.
- *Данные диалогового окна (Dialog Data)* — определяет перечисление C++, относящееся к диалоговому окну.
- *Перекрытия виртуальных функций, сгенерированные ClassWizard (Virtual function overrides generated by Class Wizard)* — содержат единственную перекрытую функцию **DoDataExchange()**, которая в действительности не нужна, поскольку поддержка DDX/DDV не требуется.
- *Реализация (Implementation)* — определяет элементы данных, которые относятся к пиктограмме диалогового окна.
- *Функции отображения сообщений (Message map functions)* — связывают сообщения с соответствующими обработчиками.

Поскольку CFourUpDlg.h является файлом заголовков, в нем находятся только прототипы функций. Файл реализации CFourUpDlg.cpp содержит тела функции, которые появляются в файле заголовка. Рассмотрим поэтапно файл реализации, начиная с листинга 4.8, где показан конструктор класса.

Листинг 4.8 Конструктор класса CFourUpDlg.

```
CFourUpDlg::CFourUpDlg(CWnd* pParent /*=NULL*/)
    : CDialog(CFourUpDlg::IDD, pParent)
{
        //{{AFX_DATA_INIT(CFourUpDlg)
        //}}AFX_DATA_INIT
        m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}
```

В то время как конструктор **CFourUpApp** был пуст, конструктор **CFourUpDlg** содержит строку, которая загружает значение в элемент данных **m_hIcon**. Этот

элемент данных содержит пиктограмму, используемую для представления диалогового окна при его минимизации. Получается значение, использующее функцию каркаса приложения MFC **AfxGetApp()**, которая возвращает указатель на объект приложения. Затем при помощи этого указателя осуществляется доступ к методу **LoadIcon()**, который возвращает соответствующую пиктограмму как определено в данных ресурса приложения.

В соответствии с **CFourUpApp** большая часть инициализации **CFourDlg** выполняется внутри функции **OnInitDialog()**, показанной в листинге 4.9. Напомним, что каркас приложения автоматически вызывает эту функцию после загрузки приложения.

Листинг 4.9 Функция CFourUpDlg::OnInitDialog().

```
BOOL CfourUpDlg::OnInitDialog()
{
      CDialog::OnInitDialog();

      // Пункт меню "Add About..." из системного меню.

      // IDM_ABOUTBOX должен быть в диапазоне системных команд.
      ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);
      ASSERT(IDM_ABOUTBOX < 0xF000);

      CMenu* pSysMenu = GetSystemMenu(FALSE);
      if (pSysMenu != NULL)
      {
            CString strAboutMenu;
            strAboutMenu.LoadString(IDS_ABOUTBOX);
            if (!strAboutMenu.IsEmpty())
            {
                  pSysMenu->AppendMenu(MF_SEPARATOR);
                  pSysMenu->AppendMenu(MF_STRING, IDM_ABOUTBOX,
   strAboutMenu);
            }
      }
// Установить пиктограмму диалогового окна.
// Каркас делает это автоматически, если главное окно приложения
// не является диалоговым
      SetIcon(m_hIcon, TRUE); // Установить большую пиктограмму
      SetIcon(m_hIcon, FALSE); // Установить малую пиктограмму

// ЧТО СДЕЛАТЬ: Здесь добавить дополнительный код
// инициализации

      return TRUE;  // вернуть TRUE, если не установлен фокус
                    // на какой-то элемент управления
}
```

OnInitDialog() сперва добавляет пункт меню About в системное меню приложения. Однако перед переходом к этой задаче **OnInitDialog()** выполняет пару макрокоманд **ASSERT**, которые проверяют, является ли соответствующим код сообщения, назначенный пункту меню About (т.е. **IDM_ABOUTBOX**). Макрокоманды ASSERT не играют особой роли до тех пор, пока вы не начнете преднамеренно изменять свойства ресурса, однако MFC все же на этом этапе проявляет особую осторожность.

Для добавления пункта меню функция **OnInitDialog()** получает указатель на системное меню приложения. Если указатель имеет значение NULL, производится обход неудачной попытки добавления пункта меню. Иначе используется функция **LoadString()** для загрузки текстового ресурса, идентифицируемого значением **IDS_ABOUTBOX,** который содержит требуемое описание пункта меню. Напоминаем, что необходимость генерации кода для поля About была затребована при создании приложения FourUp. В ответ на ваш запрос AppWizard, наряду с самим кодом, автоматически создал текстовый ресурс.

Далее функция **OnInitDialog()** вызывает функцию **AppendMenu()** один раз для добавления разделителя меню (горизонтальной линейки) и второй раз — для добавления пункта меню. (Напомним, что сейчас осуществляется лишь беглый просмотр, а не полное объяснение всех возможных случаев — меню будет рассмотрено позже.) Наконец, функция устанавливает пиктограмму окна диалога, используя предварительно загруженное значение **m_hIcon**, после чего возвращает **TRUE**.

Все, что остается от класса **CFourUpDlg**, — это отображение сообщений функции обработчиков сообщений. Отображение сообщений показано в листинге 4.10. Обратите внимание, что это отображение обеспечивает связывание с тремя сообщениями:

- **WM_SYSCOMMAND**
- **WM_PAINT**
- **WM_QUERYDRAGICON**

Для выявления имени сообщения в информации, находящейся в отображении сообщений, отбросьте префикс **ON_**. Давайте рассмотрим каждый обработчик упомянутых сообщений.

Листинг 4.10 Отображение сообщения CFourUpDlg.

```
BEGIN_MESSAGE_MAP(CFourUpDlg, CDialog)
      //{{AFX_MSG_MAP(CFourUpDlg)
      ON_WM_SYSCOMMAND()
      ON_WM_PAINT()
      ON_WM_QUERYDRAGICON()
      //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

В листинге 4.11 показана функция **OnSysCommand()**, которая обрабатывает сообщение **WM_SYSCOMMAND**. Она вызывается всякий раз, когда пользователь выбирает какой-либо пункт в системном меню. Функция проверяет, выбрал ли пользователь пункт меню About. Если это так, создается экземпляр **CAboutDlg** и вызывается функция **DoModal()**, открывающая диалоговое окно; в противном случае параметры сообщения передаются в обработчик сообщений суперкласса (**CDialog**) для заданной по умолчанию обработки.

Листинг 4.11 Функция CFourUpDlg::OnSysCommand().

```
void CFourUpDlg::OnSysCommand(UINT nID, LPARAM lParam)
{
      if ((nID & 0xFFF0) == IDM_ABOUTBOX)
      {
```

```
            CAboutDlg dlgAbout;
            dlgAbout.DoModal();
        }
        else
        {
            CDialog::OnSysCommand(nID, lParam);
        }
}
```

В листинге 4.12 показана функция **OnPaint()**, которая обрабатывает сообщение **WM_PAINT**. Каркас приложения автоматически обрабатывает сообщение, когда оно пересылается в главное окно приложения типа "документ-представление". Однако FourUp не является приложением такого типа, поэтому AppWizard генерирует код обработки сообщения. Сообщение **WM_PAINT** указывает, что окно должно быть перерисовано. В **CFourUpDlg** это происходит только тогда, когда приложение минимизировано и необходимо изобразить пиктограмму приложения. Код, выполняющий рисование несколько сложен, поэтому он в настоящий момент не рассматривается; рисованию посвящена глава 7.

Теперь обратите внимание, что отрисовка выполняется только тогда, когда функция **IsIconic()** возвращает значение TRUE, указывая на то, что приложение минимизировано. В противном случае сообщение **WM_PAINT** передается на обработку родительскому классу.

Листинг 4.12 Функция CFourUpDlg::OnPaint().

```
// В случае добавления к диалоговому окну кнопки минимизации
// потребуется нижеприведенный код, обеспечивающий отрисовку
// пиктограммы. Для приложения MFC, использующего
// модель "документ/представление", это
// выполняется автоматически каркасом.

void CFourUpDlg::OnPaint()
{
    if (IsIconic())
    {
        CPaintDC dc(this); // контекст устройства для рисования

        SendMessage(WM_ICONERASEBKGND,
                    (WPARAM) dc.GetSafeHdc(), 0);

// Пиктограмма, центрированная в клиентском прямоугольнике
int cxIcon = GetSystemMetrics(SM_CXICON);
        int cyIcon = GetSystemMetrics(SM_CYICON);
        CRect rect;
        GetClientRect(&rect);
        int x = (rect.Width() - cxIcon + 1) / 2;
        int y = (rect.Height() - cyIcon + 1) / 2;

        // Рисование пиктограммы
        dc.DrawIcon(x, y, m_hIcon);
    }
    else
    {
        CDialog::OnPaint();
    }
}
```

Последний обработчик сообщений класса **CFourUpDlg** — это **OnQueryDragIcon()** (см. листинг 4.13), связанный с сообщением **WM_QUERYDRAGICON**. Каркас приложения посылает это сообщение с целью обнаружения пиктограммы, отображаемой, когда пользователь перетаскивает минимизированное окно приложения. Поскольку вполне подходящим выбором является пиктограмма приложения, AppWizard включает код, который возвращает указатель на соответствующую пиктограмму (полученную из элемента данных **m_hIcon**, инициализированного в конструкторе класса). Далее указатель приводится к требуемому типу (**HCURSOR**, который представляет дескриптор курсора).

Листинг 4.13 Функция CFourUpDlg::OnQueryDragIcon().

```
// Код вызывается системой для получения курсора, отображаемого
// во время перетаскивания минимизированного окна
HCURSOR CFourUpDlg::OnQueryDragIcon()
{
      return (HCURSOR) m_hIcon;
}
```

Введение в ресурсы

Приложение FourUp использует три основных типа ресурсов: пиктограммы, растровые изображения и диалоговые окна. В последующих главах будут рассмотрены другие типы ресурсов, включая курсоры, меню и строки. В этом разделе мы обсудим сценарии ресурсов и компилятор ресурсов. Вы также получите первоначальное представление о ресурсных данных, управляемых Visual C++, для каждого типа ресурса, используемого в приложении FourUp.

Сценарий ресурса и компилятор ресурсов

Если вновь вернуться к рис. 4.4, можно заметить, что файлы CPP и OBJ, образующие часть C++ в программе MFC — это только половина истории. Второй процесс компиляции выполняется параллельно с процессом обработки исходного кода. Этот второй процесс компиляции имеет отношение к ресурсам, которые используются в приложении.

Resource Editor (Редактор ресурсов) уже применялся при создании пиктограммы, растрового изображения и диалогового окна. Однако, скорее всего, вы не обратили внимание на то, что за кулисами Visual C++ создал файл сценария ресурсов, в котором были записаны результаты проделанной работы. Пиктограмма и растровое изображение были сохранены, соответственно, в файлах ICO и BMP. Но, кроме того, Visual C++ создает RC-файл, который идентифицирует ресурсы, используемые в вашей программе, определяет их свойства и сообщает, какой файл содержит пиктограмму, а какой — растровое изображение.

При компиляции программы RC-файл также компилируется при помощи утилиты, известной как компилятор ресурсов. Последний создает двоичный RES-файл. Затем компоновщик объединяет этот файл с EXE-файлом приложения; таким образом, все данные ресурса оказываются доступными во время выполнения.

Чтобы представить методику управления ресурсами Visual C++, давайте рассмотрим записи ресурсного файла для каждого типа ресурса, используемого приложением FourUp.

Ресурсы пиктограмм

В листинге 4.14 показаны части файла ресурса FourUp, относящиеся к пиктограммам. Обратите внимание, что приложение определяет четыре пиктограммы — в точности те же, что и создаваемые ранее. Файл описывает каждую пиктограмму в четыре столбца:

- Имя пиктограммы, заданное явной константой.
- Тип ресурса (ICON).
- Свойства ресурса (например, **DISCARDABLE**, указывающее на ресурс только для чтения, который может быть выгружен памяти без сохранения копии).
- Файл, содержащий пиктограмму, вместе с путем относительно каталога проекта.

Поскольку файл ресурса — это текстовый файл, его можно редактировать при помощи обычного текстового редактора. В прошлом программисты в среде Windows были вынуждены поступать именно так. Однако, с одной стороны, Resource Editor *Visual C++ создает и модифицирует файлы ресурса намного проще. С одной стороны*, уменьшается возможность возникновения ошибок при вводе, приводящих к ошибкам компиляции ресурса.

Листинг 4.14 Записи сценария FourUp.rc ICON.

```
//////////////////////////////////////////////////////////
//
// Пиктограмма
//

// Первой размещается пиктограмма с самым малым значением ID,
// что гарантирует совместимость пиктограмм приложений во
// всех системах.
IDR_MAINFRAME       ICON        DISCARDABLE       "res\\FourUp.ico"
IDI_HEART           ICON        DISCARDABLE       "res\\Misc34.ico"
IDI_CLUB            ICON        DISCARDABLE       "res\\Misc35.ico"
IDI_DIAMOND         ICON        DISCARDABLE       "res\\Misc36.ico"
IDI_SPADE           ICON        DISCARDABLE       "res\\Misc37.ico"
```

Все о ресурсах растровых изображений

В листинге 4.15 показана часть файла ресурсов FourUp, которая относится к отдельному растровому изображению (bitmap). Обратите внимание, что формат записи ресурса растрового изображения, по существу, такой же, как и для ресурса пиктограммы.

Листинг 4.15 Записи сценария FourUp.rc.

```
//////////////////////////////////////////////////////////
//
// Растровое изображение
//

IDB_BITMAP1     BITMAP      DISCARDABLE       "res\\bitmap1.bmp"
```

Ресурсы диалоговых окон

В листинге 4.16 показана часть файла ресурсов FourUp, относящаяся к диалоговому окну About. Обратите внимание, что для диалоговых окон в файле записывается больший объем информации, нежели для пиктограмм или растровых изображений. В частности, в файле ресурсов записываются тип, позиция и размеры каждого элемента управления принадлежащего диалоговому окну.

Листинг 4.16 Записи сценария FourUp.rc, относящиеся к диалоговому окну About

```
//////////////////////////////////////////////////////////
//
// Диалоговое окно
//

IDD_ABOUTBOX DIALOG DISCARDABLE   0, 0, 217, 125
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "About FourUp"
FONT 8, "MS Sans Serif"
BEGIN
   CTEXT   "FourUp 1.0",IDC_STATIC,49,61,119,8,SS_NOPREFIX
   CTEXT    "© 1998, Bill and Steve",IDC_STATIC,49,72,119,8
   CONTROL   133,IDC_STATIC,"Static",SS_BITMAP,74,15,69,22
   ICON      IDR_MAINFRAME,IDC_STATIC,95,35,21,20
   GROUPBOX  "",IDC_STATIC,11,5,195,90
END
```

Если простота записей ресурса для пиктограмм и растровых изображений могла натолкнуть на мысль серьезно рассмотреть использование текстового редактора, есть надежда, что этот листинг предупредит ваше опрометчивое решение. Resource Editor действительно мощное, и в то же время, простое в применении инструментальное средство.

Завершение исследования ресурсов: краткое повторение

В главе было рассмотрено множество основных понятий. Эти понятия будут глубоко исследоваться в оставшейся части книги. Вы изучили структуру простого приложения, основанного на использовании диалоговых окон. Кроме того, вы изучили ключевые объекты, которые использует любое приложение MFC: объект приложения (экземпляр подкласса **CWinApp**) и объект главного окна (экземпляр подкласса **CWnd**). Вы также вплотную приблизились к рассмотрению типовых подклассов **CWinApp** и **CWnd**. В заключение вы заглянули за кулисы и увидели то, как Visual C++ управляет ресурсными данными.

В следующей главе будет завершено создание приложения FourUp. Оставшийся объем работы концентрируется на элементах управления: кнопки, метки и другие элементы, присущие приложениям с графическим пользовательским интерфейсом. Смело двигайтесь дальше — скоро вы получите завершенное рабочее приложение MFC.

Глава 5

Элементы управления и ClassWizard: реальные диалоговые окна

Подобно тому как словарный запас определяет возможность общения, так и словарь элементов управления MFC является ключевым фактором при создании универсальных и удобных в применении программ Windows. В этой главе значительно расширяется ваш багаж знаний о наиболее употребимых элементах управления MFC.

Возможно, вы видели типичный сюжет из фильма, когда одинокий парень или девушка приходит каждый вечер домой, в пустую квартиру, и без особой надежды нажимает кнопку автоответчика только для того, чтобы услышать, как механический голос повторяет снова и снова: "Для вас не поступало никаких сообщений. Для прослушивания ваших сообщений нажмите 1. Для записи нового приветствия..."

На наш взгляд, приложение FourUp чрезвычайно напоминает такой автоответчик. Оно имеет объект приложения, в его распоряжении находится объект главного окна и даже некоторые элементы управления. Приложение не функционирует само по себе. Для того чтобы приложение FourUp заработало, необходимо наличие некоторых сообщений.

В этой главе рассматривается другое инструментальное средство Visual C++, упрощающие разработку приложений Windows, — ClassWizard. ClassWizard помогает создавать классы в приложениях и управлять ими. Однако этим не исчерпывается та особая роль, которую играет ClassWizard в приложении MFC. ClassWizard действует как своего рода универсальный "антрепренер событий", осуществляя "связку" между элементами управления Windows, событиями и обрабатывающим их кодом.

С целью завершения обзора FourUp в этой главе будет уделено внимание следующей тематике:

- Сначала повторно рассмотрим приложение Dialog Editor из главы 3. Вы получите массу новых сведений относительно разработки главного диалогового окна FourUp.
- Затем будет показано, каким образом использовать класс MFC **CButton**. Вы уже знаете, как добавлять кнопки в приложение. Сейчас же за счет добавления методов в классы диалоговых окон вы заставите кнопки выполнять действия.
- Будет объяснено, каким образом использовать ClassWizard в процессе создания *управляющих переменных*: элементы данных C++ действуют в качестве заместителей объектов пользовательского интерфейса типа элементов управления **CEdit** и **CButtons**.
- Затем будет продемонстрировано применение одного из наиболее полезных универсальных методов Windows — функции **MessageBox()** — во всем ее разнообразии.

Планирование деятельности

Прежде чем перейти непосредственно к рассмотрению приложения Dialog Editor, рассмотрим несколько моментов, связанных с планированием. Хотелось бы уяснить, каким мы хотим видеть наше приложение и как оно должно работать. Несмотря на то что визуальная разработка пользовательского интерфейса, конечно же, являет собой серьезную альтернативу, однако, такой подход не может заменить этапа планирования. Планирование лучше всего производить на бумаге с карандашом в руках — даже если при этом используется обратная сторона старого конверта или бумажной салфетки. В конце концов, проще пересмотреть план, нежели изменить завершенную программу.

В этой связи давайте сразу решим, какой вид должно иметь главное диалоговое окно FourUp. Уже принято решение относительно того, как будет проходить игра:

- В начале игры каждый участник получает фиксированное количество денег. По мере продолжения игры программа должна учитывать выигрыши или

проигрыши игрока. Это означает, что в окне FourUp потребуется место для отображения оставшегося у игрока количества денег.

- Каждый игрок будет иметь дело с четырьмя картами, поэтому программа должна иметь игровую область для отображения карт и кнопку, на которой игрок сможет нажимать, чтобы сделать новый ход. Также необходима кнопка, позволяющая участнику выходить из игры.
- Каждой комбинации карт соответствует свой коэффициент оплаты. Для того чтобы ознакомить игрока с величинами выплат, необходимо предусмотреть область, отображающую величины выплат.
- И в заключение неплохо бы заинтересовать будущих участников игры. Для решения такой задачи используется вся роскошная графика, которая была создана в главе 3. (Вы ведь и не думали, что мы собрались оставить ее в диалоговом окне About, не так ли?)

Все перечисленное выше необходимо для составления плана создания интерфейса пользователя. Будут запланированы необходимые функции и элементы данных, которые необходимы для написания кода. На данном этапе можно основываться на черновом эскизе, показанном на рис. 5.1.

Изучая рисунок, несложно заметить, что главное диалоговое окно FourUp разделяется на четыре области. Графика размещается в верхней части, в области заголовка. Ниже заголовка располагается область для игры, где FourUp использует четыре карты. По мере развития игры в игровой области также отображаются величины выигрыша и потерь. Ниже игровой области находится таблица выплат, которая является просто таблицей статического текста, где приведены размеры выплат для каждой возможной комбинации карт. И наконец, в нижней части окна находятся кнопки, используемые игроком для управления игрой. Текст на кнопке Deal информирует игроков о том, какой суммой они рискуют.

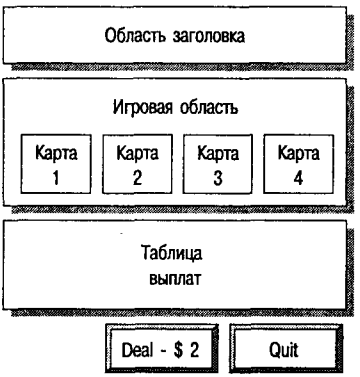


РИСУНОК 5.1 *Главное диалоговое окно FourUp.*

До этого момента у вас вряд ли появлялись неприятные ощущения при работе с Dialog Editor, поэтому мы довольно бегло рассмотрим вопросы, связанные с размещением диалогового окна. Вы готовы? Приступим.

Вернемся к Dialog Editor

Если рабочая область окна FourUp еще не открыта, то следует открыть ее сейчас, используя команду меню File | Open Workspace (Файл | Открыть рабочую область). (Заметим, что у нас нет оснований отвлекать ваше внимание, если вы хотите просто читать дальше. Но позже вас может посетить сожаление, поскольку навыки работы с инструментальными средствами типа Dialog Editor приобретаются только на практике. Поэтому включите компьютер и двигайтесь дальше.) Если вы уже загрузили проект FourUp, выберите окно ResourceView и дважды щелкните на **IDD_FOURUP_DIALOG** с тем, чтобы открыть Dialog Editor. Далее воспользуйтесь следующими шагами:

1. Удалите элемент управления статическим текстом, который утверждает: "TODO: Place Controls Here" (ЧТО СДЕЛАТЬ: Поместите элементы управления здесь). Сделать это легко. Потребуется выбрать элемент управления и нажать на клавишу Delete или комбинацию клавиш Ctrl+X.
2. Измените размеры главного диалогового окна таким образом, чтобы его ширина составляла 200 единиц, а высота — 175 единиц. Количественные значения размеров появляются в нижнем правом углу окна при изменении его размеров. Не забудьте, что эти единицы не являются пикселами. Они называются *Dialog Units (единицами измерения диалогового окна)*. Каждая единица приблизительно вдвое больше в высоту, чем в ширину, и ее фактический размер основан на шрифте, используемом в диалоговом окне. Для отслеживания этого эффекта обратите внимание на то, что главное диалоговое окно FourUp имеет лишь 175 единиц в высоту и 200 единиц в ширину. Однако диалоговое окно немного вытянуто по высоте, а не по ширине.
3. Включите Grid (Сетка), щелкнув на переключателе Toggle Grid (Переключить сетку), расположенном в панели инструментов Dialog.
4. Перетащите кнопки OK и Cancel в нижний правый угол и разместите их в требуемом порядке. Результаты выполненных действий показаны на рис. 5.2.

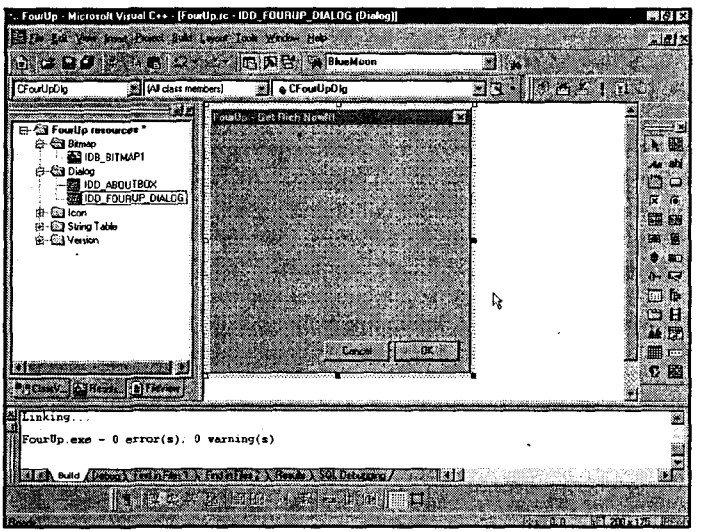


РИСУНОК 5.2
Главное диалоговое окно после изменения размеров и позиционирования элементов управления.

Добавление карт

Для отображения игровых карт используются элементы управления **CStatic** (так MFC называет элементы управления изображением). Первоначально каждый элемент управления **CStatic** заполняется пиктограммами игрового набора, загруженными ранее. Для этого потребуется выполнить следующие шаги:

1. Перетащите четыре элемента управления изображением с панели инструментов Control на главное диалоговое окно. Поместите их в горизонтальной строке недалеко от верхней части окна. (Не заботьтесь относительно интервалов или выравнивания. На самом деле, чем больше беспорядка — тем

лучше. Благодаря этому можно увидеть, каким образом работают инструменты Dialog Editor, выполняющие выравнивание и установку интервалов для компонентов.) Обратите внимание, что при установке каждого элемента управления в окне он выглядит как маленькая черная рамка.

2. Выберите первое окно изображения. (Необходимо щелкнуть на краях, поскольку щелчок посредине приводит к выбору диалогового окна.) Щелкните правой кнопкой мыши, чтобы открыть контекстное меню для первого окна изображения, и выберите пункт меню Properties (Свойства).
3. В диалоговом окне Picture Properties (Свойства изображения) установите комбинированный список Type (Тип) и измените характер выделения с Frame (Рамка) на Icon (Пиктограмма). После выполнения этой операции обратите внимание на изменение внешнего вида окна элемента изображения.
4. На данном этапе становится доступным список Image. Используйте его для выбора **IDI_HEART** из списка доступных пиктограмм. При закрытии диалогового окна Properties обратите внимание, что пиктограмма масти червей замещает рамку в окне FourUp.
5. Измените значение поля ID с **IDC_STATIC** на **IDC_CARD1**. Внутри вашей программы этот идентификатор используется для получения доступа к этому конкретному элементу управления **CStatic**.

По завершению работы с **IDC_CARD1** выполните рассмотренную последовательность операций для каждой из оставшихся пиктограмм: **IDI_CLUB**, **IDI_DIAMOND** и **IDI_SPADE**. В конечном итоге ваше диалоговое окно должно походить на окно, показанное на рис. 5.3.

ПРИМЕЧАНИЕ

Убедитесь в том, что каждому элементу управления присвоен корректный уникальный идентификатор (ID). Присвойте элементу управления с IDI_CLUB идентификатор IDC_CARD2, элементу управления с IDI_DIAMOND — идентификатор IDC_CARD3, элементу управления с IDI_SPADE — идентификатор IDC_CARD4.

Размещение карт

Теперь необходимо разместить карты. Для этого воспользуйтесь любезностью и талантами панели инструментов Dialog.

Выполните следующее:

1. Сначала разместите "крайние" карты. Выберите и переместите **IDC_CARD1** (Черви) и **IDC_CARD4** (Пики), соответственно, в позиции слева и справа в диалоговом окне. Не волнуйтесь относительно интервала между картами или их выравнивания — просто убедитесь в том, что имеется достаточно места между **IDC_CARD1** и **IDC_CARD4** для размещение двух других карт.
2. Выберите все четыре карты. Помните, как это делается? Щелкните на первой карте, после чего, удерживая клавишу Control (или Shift), щелкните на трех других элементах управления. Последняя выбранная карта становится доминирующей — не забывайте, что ее можно распознать по активным маркерам изменения размеров.

3. Выключите отображение сетки так, чтобы элементы управления могли перемещаться в любое место внутри диалогового окна, а не просто по узлам сетки. Для упорядочения карт используйте кнопку Space Across, расположенную в панели инструментов Dialog, как показано на рис. 5.4. (Можно также нажать комбинацию клавиш Alt + стрелка вправо или выбрать команду Layout | Space Evenly | Across (Размещение | Равномерный интервал | Поперек) из главного меню.) Посмотрите на рис. 5.5 и отметьте, насколько равномерно элементы управления располагаются между **IDC_CARD1** и **IDC_CARD4**.

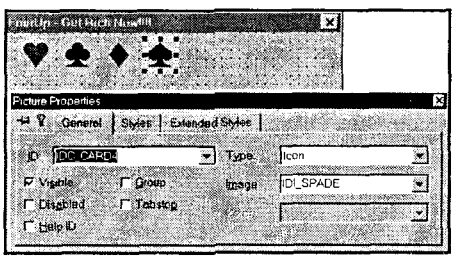


РИСУНОК 5.3 *Добавление пиктограмм к элементам управления изображением.*

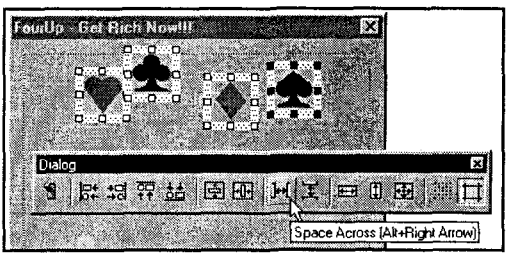


РИСУНОК 5.4 *Выравнивание интервалов между элементами управления при помощи панели инструментов Dialog.*

4. Теперь, когда карты равномерно размещены, их можно выровнять по верхней и нижней границах. Поскольку элементы управления имеют одинаковый размер, не имеет значения, используете ли вы команду Align With Bottom (Выровнять по нижней границе) или Align With Top (Выровнять по верхней границе). На рис. 5.5 видно, что выбрана команда Align With Bottom. (Тот же эффект можно получить, нажав комбинацию клавиш Ctrl + стрелка вниз или команду Layout | Align | Bottom (Размещение | Выровнять | По нижней границе) из главного меню.) Эффект выравнивания карт по нижнему краю показан на рис. 5.6.

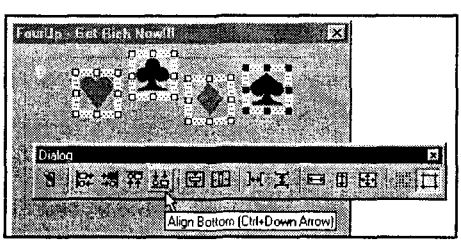


РИСУНОК 5.5 *Выравнивание группы элементов управления с использованием панели инструментов Dialog.*

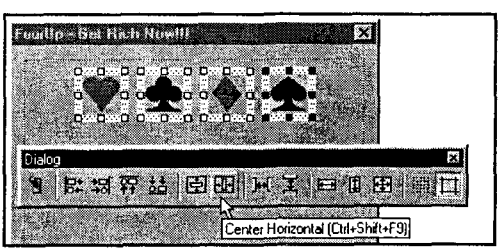


РИСУНОК 5.6 *Центрирование группы элементов управления внутри главного окна при помощи панели инструментов Dialog.*

5. Отцентрируйте группу элементов управления относительно горизонтали на игровой поверхности. Это несложно — просто используйте кнопку Center Horizontal, которая применялась для центрирования отдельной кнопки в главе 3 (см. рис. 5.6).

Добавление игровой области

Как уже было показано в нашем эскизе, карты упорядочиваются в игровой области, представленной группой элементов управления. Вспомните, что когда вы работали с групповым элементом управления в главе 3, у вас появлялась красивая кривая линия, окружающая окно About. Для отображения такой линии пришлось уничтожить текст, который появляется по умолчанию вверху группы.

Однако, в случае с игровой областью этому тексту можно найти хорошее применение. Он может выступать в качестве области отображения, информирующей игроков о том, сколько денег у них имеется. Для выполнения этого следуйте приведенным ниже шагам:

1. Перетащите групповой элемент управления из панели инструментов Control и разместите его таким образом, чтобы равномерно окружить набор карт.
2. Откройте диалоговое окно Properties для группы. Измените идентификаторы элементов управления на **IDC_AMT_LEFT** и заголовок на "Оставшаяся сумма $ 100.00".
3. Теперь давайте переместим игровую область приблизительно в позицию, указанную на рис. 5.1. Оставьте место для добавления заголовка и пиктограммы в верхней части диалогового окна. Для перемещения карт как группы сначала выберите их, создав прямоугольник выделения (при помощи мыши), который полностью включает группу **IDC_AMT_LEFT**. (Таким образом одновременно выбираются все карты и группа.) Если только выбирались элементы управления, то, удерживая клавишу Shift, щелкните на группе, чтобы сделать ее доминирующим элементом управления. Затем перемещайте элементы управления вниз, пока индикатор позиции не отобразит цифры 10, 50. Последнего гораздо проще достигнуть скорее при помощи клавиш управления курсором, нежели при помощи мыши.

Добавление области заголовка

Теперь, когда создана игровая область, можно добавлять заголовок. Можно просто сделать копию пиктограммы и эмблемы FourUp из диалогового окна About и упорядочить их в верхней части окна. Для выполнения этого придерживайтесь следующих шагов:

1. Получите доступ к диалоговому окну About, дважды щелкнув на **IDD_ABOUTBOX** в окне ResourceView.
2. Выберите эмблему и пиктограмму FourUp, сначала щелкнув на первом элементе, а затем и на втором, удерживая клавишу Shift. Перед переходом к следующему шагу оба элемента должны быть выбраны.
3. Скопируйте оба элемента в буфер обмена Windows, используя комбинацию клавиш Ctrl+C или команду Edit | Copy (Правка | Копировать) из главного меню.
4. Переключитесь обратно на главное диалоговое окно, дважды щелкнув на **IDD_FOURUP** в окне Resource View.
5. Нажмите комбинацию клавиш Ctrl+V либо выберите команду Edit | Paste (Правка | Вставить) из главного меню для размещения эмблемы и пиктограммы в диалоговом окне **IDD_FOURUP**. Разместите эти элементы в желаемом порядке.

Создание таблицы выплат

Под игровой областью размещается другая группа, отображающая ожидаемые выигрыши, которые получат игроки в случае выпадения двух пар, трех карт одной масти или четырех карт одной масти. (Если игроку выпадает только одна пара, никакого выигрыша он не получает.) Выполните следующие шаги:

1. Перетащите еще один групповой элемент управления из панели инструментов Control и разместите его под игровой областью. Заголовок группы должен выглядеть как "Выплаты".
2. Перетащите три метки со статическим текстом из панели инструментов Control и разместите их в группе Payouts (Выплаты). В три метки поместите следующий текст:

   "2 пары дают 3.00"

   "3 карты одной масти дают 6.00"

   "4 карты одной масти дают 9.00"
3. Используйте несколько управляющих последовательностей (\t), соответствующих символу табуляции, для позиционирования величины выплаты справа от текста описания. Это позволит выровнять суммы выплат без использования пробелов — задача, обреченная на неудачу из-за таких мелочей, как переменная ширина символов. Переместите диалоговое окно Properties в сторону таким образом, чтобы можно было видеть диалоговое окно FourUp. Это нужно, чтобы наблюдать за результатом добавления управляющих последовательностей. Получившийся в итоге результат показан на рис. 5.7.

Надписи на кнопках

К этому моменту времени интерфейс практически завершен. Единственное, что остается сделать, — это изменить надписи на двух кнопках. Для этого необходимо проделать следующее:

1. Замените заголовок кнопки OK на "Deal — $ 2.00". (Выберите пункт Properties (Свойства) из контекстного меню, доступного по щелчку правой кнопки мыши, точно так же, как это делается для других элементов управления.)
2. Прежде чем закрыть диалоговое окно Properties для кнопки OK, измените идентификатор кнопки с **IDOK** на **IDC_DEALCARDS**.
3. Откройте диалоговое окно Properties для кнопки Cancel. Измените ее заголовок на "Cash Out", но оставьте идентификатор **IDCANCEL**.
4. В заключение измените размещение двух кнопок так, чтобы кнопка Deal находилась в центре, а кнопка Cash Out — справа.
5. Отцентрируйте кнопки, используя инструмент Center Horizontal.

Хорошо, вы сделали это. Вами создан пользовательский интерфейс, и это было реализовано намного быстрее, чем формирование диалогового окна About. Гордитесь собой! Скомпилируйте программу и испытайте ее в действии. Вы увидите, что программа выполняется, и можно открыть диалоговое окно About как и прежде, однако больше ничего не происходит. Результат проделанной работы показан на рис. 5.8.

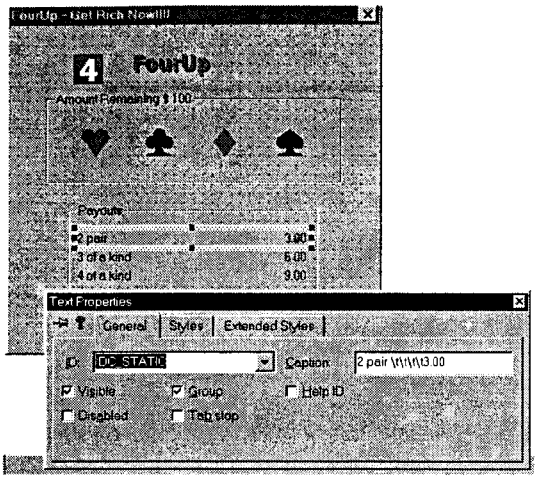


РИСУНОК 5.7 *Добавление текста в группу Payouts (Выплаты).*

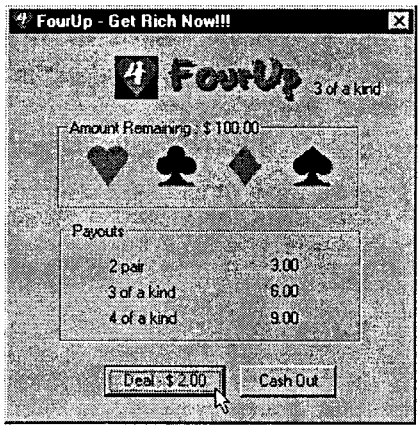


РИСУНОК 5.8 *Работающее приложение FourUp.*

Создание кода

Пользовательский интерфейс для FourUp завершен. Теперь необходимо создать код, который реализует ранее описанную логику игры. При этом нужно исходить из оценки, насколько мал ваш код по объему и насколько хорошо он удовлетворяет спецификации.

На самом деле мы не оставим вас в затруднительном положении после того, как был проделан столь большой объем работы. Написание кода для приложения является, вероятно, наиболее простой вещью благодаря ClassWizard, с которым вы вскоре встретитесь.

Но сначала, как это было с интерфейсом пользователя, необходимо потратить несколько минут на планирование стратегии. Во время разработки пользовательского интерфейса в качестве справки использовались несколько грубых эскизов. Однако когда вы планируете работу программы, нет ничего лучше простого подчеркивания ключевых моментов. На сей раз заголовки окажутся *событиями*, а пункты нижнего уровня — ответами на них.

Планирование событий

Давайте вначале зададим вопрос: "Что произойдет, если игрок выполнит действие X?". Ввиду того что X в этом случае может быть только одним из двух возможных событий, список получается не слишком длинным:

- Что происходит, когда игрок *щелкает на кнопке Cash Out*?

  1. Поблагодарить игрока и отобразить оставшуюся сумму.

- Что происходит, когда игрок *щелкает на кнопке Deal*?

  1. Выполните проверку, что игрок имеет достаточное количество денег для игры. Если это не так, игрока следует уведомить о том, что он должен раздобыть где-нибудь большее количество денег. (Задачу добавления средств на счет игрока мы оставляем в качестве упражнения.)

2. Если игрок имеет, по крайней мере, 2$, он может сделать ставку.
3. Случайным образом выберите набор для каждой из 4 карт.
4. Обновите изображение каждой карты для указания подходящей.
5. Вычислите, выиграл ли игрок, и если так, добавьте выигрыш на счет игрока.

Теперь отвлекитесь на момент и спросите себя:

- В каких элементах данных я буду нуждаться?
- В каких методах я буду нуждаться?

Написание кода вручную: добавление элемента данных

Дальнейшее краткое рассмотрение должно убедить в том, что если не случится ничего непредвиденного, то нет необходимости отслеживать количество денег, оставляемых игроком. Для реализации этого потребуется создать "старый добрый" элемент данных — нет оснований выполнять нечто экстраординарное, за исключением определения класса, в котором все будет размещаться.

После изучения структуры приложения FourUp в главе 4 вы знаете, что доступны два выбора: класс приложения (**CFourUp**) или класс диалогового окна (**CFourUpDlg**). Поскольку обычно происходит взаимодействие с компонентами, которые являются (или будут являться) непосредственной частью диалогового окна (и таким образом, класса **CFourUpDlg**), давайте поместим новый элемент данных именно там.

Ниже приведены шаги, которым нужно следовать при определении переменной внутри заголовка для **CFourUpDlg**:

1. Выберите панель ClassView в окне Project Workspace.
2. Найдите класс **CFourUpDlg** и дважды щелкните на имени класса. (Убедитесь в том, что вы дважды щелкаете на имени класса, а не на имени конструктора или одного из методов. Двойной щелчок на имени функции приводит к генерированию файла реализации. Двойной щелчок на имени класса или на имени элемента данных приводит к объявлению класса в файле заголовков.)
3. Теперь добавьте в определение класса раздел **private** (приватный). Определите элемент данных **m_Amt_Remaining** типа **double**. Поле, добавляемое в определение класса, можно видеть на рис. 5.9. Обратите внимание, что элемент данных появляется в панели ClassView непосредственно после его ввода.

Далее необходимо инициализировать поле **m_Amt_Remaining**. Это можно выполнить в конструкторе класса. Просто дважды щелкните на имени конструктора в панели ClassView для перехода к файлу CPP, в котором определен конструктор. Инициализируйте элемент данных **m_Amt_Remaining**, добавляя выделенный код, показанный в листинге 5.1.

Листинг 5.1 Инициализация элемента данных m_Amt_Remaining в конструкторе CFourUpDlg.

```
////////////////////////////////////////////////////////////
// Диалоговое окно CFourUpDlg

CFourUpDlg::CFourUpDlg(CWnd* pParent /*=NULL*/)
    : CDialog(CFourUpDlg::IDD, pParent)
```

```
{
    //{{AFX_DATA_INIT(CFourUpDlg)
    //}}AFX_DATA_INIT
    m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
    // Инициализация поля m_Amt_Remaining
    m_Amt_Remaining = 100.0;
}
```

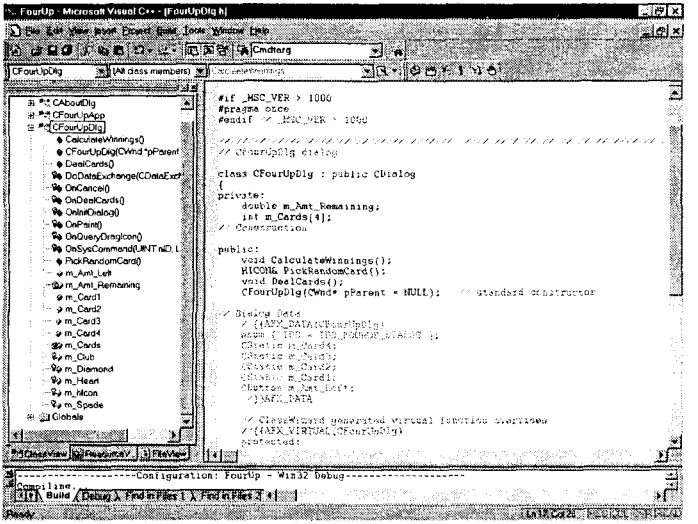


РИСУНОК 5.9
Добавление приватного элемента данных в класс CFourUpDlg.

Ответ на события BN_CLICKED

Иногда программирование в среде Windows вводит в заблуждение: вы удивляетесь, почему что-либо происходит определенным образом. Обработка событий кнопки является одним из приятных исключений: абсолютно очевидно, что когда пользователь осуществляет щелчок на кнопке, при этом происходит событие. В таком случае в Windows вызывается событие **BN_CLICKED**. При этом хорошо то, что работа с кнопками настолько проста, что не нужно углубляться в суть происходящего. ClassWizard знает, *что* вы обычно желаете получить от функции обработчика события щелчка на кнопке. Чтобы увидеть, как выполняется ответ на это событие, давайте вернемся к Dialog Editor.

Генерация метода OnCancel()

Мы уже приняли решение о том, что должно случиться, когда игрок щелкает на кнопке Cash Out — это не представляет сложности для понимания. Однако это похоже на то, что происходит, когда игрок щелкает на кнопке Deal. Поэтому давайте сначала рассмотрим обработку события, представляющего собой щелчок на кнопке Cash Out.

Для выполнения данной задачи потребуется использовать ClassWizard для генерации метода, который вызывается всякий раз, когда игрок нажимает на кнопке Cash Out. Для этого необходимо выполнить следующее:

1. Вернитесь в ResourceView.
2. Дважды щелкните на **IDD_FOURUP_DIALOG** для открытия Dialog Editor.
3. Дважды щелкните на кнопке Cash Out.

Как только вы сделаете это, откроется диалоговое окно Add Member Function (как показано на рис. 5.10). При этом предполагается, что добавляется функция **OnCancel**.

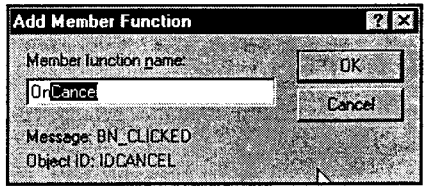


РИСУНОК 5.10 *Диалоговое окно Add Member Function.*

Если близко познакомиться с диалоговым окном, несложно заметить, что Windows генерирует сообщение **BN_CLICKED**, означающее, что на *кнопке был произведен щелчок*. Необходимо также обратить внимание на то, что идентификатором кнопки все еще остается **IDCANCEL**, поскольку *имя* кнопки Cancel не изменялось, изменялся только заголовок, отображаемый на кнопке.

У вас мог бы возникнуть вопрос, почему имя кнопки OK изменилось с **IDOK** на **IDC_DEALCARDS**, *но подобное изменение не выполнялось для кнопки* Cancel. Причина этого заключается в том, что в Visual C++ кнопки OK и CANCEL обрабатываются специальным образом. В большинстве случаев, когда Visual C++ работает с именем, последнее можно изменить безнаказанно. Однако если ***OnCancel*** изменяется на ***OnCashOut***, щелчок на кнопке Cash Out больше не приводит к выходу из приложения. Оставьте имя **OnCancel**, чтобы была доступна заданная по умолчанию обработка выхода.

Сразу же после щелчка на кнопке OK в диалоговом окне Add Member Function Visual C++ записывает новую пустую функцию с соответствующим прототипом в заголовке класса. Затем открывается редактор кода, где можно создать код, необходимый для обработки события. В листинге 5.2 показан код, созданный при помощи ClassWizard.

Листинг 5.2 Функция OnCancel(), сгенерированная ClassWizard.

```
void CFourUpDlg::OnCancel()
{
      // ЧТО СДЕЛАТЬ: Добавьте здесь дополнительный код обработки
      CDialog::OnCancel();
}
```

Создание кода: завершение определения функции OnCancel()

Если игрок щелкает на кнопке Cash Out, то согласно нашему плану ему должна быть вынесена благодарность за игру и передано сообщение о количестве оставшихся денег. Так каким же образом вы будете благодарить игрока за использование вашей программы?

Хорошо, что Windows поставляется с полным набором функций **MessageBox()**, разработанных для упрощения реализации подобных действий. Функция **MessageBox()** существует в виде нескольких версий, одна из которых — элемент функции класса **CWnd**. К счастью, используемый класс **CFourUpDlg** — это прямой

потомок **CWnd**, так что при этом функция **MessageBox()** может вызываться без проблем. Если вы находитесь в другом классе, который не наследуется от **CWnd()**, то будете вызывать универсальную версию **MessageBox()**, носящую наименование **AfxMessageBox()**.

Функция **MessageBox()** имеет три параметра:

- Отображаемое сообщение
- Заголовок для поля сообщения всплывающего окна
- Целое число, определяющее вид кнопок, отображаемых в окне сообщения

Необходимо передать только первый параметр, поскольку остальные два — это заданные значения по умолчанию. Для метода **OnCancel()** передаются первых два параметра.

Преобразование данных m_Amt_Remaining

Функции **OnCancel()** присуща одна сложность: желательно отображать количество денег для остающегося игрока, равно как и дружеское сообщение. Опытные программисты на языке C, вероятно, использовали бы библиотечную функцию **sprintf()** из стандартной библиотеки C:

```
char buffer[80];
sprintf(buffer, "Хорошая игра. У вас осталось $ %.2",
   m_Amt_Remaining);
```

Этот код будет прекрасно выполняться, но необходимо, чтобы с функцией **sprintf()** был связан код из стандартной библиотеки C, который не является автоматически доступным для программ MFC. Если вы ранее занимались программированием Windows API, то знаете, что Windows уже включает подобную функцию — **wsprintf()**. К сожалению, **wsprintf()** не является адекватной для этой специфической задачи, поскольку она не обрабатывает форматирование с плавающей точкой.

К счастью, MFC обеспечивает лучший способ решения этой проблемы: класс **CString**. **CStrings** — это динамические строки переменной длины, которые можно использовать посредством MFC везде, где предыдущие версии Windows требуют наличия строки C-стиля, завершающейся нулем. Используя класс **CString**, нет необходимости волноваться относительно того, насколь большим должен быть создаваемый буфер — просто создается объект **CString** и используется метод **Format**. **CString** получает размер непосредственно при обработке всего распределения памяти и освобождения памяти за сценой.

При использовании класса **CString** и метода **MessageBox()** можно завершить определение метода **OnCancel()** путем добавления выделенных строк, показанных в листинге 5.3. Метод **OnCancel()** можно видеть в работе на рис. 5.11.

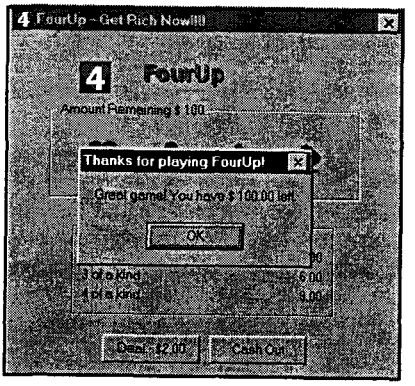


РИСУНОК 5.11 *Тестирование метода OnCancel().*

Листинг 5.3 Завершенная функция OnCancel().

```
void CFourUpDlg::OnCancel()
{
      // ЧТО СДЕЛАТЬ: Добавьте здесь дополнительный код обработки
      CString s;
      s.Format("Отличная игра! У вас осталось $ %.2f.",
  m_Amt_Remaining);
      MessageBox(s, "Благодарим за игру в FourUp!");

      CDialog::OnCancel();
}
```

Создание функции OnDealCards()

Прекрасно, обработка щелчка на кнопке Cash Out не была слишком сложной, не так ли? Давайте посмотрим, является ли код обработки щелчка на кнопке Deal столь же простым. Начните с возврата к Dialog Editor и дважды щелкните на кнопке Deal.

Так же как и ранее, появится диалоговое окно Add Member Function. По-прежнему отображается сообщение Windows **BN_CLICKED**, но на этот раз именем функции будет **OnDealcards()**, поскольку имя кнопки было изменено с **IDOK** на **IDC_DEALCARDS**. Если оставить имя **IDOK**, то ClassWizard создаст стандартный код закрытия диалогового окна, т.е. каждый раз, когда игрок запрашивает новую сдачу, игра бы закрывалась! Определенно это не то, что должно происходить, если вы собираетесь получить все деньги с игрока..., особенно, если нужно дать игрокам настоящий опыт деловой игры.

Для того чтобы не вступать в противоречие с вашими соглашениями по именованию, измените имя функции на **OnDealCards()** (обратите внимание на заглавную букву С), а затем щелкните на кнопке OK. Теперь вы готовы сдавать карты!

OnDealCards(): первый черновик

Одно из первых правил программирования гласит: "Делайте изначально простые вещи!". Если же вы не придерживаетесь этого правила, то рискуете увязнуть там, где этого можно было избежать.

Так что же в этом случае является очевидной вещью? Очевидно, что необходимо выполнить следующее:

- Вычитать $2 всякий раз, когда игрок щелкает на кнопке Deal.
- Восстановить изображение общей суммы игрока после сдачи карт и вычисления выигрыша.

В листинге 5.4 проиллюстрированы описанные действия.

Листинг 5.4 Первый черновик реализации функции OnDealCards().

```
void CFourUpDlg::OnDealCards()
{
      // 1. Вычесть 2.00, что бы ни случилось
      m_Amt_Remaining -= 2.00;

      // 2. Сдать карты. Как это делается, пока не известно.
```

```
    // 3. Вычислить выигрыш, который затем добавить к m_Amt_Remaining

    // 4. Отобразить итог m_Amt_Remaining
    CString s;
    s.Format("Остающаяся сумма $ %.2f", m_Amt_Remaining);
}
```

Как видно, вы уже удостоверились в том, что деньги получены, следовательно шаг 1 пройден. Шаги 2 и 3 кажутся немного трудными, поэтому давайте сейчас упростим алгоритм и предположим, что игроки всегда теряют часть суммы. Это означает, что необходимо просто отобразить остаток общей суммы. Как было показано ранее, сообщение уже форматируется: единственно, что необходимо — это отобразить все на экране.

Вместо использования **MessageBox()**, как это имело место, когда игрок производил выплату, на этот раз отображается общая сумма, остающаяся в заголовке в верхней части группы (под именем **IDC_AMT_LEFT**), которая окружает игровую область. Для этого потребуется использование *управляющих переменных.*

Введение в ClassWizard и управляющие переменные

Если вы еще не знаете об этом, имейте в виду, что объекты группы — это окна (унаследованные от **CWnd**), практически такие же, как и все остальные окна в Windows. Как и для других окон, можно изменять текст, используемый для заголовка группы, путем вызова метода **SetWindowText()**.

Это звучит просто, не правда ли? В функции **OnDealCards()** вызывается **SetWindowText()**, в которую передается отформатированная строка **s** типа **CString**. (Обратите внимание на текст после шага 4 в листинге 5.4).

К сожалению, как вы, вероятно, уже догадались, все не так просто. **SetWindowText()** — это **метод**, вызываемый за счет использования объекта, производного от **CWnd**. Проблема заключается в том, что **IDC_AMT_LEFT** не является на самом деле объектом группы. Вместо этого справедливо то, что как и все идентификаторы, присваиваемые Dialog Editor, **IDC_AMT_LEFT** ссылается на данные, используемые Windows для создания объекта группы во время выполнения.

Перед отправкой сообщения **SetWindowText()** группе, представляемой **IDC_AMT_LEFT**, потребуется получить ссылку на действительный объект группы, созданный Windows, а затем послать сообщение этому объекту. Для достижения такой цели используется ClassWizard. При этом создается *управляющая переменная (control variable)*, которая поддерживает ссылку на элемент управления, создаваемый во время выполнения. Для этого необходимо проделать следующее:

1. Откройте главное диалоговое окно в Dialog Editor. Выделите группу **IDC_AMT_LEFT** и щелкните правой кнопкой мыши на ней для отображения контекстного меню. Выберите из контекстного меню ClassWizard, как показано на рис. 5.12.
2. ClassWizard отображает диалоговое окно с закладками. Выберите закладку Member Variables. Появившееся окно походит на окно, показанное на рис. 5.13.

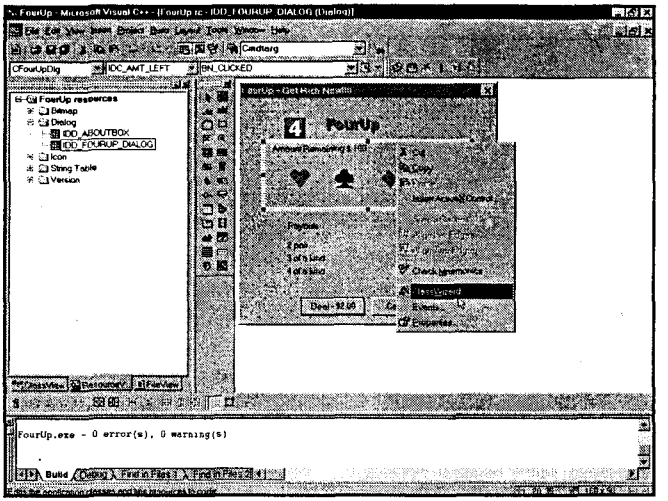


РИСУНОК 5.12
Запуск ClassWizard для группы IDC_AMT_LEFT.

3. В списке идентификаторов элементов управления найдите и выделите **IDC_AMT_LEFT**. Затем щелкните на кнопке Add Variable (Добавить переменную), как показано на рис. 5.13.

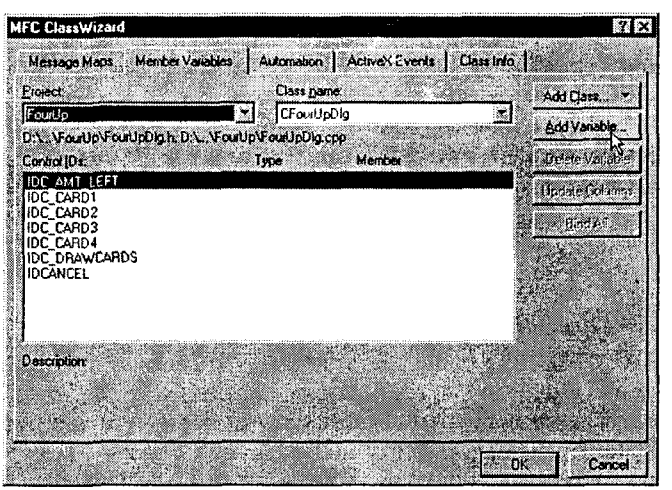


РИСУНОК 5.13
Добавление нового элемента данных для получения IDC_AMT_LEFT.

4. Когда откроется диалоговое окно Add Member Variable, введите значение "m_Amt_Left" в поле Member Variable Name (Имя элемента данных). Затем выберите пункт Control (Элемент управления) из раскрывающегося списка Category (Категория), и далее — **CButton** из раскрывающегося списка Variable Type (Тип переменной), как показано на рис. 5.14.
5. Добавьте следующую строку в конец описания функции **OnDealCards()**:

```
m_Amt_Left.SetWindowText(s);
```

6. Откомпилируйте программу и запустите ее на выполнение. При этом можно увидеть, что всякий раз, когда игрок щелкает на кнопке Deal, FourUp уменьшает значение переменной **m_Amt_Remaining** на $2 и отображает новый результат в окружении группы для игровой области.

Создание кода: сдача карт и подсчет общего выигрыша

Вы приблизились к цели. Теперь все, что необходимо сделать — это завершить шаги 3 и 4 в функции **OnDealCards()**. В настоящий момент, когда вы знаете, как использовать ClassWizard для создания управляющих переменных, оставшаяся часть кода не будет слишком сложной (даже при условии существенного объема этой части кода).

Создание переменных, управляющих картами

Приступим к созданию управляющих переменных для каждой из карт, с **IDC_CARD1** по **IDC_CARD4**. Можно было бы по очереди выбрать каждый элемент управления и затем открыть ClassWizard при помощи контекстного меню, как это делалось ранее. ClassWizard можно также открывать, используя комбинацию клавиш Ctrl+W или выбирая ClassWizard из меню View. Давайте проделаем это:

1. Откройте ClassWizard, воспользовавшись меню View.
2. Добавьте переменные для **IDC_CARD1**, **IDC_CARD2**, **IDC_CARD3** и **IDC_CARD4**, именуя их, соответственно, **m_Card1**, **m_Card2**, **m_Card3** и **m_Card4**. Убедитесь в том, что поле **Category** во всех случаях установлено в значение Control, а поле Variable Type — в **CString**. ClassWizard сбрасывает значения по умолчанию к Value и **CString**, которые являются некорректными для этого приложения.

Просмотрите на рис. 5.15, проверив корректность определений переменных.

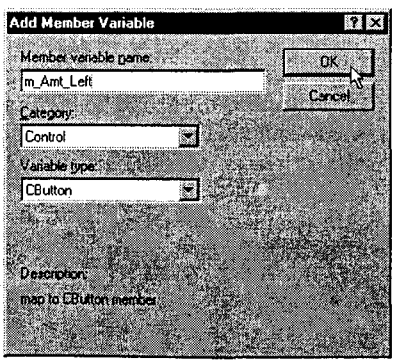


РИСУНОК 5.14 *Использование диалогового окна Add Member Variable.*

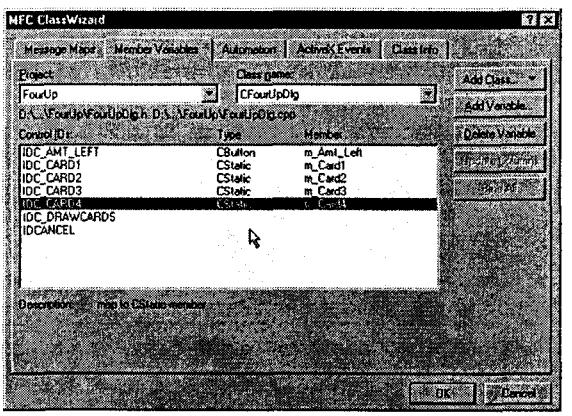


РИСУНОК 5.15 *Добавление переменной управления к*

Создание переменных пиктограмм

В дополнение к переменным управления для каждой карты необходимы также переменные для каждого изображения карты. Поскольку уже были добавлены четыре пиктограммы (от **IDI_HEART** до **IDI_SPADE**), можно ожидать их появления на странице Member Variables в ClassWizard.

Однако ничего подобного не происходит. ClassWizard обрабатывает только элементы управления; он не обрабатывает более прозаические ресурсы типа пиктограмм и растровых изображений. Это необходимо делать вручную. К счастью, нет необходимости создавать все с нуля. AppWizard создает код для обработки оригинальной пиктограммы, и все, что необходимо сделать — это найти упомянутый код и подражать действиям AppWizard.

Для реализации подобного рода технологии выполните следующие шаги:

1. Найдите переменную **m_hIcon** в окне ClassView. Эта переменная хранит оригинальную пиктограмму.
2. Дважды щелкните на **m_hIcon**. Перейдите к файлу определения для класса CFourUpDlg, в котором можно увидеть следующие строки кода:

```
// Реализация
protected:
      HICON m_hIcon;
```

3. Тип **HICON** означает **H**andle to an **ICON** (дескриптор пиктограммы). Необходимо иметь по одной переменной для каждой из ваших собственных пиктограмм, поэтому добавьте еще четыре переменных **HICON**. Назовите их **m_Heart**, **m_Club**, **m_Diamond** и **m_Spade**. Теперь код должен выглядеть следующим образом:

```
// Реализация
protected:
      HICON m_HICON;
      HICON m_Club;
      HICON m_Diamond;
      HICON m_Heart;
      HICON m_Spade;
```

4. Перед использованием элементы данных должны быть инициализированы. Давайте проследим за работой AppWizard и выясним, каким образом выполняется инициализация **m_hIcon**. Дважды щелкните на конструкторе класса **CFoupUpDlg()** в ClassView и найдите строку, в которой инициализируется **m_hIcon**. Создайте подобные строки для загрузки **IDI_HEART** и других пиктограмм. Результаты проведения необходимых изменений для **CFourUpDlg()** показаны в листинге 5.5.

Листинг 5.5 Инициализация дополнительных пиктограмм в CFourUpDlg().

```
CFourUpDlg::CFourUpDlg(CWnd* pParent /*=NULL*/)
    : CDialog(CFourUpDlg::IDD, pParent)
{
      //{{AFX_DATA_INIT(CFourUpDlg)
      //}}AFX_DATA_INIT
      // Обратите внимание, что в Win32 LoadIcon не нуждается в
      // последующем вызове DestroyIcon
```

```
    m_hIcon       = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
    m_Club        = AfxGetApp()->LoadIcon(IDI_CLUB);
    m_Diamond     = AfxGetApp()->LoadIcon(IDI_DIAMOND);
    m_Heart       = AfxGetApp()->LoadIcon(IDI_HEART);
    m_Spade       = AfxGetApp()->LoadIcon(IDI_SPADE);

    // Инициализация поля m_Amt_Remaining
    m_Amt_Remaining = 100.0;
}
```

Метод DealCards()

Если вы еще раз посмотрите на функцию обработки события щелчка на кнопке **OnDealCards()**, то увидите, что пока ничего не сделано для реализации промежуточных шагов 2 и 3. Давайте устраним этот недостаток, добавив функцию **DealCards()** для реализации шага 2 и функцию **CalculateWinnings()** — для реализации шага 3.

Сначала напишем код функции **DealCards()**. Для этого необходимо выполнить следующие шаги:

1. Выберите пункт Add Member Function из меню WizardBar. (Не забудьте, что WizardBar появляется в панели инструментов. Можно получить доступ к этому меню за счет выполнения щелчка на пиктограмме со стрелкой вниз, размещенной с правой стороны в панели инструментов.)
2. В открытом диалоговом окне Add Member Function (см. рис. 5.16) определите возвращаемый тип функции как **void**, а ее имя — как **DealCards**.
3. Щелкните на кнопке OK. На этом этапе Visual C++ откроет окно редактора кода, где уже присутствует каркасная функция. Для завершения функции **DealCards()** добавьте в нее код из листинга 5.6.

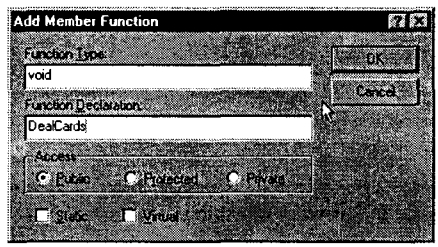


РИСУНОК 5.16 *Диалоговое окно Add Member Function.*

Листинг 5.6 Метод DealCards().

```
void CFourUpDlg::DealCards()
{
    // Инициализация массива m_Cards
    for(int i = 0; i < 4; i++)
        m_Cards[i] = 0;

    m_Card1.SetIcon(PickRandomCard());
    m_Card2.SetIcon(PickRandomCard());
    m_Card3.SetIcon(PickRandomCard());
    m_Card4.SetIcon(PickRandomCard());
}
```

Краткое объяснение работы метода DealCards()

Метод **DealCards()** выполняет две задачи. Во-первых, для подсчета очков необходимо отслеживать карты, находящиеся в игре. Поскольку имеются четыре

масти, превосходно подходит массив из четырех целых чисел. Первое целое число в массиве представляет количество треф, второе число — количество бубен и т.д. Однако каждый раз, когда FourUp сбрасывает карты, значения массива необходимо обнулить. Эту задачу реализует первый цикл в функции **DealCards()**.

После этого отображается произвольная карта в каждой из управляющих переменных карт, от **m_Card1** до **m_Card4**. Каждая управляющая переменная — это элемент класса **CStatic**, который имеет метод **SetIcon()**. Функция **SetIcon()** принимает в качестве параметра **HICON**. К счастью, уже имеются четыре **HICON** в виде полей **m_Club**, **m_Diamond** и т.д. Потребуется лишь создать вспомогательную функцию, которая будет выбирать случайным образом одну из переменных и возвращать ссылку на нее.

Завершение метода DealCards()

Как уже объяснялось ранее, для завершения функции **DealCards()** создается массив, используемый для подсчета очков, и вспомогательная функция **PickRandomCard()**, которая осуществляет фактический выбор. Для реализации этих положений на практике выполните следующие шаги:

1. Добавьте массив **int** с именем **m_Cards[4]** к определению класса **CFourUpDlg** там, где был определен элемент данных **m_Amt_Remaining**.
2. Добавьте код, показанный в листинге 5.7, который реализует вспомогательную функцию **PickRandomCard()**, отвечающую за выбор произвольной карты. (Не забудьте для добавления функции использовать команду Add Member Function из меню WizardBar. Если вы добавляете код вручную, потребуется также изменить файл заголовка класса. Если используется WizardBar, он полностью выполняет данную задачу.) Обратите внимание, что функция возвращает ссылку на **HICON**, т.е. на дескриптор для пиктограммы.

Листинг 5.7 Метод PickRandomCard().

```
HICON& CFourUpDlg::PickRandomCard()
{
      int num = (rand() % 4);
      m_Cards[num] ++; // Обновление массива подсчета очков
      switch(num)
      {
            case 0: return m_Club;
            case 1: return m_Diamond;
            case 2: return m_Heart;
      }
      return m_Spade;
}
```

Произвольное отклонение

Поскольку функция **PickRandomCard()** использует библиотечную функцию **rand()** для генерации псевдослучайных чисел, убедитесь в наличии начального значения функции. При этом должна генерироваться различная последовательность чисел при каждом выполнении приложения. Это можно сделать, добавив в конструктор класса **CFourUpDlg** следующие строки:

```
// Присвоить начальное значение генератору случайных чисел
srand( (unsigned) time(NULL));
```

Метод CalculateWinnings()

Последний модуль программы вычисляет выигрыш игрока. Давайте посмотрим, как он реализован.

Всякий раз когда вызывается **DealCards()**, массив **m_Cards** очищается за счет обнуления четырех его значений. Затем четыре раза вызывается функция **PickRandomCard()**. Каждый раз функция **PickRandomCard()** генерирует случайное число в диапазоне от 0 до 3 с использованием оператора взятия модуля (%). Это случайное число регулирует отображение масти карты при помощи оператора **switch**. Однако, прежде чем FourUp отображает карту, случайное число используется для обновления "очков" для текущей масти при помощи строк

```
int num = (rand() % 4);
m_Cards[num] ++; //Обновление массива подсчета очков
```

Таким образом, после того как все четыре обращения к функции **PickRandomCard()** завершены, каждый из четырех элементов в **m_Cards** имеет значения, заключенные между 0 и 4. Все, что нужно сделать — это пройти в цикле по всем элементам, а затем вычислить выигрыш, основанный на найденных значениях.

Так как же вычисляется выигрыш? Для этого используется следующая логика:

- Если в каком-либо элементе находится значение 4, это значит, что FourUp выполнила сдачу четырех карт одной масти — можно немедленно зачислять выигрыш.
- Аналогично, если вы сталкиваетесь со значением 3 в любом элементе, это значит, что FourUp выполнила сдачу трех карт одной масти, и можно сразу же зачислять выигрыш.
- Если имеются значения 0 или 1, то это не влияет на число очков. (Упомянутое значение можно игнорировать.)
- Если имеется значение 2, необходимо выяснить, является ли это первая пара либо вторая. Если это вторая пара, можно вычислять выигрыш. Если первая пара, этот факт следует отметить, и продолжить проверку, чтобы увидеть, имеется ли другая пара. Последнее достигается за счет использования локальной переменной **pairs**.

Код функции **CalculateWinnings()** показан в листинге 5.8. При его вводе не забудьте воспользоваться WizardBar.

Листинг 5.8 Метод CalculateWinnings().

```
void CFourUpDlg::CalculateWinnings()
{
      int pairs = 0;
      for (int i = 0; i < 4; i++)
      {
            if (m_Cards[i] == 2)
            {
                  if (pairs > 0)
                  {
                        m_Amt_Remaining += 3.00;
                        break;
                  }
                  else
                  {
```

```
/*
```

```
                    pairs++;
                }
            }
            else if (m_Cards[i] == 3)
            {
                m_Amt_Remaining += 6.00;
                break;
            }
            else if (m_Cards[i] == 4)
            {
                m_Amt_Remaining += 9.00;
                break;
            }
        }
}
```

Последние замечания

После ввода всех рассмотренных фрагментов кода еще потребуется "сцепить их", изменив **OnDealCards()** так, чтобы вызывались новые методы. В листинге 5.9 показаны необходимые изменения.

Листинг 5.9 Завершенный метод OnDealCards().

```
void CFourUpDlg::OnDealCards()
{
    // 1. Вычесть 2.00, что бы ни случилось
    m_Amt_Remaining -= 2.00;

    // 2. Сдать карты
    DealCards();

    // 3. Вычислить выигрыш, который затем добавить к m_Amt_Remaining
    CalculateWinnings();

    // 4. Отобразить итог m_Amt_Remaining
    CString s;
    s.Format("Остающееся количество $ %.2f", m_Amt_Remaining);
    m_Amt_Left.SetWindowText(s);
}
```

Рискуйте!

Мы рассмотрели все, что нужно. Сохраните рабочую область, постройте приложение и запустите его на выполнение. К счастью, FourUp никогда не будет "раздевать вас до нитки". Если ситуация становится слишком мрачной, просто запустите приложение повторно.

На настоящий момент времени вы увидели, как создавать завершенное приложение MFC. Примите поздравления: вы готовы создавать ваши собственные программы MFC. Конечно, вам потребуются дополнительные знания о классах MFC и их возможностях.

В следующей главе исследуются вопросы создания и управления окнами. Вы также детально познакомитесь с двумя из наиболее популярных элементов управления MFC: **CStatic** и **CButton**. Вы увидите, что **CButton** — это фактически все семейство элементов управления, включающее кнопки, флажки, переключатели и группы.

Глава 6

Понятие об элементах управления

В предыдущей главе вы встретились со многими обитателями из зверинца элементов управления Windows. В этой главе более подробно исследуются их повадки и места обитания.

Шпионаж стал распространенной тактикой государств с тех пор, как Мозес (Moses) послал двух шпионов на Землю обетованную. Шпионаж редко является открытым, вместо этого большинство государств покрывает действия рыцарей "плаща и кинжала" завесой тайны.

Во времена холодной войны в 1950-х и 1960-х американцы встретились с миром шпионажа как с многочисленными реальными (так и воображаемыми) случаями коммунистического шпионажа. Драматические обвинения и испытания, подобно случаям с Юлием и Этель Розенбергами (Julius и Ethel Rosenberg) и Алжиром Хиссом (Alger Hiss), переполняли вечерние выпуски новостей.

Интерес публики в подобных случаях обычно направлен на шпиона, или агента; но его роль, возможно, не является самой важной в шпионской игре. Каждый шпион *управляется* вторым агентом, который осуществляет связь со специальным государственным органом. По каналу управления передаются команды от управляющего органа к шпиону и наоборот.

Элементы управления в программе Windows немного напоминают каналы управления в мире шпионажа. Они могут выглядеть довольно обычными объектами, однако при этом формируют жизненно важный информационный канал между прикладной программой и пользователем: ничего не принимается или не передается пользователю в обход элементов управления.

В этой главе у вас начнет формироваться детальное представление об элементах управления Windows. Будет уделено особое внимание важным элементам управления **CStatic** и **CButton**. Поэтому возьмите ваш верный пистолет Walther PPK, усаживайтесь в автомобиль Astin Martin DB5 и отправляйтесь. О, и не забудьте пристегнуть привязные ремни!

Краткий обзор CWnd

В главе 4 была рассмотрена базовая структура приложения, основанного на диалоговых окнах, которое использует производный класс **CDialog** как главное окно. Напомним, что класс **CDialog** является производным от **CWnd**, поэтому экземпляр **CDialog** наследует все атрибуты и поведение **CWnd**. Однако класс **CDialog** также и отличается от **CWnd**. Например, Visual C++ создает окна **CDialog** иным образом, нежели окна **CWnd**. Давайте рассмотрим эти объекты, используя наше приложение FourUp.

Когда класс **CFourUpApp** начинает выполняться, он вызывает виртуальный метод **InitInstance()**, который создает главное окно приложения. Соответствующий код выглядит примерно так:

```
CFourUpDlg dlg;              // Создать окно
m_pMainWnd = &dlg;           // Сохранить указатель на полученное
                             // главное окно
```

Если главное окно FourUp было непосредственно получено из **CWnd**, а не из подкласса **CDialog**, метод **InitInstance()** будет выглядеть очень похоже:

```
m_pMainWnd = new CMainWindow;
```

Окно, основанное на **CDialog**, создается как локальная переменная в стеке, в то время как окно, основанное на **CWnd**, создается как динамическая переменная в свободной области памяти (куче) — иначе две части кода практически *не* отличаются. Каждая часть кода вызывает заданный по умолчанию конструктор

класса главного окна, затем сохраняет адрес главного окна объекта в элементе данных **m_pMainWnd** класса **CWinApp**. Только заглянув вовнутрь двух конструкторов, можно увидеть реальное отличие между ними.

Устройство окна

Давайте начнем с исследования работы конструктора класса **CFourUpDlg**. Для нас представляют интерес следующие фрагменты кода:

```
CFourUpDlg::CFourUpDlg(CWnd* pParent /*=NULL*/)
    : CDialog(CFourUpDlg::IDD, pParent)
{
    // Опущенные детали
}
```

Обратите внимание на то, что класс **CFourUpDlg** на самом деле не выполняет никаких конструкторских задач. Вместо этого он использует список инициализации для вызова конструктора **CDialog**, передавая два фрагмента информации. Фрагмент, представляющий для нас интерес — первый параметр **CFourUpDlg::IDD**.

Значение **CFourUpDlg::IDD** — это перечень (определенный в файле заголовков для CFourUpDlg), который содержит идентификатор ресурса для шаблона диалогового окна (содержится в сценарии ресурса, т.е. в CFourUp.RC). Конструктор **CDialog** считывает информацию из шаблона диалогового окна, а затем создает окно, используя спецификации, предоставленные шаблоном. Для создания главного диалогового окна (и всех элементов управления, содержащихся в шаблоне диалогового окна), конструктор **CDialog** использует функцию **CWnd::Create()** или **CWnd::CreateEx()**, вызывая функцию один раз для главного диалогового окна и повторно для каждого возникающего элемента управления.

Функция CWnd::Create()

Даже если вам не требуется явно использовать функцию **CWnd::Create()** для создания элементов управления, которые будут помещаться в диалоговом окне, необходимо понять, как функция работает. Это нужно уже по причине частого использования данной функции в Windows и MFC.

Ниже показан прототип функции **CWnd::Create()**:

```
virtual BOOL Create( LPCTSTR lpszClassName,
                     LPCTSTR lpszWindowName,
                     DWORD dwStyle,
                     const RECT& rect,
                     CWnd* pParentWnd,
                     UINT nID,
                     CCreateContext* pContext = NULL);
```

Как видно из прототипа, функция **Create()** имеет семь параметров, последний из которых по умолчанию устанавливается в **NULL**. Вкратце опишем назначение каждого параметра:

- *lpszClassName* — строка, именующая "класс" окна, т.е. структура данных, которая динамически создается Windows с целью упрощения создания нескольких окон, имеющих одинаковые свойства. Поскольку это — не подлинный класс C++, необходимо представлять его как структуру данных, однако в документации по Windows используется термин *класс*.

Обычно в MFC передается **NULL**, который указывает функции **Create()** на использование заданного по умолчанию оконного класса **CWnd**. Если был зарегистрирован пользовательский класс, здесь можно передавать имя. Если создается экземпляр встроенного элемента управления, подобно элементам управления редактированием Windows, здесь передается имя **WNDCLASS** — **"EDIT"**.

- *lpszWindowName* — "текстовый" атрибут, связанный с окном. Если окно является главным, то эта строка появляется в названии окна. С другой стороны, если окно является кнопкой, эта строка появляется в качестве заголовка кнопки. Некоторые окна, например, прямоугольный статический элемент управления, не имеют текстовых свойств. В подобных случаях просто передается пустая строка, т.е. "".
- *dwStyle* — наиболее важная характеристика, которую необходимо понять (она будет обсуждаться ниже). Параметр **dwStyle** обеспечивает возможность определения опций для специфического объекта окна после его создания. Например, можно определить, что это окно имеет кнопку закрытия или изменения размера. Подобные атрибуты не являются характеристиками класса окна, но определяются при создании любого окна.
- *rect* — определяет координаты окна. **RECT** — это структура C, которая содержит четыре общедоступных поля: **left**, **top**, **bottom** и **right**. Если создается окно верхнего уровня или главное окно, эти координаты измеряются начиная от левого верхнего угла вашего экрана. Если создается дочернее окно или элемент управления, то координаты отсчитываются относительно родительского окна.
- *pParentWnd* — указатель на родительское окно. Для дочерних окон необходимо всегда обеспечить соответствующее значение; для окон верхнего уровня можно передавать **NULL**.
- *nID* — целочисленный идентификатор, который идентифицирует окно для его родительского окна и позволяет родительскому окну посылать сообщения дочернему окну. Этот параметр не используется для окон верхнего уровня — можно передавать **NULL**, если окно не имеет родительского окна.
- *pContext* — используется только тогда, когда необходимо перекрыть обычный процесс соединения документа и его представлений. Этот параметр будет использоваться редко, и он никогда не применяется вне архитектуры "документ-представление".

Обзор стилей окна

Стили окна — параметр **dwStyle**, переданный в метод **CWnd::Create()** — позволяет настраивать различные атрибуты окна. Общие стили окна (применимые для широкого разнообразия классов окна) состоят из набора идентификаторов, которые начинаются с **WS_**. Каждый идентификатор ссылается на специфический атрибут. В табл. 6.1 показаны некоторые наиболее общие стили.

Таблица 6.1 Наиболее часто используемые стили окна

Стиль окна	*Описание*
WS_BORDER	Добавляет рамку к окну
WS_CAPTION	Добавляет название к окну

Стиль окна	*Описание*
WS_CHILD	Создает дочернее окно
WS_HSCROLL	Добавляет горизонтальную полосу прокрутки к окну
WS_MAXIMIZEBOX	Добавляет кнопку максимизации к окну
WS_MINIMIZEBOX	Добавляет кнопку минимизации к окну
WS_OVERLAPPED	Создает перекрывающееся окно
WS_POPUP	Создает всплывающее окно
WS_SYSMENU	Добавляет системное меню к заголовку окна
WS_THICKFRAME	Создает окно изменяемого размера
WS_VISIBLE	Делает окно видимым после его создания
WS_VSCROLL	Добавляет вертикальную полосу прокрутки в окно

Три стиля окна определяют связь окна с другими окнами. Если используется стиль **WS_OVERLAPPED**, то у окна нет родительского окна: это окно верхнего уровня. Если присваивается стиль **WS_POPUP**, то обычно у окна имеется родительское окно, но окно не ограничивается местоположением родительского окна. Вместо этого оно "плавает" поверх родительского окна. Если окно имеет стиль **WS_CHILD**, оно должно иметь родительское окно, и при этом "отсекается" областью, занимаемой родительским окном на экране. Если подобное окно перемещается за пределы области родительского окна, то оно не отображается. Рассмотренные три стиля являются взаимно-исключающими.

Другие стили определяют различные характеристики окон, которые они описывают. Например, присвоение стиля **WS_CAPTION** добавляет название окна, а также делает окно перемещаемым. Пользователь может перемещать окно при помощи клавиатуры или мыши, если только окно имеет стиль **WS_CAPTION**. Аналогично, если окну назначается стиль **WS_THICKFRAME**, пользователь может изменять размеры окна. Окна со стилем **WS_BORDER** изменять размеры не могут.

Обычно создаются окна, которые имеют более одного стиля. Для этого различные стили объединяются при помощи поразрядного оператора **OR** — |. Рассмотрим пример. Предположим, что требуется получить окно верхнего уровня, которое имеет заголовок, кнопку максимизации, не имеет кнопки минимизации и не может изменять свои размеры. Для этого функции **CWnd::Create()** нужно передать следующий стиль окна:

```
WS_OVERLAPPED | WS_CAPTION | WS_MAXIMIZEBOX | WS_BORDER
```

Для уменьшения объема работы по вводу информации несколько стилей объединяются в наиболее часто используемые опции. Стиль **WS_OVERLAPPEDWINDOW** создает перемещаемое, с изменяемыми размерами окно верхнего уровня со всеми настройками, в то время как **WS_POPUPWINDOW** создает стандартное диалоговое окно.

Создание дочерних окон

Как уже рассматривалось в главе 4, Windows включает шесть универсальных классов управления дочерними окнами. В MFC эти классы представлены классами C++ **CStatic**, **CButton**, **CEdit**, **CListBox**, **CComboBox** и **CScrollBar**. Каждый из

этих классов получен из **CWnd** и, следовательно, наследует все неприватные функции **CWnd**.

Однако каждый элемент управления перекрывает функцию **CWnd::Create()** и обеспечивает собственные стили окна. Эти специфические для класса стили окна используются наряду с общими стилями окна для создания специфических атрибутов, которые желательны для элементов управления дочернего окна.

Рассмотрим пример. Предположим, что необходимо создать кнопку OK, подобную используемой в классе **CAboutDlg**. Используя класс **CButton**, запишем следующий код:

```
CButton okBtn;
// Установить размер: x,y = 100,100, x1,y1 = 200,150
CRect rect(100, 100, 200, 150);
okBtn.Create( "OK",
              WS_CHILD | WS_VISIBLE | BS_DEFPUSHBUTTON,
              rect, this, IDOK);
```

Эти строки создают заданную по умолчанию кнопку (кнопка OK), на которой помещен текст "OK", 100 единиц в ширину (x1 - x = 100) и 50 единиц в высоту (y1 - y = 50), с левым верхним углом, находящимся в позиции 100,100. Окно имеет идентификатор времени выполнения, заданный значением **IDOK**.

Вы можете сказать себе: "Это большая работа!" Вы правы, действительно так. Также очень трудно записать подобный код с любой точностью, поскольку необходимо очень точно позиционировать каждый элемент управления с использованием координат x-y. Применение Dialog Editor немного упрощает ситуацию.

Изучить функцию **Create()** необходимо, даже если не будете использовать ее, просто для ознакомления со стилями окна, специфичными для элементов управления. Вы видели стиль **BS_DEFPUSHBUTTON**, используемый во втором параметре функции **Create()**? Он определяет, что вы хотите создавать заданную по умолчанию кнопку. Префикс ***BS_*** идентифицирует его как идентификатор Button Style (Стиль кнопки). Подобно всем элементам управления дочернего окна, этот специфический для элементов управления идентификатор окна объединен с общими стилями **WS_VISIBLE** и **WS_CHILD**.

Если для создания элементов управления используется Dialog Editor, нет необходимости применять идентификаторы стиля кнопки. Вместо этого потребуется отметить некоторые флажки поле в диалоговом окне Properties (Свойства) элемента управления. Здесь все еще нужно ориентироваться в различных стилях, однако, поскольку они описаны в документации Windows, выбор опций в Dialog Editor особых трудностей не вызовет.

Полезные функции CWnd

Элементы управления, полученные из **CWnd**, наследуют методы **CWnd**. Однако подобное наследование часто затрудняет поиск нужной функции. Например, если вы необходимо изменить шрифт элемента управления **CStatic**, то реализацию метода **SetFont()** в классе **CStatic** вы не найдете. Это не означает, что шрифт изменять нельзя — просто придется смотреть немного выше в иерархии наследования.

Достаточно хорошо изучив класс **CWnd**, вы найдете **SetFont()** наряду с целым рядом других полезных методов. В табл. 6.2 показан ряд методов, используемых при работе с элементами управления дочернего окна.

Таблица 6.2 Полезные методы CWnd

Имя функции	*Описание*
CenterWindow	Центрирует окно относительно родительского окна
EnableWindow	Разрешает или запрещает окно или элемент управления
GetClientRect	Выбирает размерности элемента управления
GetDlgCtrlId	Выбирает идентификатор, связанный с элементом управления
GetDlgItem	Выбирает элемент управления, связанный с идентификатором
GetFont	Выбирает шрифт, используемый окном
GetParent	Выбирает родителя элемента управления
GetWindowText	Выбирает заголовок, связанный с окном
MoveWindow	Изменяет одновременно позицию и размер окна или элемента управления
SetFont	Изменяет шрифт, используемый окном
SetWindowText	Изменяет заголовок, связанный с окном
ShowWindow	Показывает или скрывает окно или элемент управления

CWnd — это один из самых больших классов в иерархии MFC, поэтому поиск требуемых методов может оказаться затруднительным. Если вы не видите ожидаемых методов в таблице 6.2, просмотрите оперативную документацию — описание искомого метода может оказаться именно там!

Теперь давайте начнем рассмотрение универсальных классов элементов управления Windows. Начнем с исследования класса **CStatic**.

Углубленное знакомство с CStatic

Вы уже имели большой опыт использования объектов класса **CStatic**. В приложении **FourUp** элементы управления **CStatic** применялись для отображения текстовой информации, растровых изображений и пиктограмм — в общем, целого набора функций. Но объекты **CStatic** способны на большее.

Элементы управления **CStatic** делятся на три различных типа: отображающие текст, отображающие прямоугольники и выводящие изображения. Как и можно было ожидать, различные типы отличаются при помощи стилей окна. Давайте сначала рассмотрим, как создать элементы управления **CStatic** программным путем. Затем будут исследоваться все три упомянутых типа.

Создание элементов управления CStatic

Для создания элемента управления **CStatic** программным способом используется перекрытая функция **Create()**. В функции **CStatic::Create()** присутствует только пять параметров (из семи параметров **CWnd**). Вот пример использования этой функции:

```
m_Mom.Create("Привет, мама",                          // Текст
             WS_CHILD | WS_VISIBLE | SS_CENTER,       // Стили
             rect,                                    // Позиция
             this,                                    // Родитель
             IDC_MOM);                                // Идентификатор
                                                      // элемента
                                                      // управления
```

Как и в случае со всеми элементами управления, определяются стили окна **WS_CHILD** и **WS_VISIBLE**. Можно также определить атрибуты стиля окна, относящегося к **CStatic**, имена которых начинаются с префикса ***SS_***.

Давайте рассмотрим, как этот процесс создания работает в миниатюрных приложениях. Мы не будем знакомить вас со всем проектом, но если вы захотите последовать дальше, то сможете легко это сделать самостоятельно. (Другие примеры в этой главе являются модификацией рассматриваемого.) Ниже приведены шаги, которым нужно следовать при создании миниатюрного приложения:

1. Используйте AppWizard для создания нового приложения, основанного на диалоговых окнах. Назовите его TestBedOne.
2. Удалите метку TODO:... (ЧТО СДЕЛАТЬ:...) в главном диалоговом окне.
3. Вручную добавьте элемент данных типа **CStatic** с именем **m_Mom** в заголовок класса **CTestBedOneDlg**, используя следующую строку кода:

```
CStatic m_Mom;
```

4. В функции **TestBedOneDlg::OnInitDialog()** разыщите комментарий, гласящий: *// TODO: Add extra initialization here* (*// ЧТО СДЕЛАТЬ: Добавьте здесь дополнительный код инициализации*).

   Ниже комментария добавьте следующие строки кода, которые создают статическую метку:

```
CRect rect;
GetClientRect(&rect);
rect.bottom = 25;
rect.right -= 100;
rect.left += 100;
M_Mom.Create("Привет, мама", WS_CHILD | WS_VISIBLE | SS_CENTER,
                        rect, this, IDC_MOM);
```

5. По завершении выберите команду View | Resource Symbols (Вид | Символы ресурса) из главного меню и добавьте новый символ **IDC_MOM** со значением 2000.

Теперь можно откомпилировать и выполнить приложение. Результат показан на рис. 6.1.

Конечно, в большинстве случаев всю эту работу выполнять не придется. Потребуется лишь перетащить элемент управления текстом с панели инструментов Control, опустить его в форму, изменить его размеры и позволить Dialog Editor Visual C++ позаботиться о подробностях.

Теперь давайте рассмотрим стили **CStatic**, из которых будет производиться выбор.

Текстовые стили CStatic

Стиль текста элемента управления **CStatic** выбирается из следующих пяти: **SS_CENTER**, **SS_LEFT**, **SS_RIGHT**, **SS_LEFTNOWORDWRAP** и **SS_SIMPLE**. Большинство этих стилей имеют очевидные имена; мы рассмотрели соответствующие выборы в Dialog Editor в главе 2. Стиль **SS_SIMPLE** очень полезен, поскольку он обеспечивает несколько более быстрое отображение и задействует меньшее количество ресурсов.

Стили **SS_LEFT**, **SS_CENTER** и **SS_RIGHT** выполняют автоматический перенос слова, если определенный текст не умещается в одной строке; другие два стиля не переносят по словам. Если текст не вмещается внутри прямоугольника, определенного при создании объекта **CStatic**, оставшийся текст не отображается.

Эти текстовые стили можно объединять с **SS_NOPREFIX**, чтобы предотвратить автоматическое преобразование символа амперсанда на символ подчеркивания, с **SS_CENTERIMAGE** — для центрирования текста в вертикальном направлении внутри элемента управления, или с **SS_SUNKEN** — для отображения вдавленного прямоугольника вокруг элемента управления.

При работе с Dialog Editor используется диалоговое окно Text Properties (Свойства текста) для манипулирования элементами управления **CStatic**, принадлежащими к текстовому семейству. Посмотрите на рис. 6.2, на котором показано диалоговое окно Text Properties.

Стили рамки CStatic

Стили рамки **CStatic** используют ту же самую функцию **Create()**, которая применялась в случае с текстовыми стилями, но здесь игнорируется первый (текстовый) параметр. Семейство рамок состоит из трех множеств рамок: заполненная, пустая и "вытравленная".

Заполненные рамки просто рисуют сплошной прямоугольник на экране. Здесь имеются три опции: **SS_BLACKRECT**, **SS_GRAYRECT** и **SS_WHITERECT**. Несмотря на имена, эти рамки не всегда рисуют в черном, сером и белом цветах (хотя и могут делать это). Вместо этого используются системные цвета (определенные пользователем в Control Panel), идентифицированные как **COLOR_WINDOWFRAME**, **COLOR_BACKGROUND** и **COLOR_WINDOW**.

Пустые рамки используются с теми же самыми тремя цветами и рисуют рамку шириной в один пиксел заданных размеров. Стили пустой рамки следующие — **SS_BLACKFRAME**, **SS_GRAYFRAME** и **SS_WHITEFRAME**.

ПРИМЕЧАНИЕ

Не путайте метод CStatic::SetCursor() с функцией CWnd::OnSetCursor(). Последняя позволяет ассоциировать курсор Windows с элементом управления. Метод CStatic::SetCursor() просто выводит изображение курсора как любой другой ресурс растрового изображения.

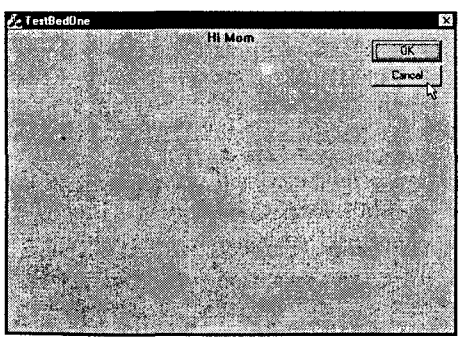


РИСУНОК 6.1 *Приложение TestBedOne.*

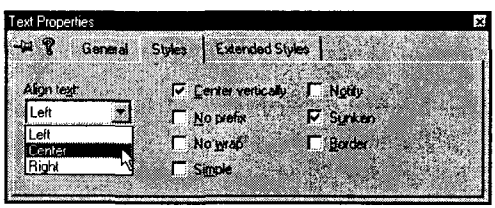


РИСУНОК 6.2 *Использование диалогового окна Text Properties для манипулирования элементами управления CStatic.*

Три стиля вытравленной рамки — **SS_ETCHEDFRAME**, **SS_ETCHEDHORZ** и **SS_ETCHEDVERT** — отображают трехмерные пустые рамки, которые лучше всего выглядят при отображении в стандартном сером диалоговом окне. Стили **SS_ETCHEDHORZ** и **SS_ETCHEDVERT** приводят к эффекту трехмерности только для верхних и нижних границ либо для левых и правых границ.

Стили изображения CStatic

В дополнение к тексту и рамкам, класс CStatic можно использовать для создания и быстрого вывода изображений, как было показано в главе 3. Существуют три стиля изображений: **SS_BITMAP**, **SS_ENHMETAFILE** и **SS_ICON**.

Создание объекта **CStatic**, выводящего изображение, требует выполнения множества шагов. Необходимо отдельно загрузить ресурсы изображения (растры, метафайлы или пиктограммы). Затем для вывода изображения используются функции **CStatic SetIcon()**, **SetBitmap()**, **SetCursor()** и **SetEnhMetaFile()**.

В Dialog Editor и стили рамки **CStatic**, и стили изображения **CStatic** отображают диалоговое окно Picture Properties (Свойства изображения), позволяющее выбрать соответствующий стиль. Вместо флажков, как это было в случае с диалоговым окном Text Properties, диалоговое окно Picture Properties предлагает пару взаимодействующих полей со списками. Сначала выбирается тип изображения, которое нужно вывести: рамка, пиктограмма и т.д. Затем, в зависимости от выбранного типа, становится активным поле со списком Image (Изображение) или Color (Цвет). Диалоговое окно Picture Properties показано на рис. 6.3.

Работа с элементами управления CStatic

Элементы управления **CStatic** на самом деле не делают много — как правило, они находятся на месте и хорошо смотрятся. Часто этого и достаточно. Однако иногда возникает желание отобразить метку или изображение для сообщения пользователю о щелчке на кнопке. Для выполнения этого при создании элемента управления **CStatic** потребуется добавить еще один стиль: **SS_NOTIFY**.

Однако, даже если определяется стиль **SS_NOTIFY**, элементы управления **CStatic** остаются "неразговорчивыми". Они отвечают только на события **STN_CLICKED**, **STN_DBLCLK**, **STN_DISABLE** и **STN_ENABLE** — и для реализации отклика необходимо вручную добавить таблицы ответов на сообщения.

Большинство из нас предпочитает избегать такой тяжелой работы. К счастью, ClassWizard помогает обнаруживать одиночные щелчки мышью (как будто бы элемент управления — это кнопка). Существует пример, который является модификацией созданной ранее программы TestBedOne:

1. Добавьте текстовую метку в центре главного диалогового окна TestBedOne. Измените имя элемента управления на **IDC_CLICKME** и предоставьте ему возможность сказать следующее — *Щелкните на мне, если посмеете!!!*. Кроме того, отметьте Notify (Уведомление) в диалоговом окне Text Properties.
2. Дважды щелкните на элементе управления, чтобы ClassWizard открыл диалоговое окно Add Member Function. Примите предлагаемое имя (**OnClickMe**) и затем добавьте следующую выделенную строку, завершающую метод:

```
void CTestBedOneDlg::OnClickme()
{
```

```
        // ЧТО СДЕЛАТЬ: Добавьте код обработчика уведомления
        // элемента управления
        m_Mom.SetWindowText(
        "Ну, что? Я похожу на кнопку?");
}
```

3. Откомпилируйте и запустите приложение. Если выполнить щелчок на метке в центре экрана, динамическая метка в верхней части экрана изменится. На рис. 6.4 показано, как это выглядит.

Методы CStatic

Объекты **CStatic** не только относительно тихие, они также не обладают множеством талантов. Если найти описание **CStatic** в оперативной справке, можно обнаружить, что для этого объекта определено только восемь методов. Конечно, поскольку было упомянуто, что каждый объект **CStatic** также является объектом **CWnd**, то и все функции **CWnd** — в вашем распоряжении.

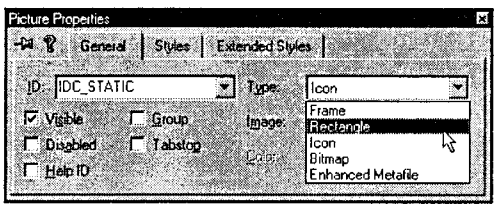


РИСУНОК 6.3 *Использование диалогового окна Picture Properties для манипулирования элементом управления CStatic.*

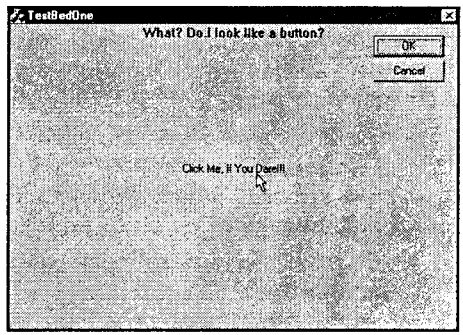


РИСУНОК 6.4 *Обнаружение щелчков мыши при помощи метки CStatic.*

Каждый метод **CStatic** либо отображает, либо выбирает один из четырех типов изображений, которые может обрабатывать класс **CStatic**. Методы отображения изображения в элементе управления **CStatic** — это **SetBitmap()**, **SetCursor()**, **SetEnhMetaFile()** и **SetIcon()**. Методы выборки — это **GetBitmap()**, **GetCursor()**, **GetEnhMetaFile()** и **GetIcon()**.

В приложении FourUp эти методы использовались с играющими картами. Там они были нужны для изменения карт, когда игрок выполнял щелчок на кнопке Deal. Из этого примера вы узнали, что процесс отображения пиктограммы является несколько более сложным, чем просто передача имени файла изображения. (Кроме того вы знаете, что процесс получения расширенного метафайла немного затруднен.)

Только с целью освежить познания, полученные в последней главе, выведем изображение программным путем. Для этого потребуется выполнить следующие шаги:

1. Добавьте файл изображения в свой проект как ресурс, используя пункт меню Insert | Resource (Вставить | Ресурс).

2. Получите дескриптор ресурса во время выполнения. Для пиктограмм это делается с использованием **CWinApp::LoadIcon()**. Для растрового изображения можно присоединить растр к элементу управлению **CStatic**, используя Dialog Editor (как было сделано для текстовой эмблемы FourUp), затем вызовите метод **GetBitmap()** для получения требуемого **HBITMAP**.
3. Вызовите метод **SetIcon()** или **SetBitmap()** из **CStatic**, передавая в качестве параметра **HICON** или **HBITMAP**, полученный на шаге 2.

Перечисленное практически истощает возможности класса **CStatic**. Давайте перейдем к другому классу, который является немного более активным, но который, подобно классу **CStatic**, пытается совместить несколько очень различных возможностей в одном пакете.

Семейство CButton

Класс **CButton** состоит прежде всего из различных элементов, которые выполняют действия, когда на них щелкать. Явное исключение из этого семейства — раздел групп (groupbox), который выглядит принадлежащим классу **CStatic**, но помещенный вместе с кнопками из-за исторической ошибки.

Класс **CButton** имеет четыре элемента: кнопки (или команды), флажки, переключатели и группы.

- Кнопки срабатывают, когда на них осуществлять щелчки. Начиная с Windows 3.0, кнопки изображаются с имитацией трехмерности). При щелчке кнопка "вдавливается" в поверхность экрана; когда ее отпустить, она поднимается в прежнее положение.
- Флажки требуют от пользователя приложения неисключающий ввод типа истина-ложь. Флажок имеет заголовок и **область действия**. Область действия — это маленькое поле, которое отмечается галочкой в случае выбора и очищается при отмене выбора. Флажок действует подобно переключателю и сохраняет информацию о состоянии — либо установлен, либо не установлен. При размещении в диалоговом окне флажки отображаются в трехмерном виде; при размещении в обычном окне — это не так.
- Переключатели представляют собой другую форму флажков и применяются для выполнения исключительного выбора из нескольких вариантов. Когда несколько переключателей принадлежат одной и той же группе, одновременно можно выбрать только один из них. Windows отменяет выбор других элементов группы, когда выбран один из них. Подобно флажкам, переключатели сохраняют информацию о своем состоянии в индивидуальном порядке. Также подобно флажкам, переключатели отображаются в трехмерном виде в диалоговых окнах (их расположение в большинстве случаев), но не других случаях.
- Группы, последний элемент семейства **CButton**, являются очень похожими на стиль **SS_ETCHEDFRAME** из **CStatic**. Единственное отличие состоит в том, что **CButton** имеют работающее свойство заголовка. **CButton** не имеет состояния и не посылает сообщений, однако (как было показано в **CFourUp**) допускается посылать им сообщения.

Создание элементов управления CButton

Для создания объекта **CButton** программным путем используется заданный по умолчанию конструктор, затем вызывается функция **Create()**, как в случае с классом **CStatic**. Различия, конечно, заключаются в деталях, но эти различия не являются значительными. Большинство этих различий, как можно было ожидать, имеют отношение к использованию стилей для создания различных множеств **CButtons**.

Ниже приведен пример кода, который можно добавить в приложение TestBedOne для создания стиля кнопки объекта **m_Dad** типа **CButton**:

```
GetClientRect(&rect);
rect.top = rect.bottom - 55;
rect.bottom -= 25;
rect.left = (rect.right - 175);
rect.right -= 25;
m_Dad.Create("Самый дорогой на свете папа",     // Текст
      WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON,    // Стиль
      rect,                                     // Позиция
      this,                                     // Родитель
      IDC_DAD);                                 // Идентификатор
                                                // элемента управления
```

Выполните следующие шаги для апробирования этого кода в TestBedOne:

1. Добавьте элемент данных **CButton** с именем **m_Dad** в определение класса **CTestBedOneDlg**:

```
CButton m_Dad;
```

2. Добавьте новый идентификатор ресурса **IDC_DAD**, используя пункт меню View | Resource Symbol. Присвойте ему значение 2001.
3. Добавьте код, показанный ранее в **CTestBedOneDlg::OnInitDialog()**, после кода, устанавливающего старый добрый **m_Mom**.

Результат можно видеть на рис. 6.5.

Стили кнопки CButton

Существуют два стиля кнопки: **BS_PUSHBUTTON** и **BS_DEFPUSHBUTTON**. Их можно использовать при создании элемента управления **CButton**. Эти два стиля имеют только одно реальное различие: в шаблоне диалогового окна, подобного созданному в Dialog Editor, стиль **BS_DEFPUSHBUTTON** указывает, что определенная **CButton** должна стать кнопкой по умолчанию.

Кнопка по умолчанию уведомляется, если пользователь нажимает клавишу ENTER без предварительного выбора другой кнопки. Однако если пользователь выбирает другую кнопку (при помощи клавиши Tab), то нажатие на Enter имеет эффект щелчка на выбранной кнопке, а не на кнопке по умолчанию.

Windows позволяет определить в диалоговом окне только одну кнопку по умолчанию. Если вы определите более одной кнопки, то последняя созданная кнопка становится кнопкой по умолчанию. Когда заданная по умолчанию кнопка отображается, она выводится с полужирной рамкой.

В Dialog Editor атрибуты **CButton** элементов управления кнопкой **CButton** определяются с использованием диалогового окна Push Button Properties (см. рис. 6.6). Стили отображения обсуждаются в следующем разделе.

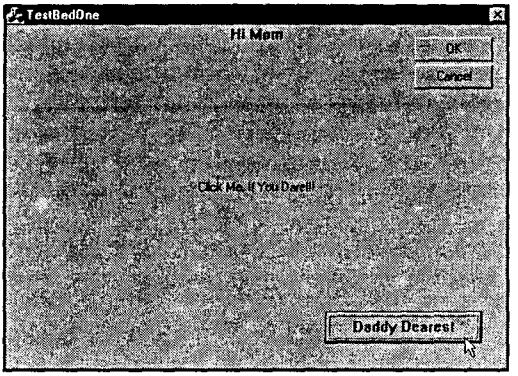


РИСУНОК 6.5 *Добавление стиля кнопки CButton TestBedOne.*

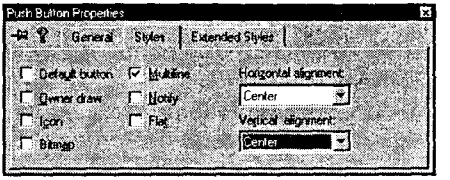


РИСУНОК 6.6 *Использование диалогового окна Push Button Properties для манипулирования элементами управления в CButton.*

Стили отображения

Подобно большинству других элементов управления, элементы управления **CButton** имеют несколько стилей, которые определяют скорее внешний вид кнопки, чем ее тип или поведение. Большинство этих стилей "отображения" появилось с приходом Windows 95.

Первая группа стилей отображения воздействует на выравнивание текста или изображений внутри объекта **CButton**. Стили **BS_LEFT**, **BS_RIGHT** и **BS_CENTER** управляют выравниванием по горизонтали, в то время как **BS_TOP**, **BS_BOTTOM** и **BS_VCENTER** управляют выравниванием по вертикали. Два дополнительных стиля выравнивания работают только с флажками и переключателями: стили **BS_LEFTTEXT** и **BS_RIGHTBUTTON** помещают заголовок такого элемента управления слева, а не в заданной по умолчанию позиции справа.

Два стиля позволяют вместо текста, который обычно появляется на изображении кнопки, использовать растровое изображение или пиктограмму. Вы можете использовать стили **BS_BITMAP** или **BS_ICON** с кнопками, флажками и переключателями. Применение растра или пиктограммы с кнопкой не столь удобно как использование элемента управления **CStatic** в Dialog Editor. В элементах управления **CStatic** можно выбирать изображения из раскрывающегося списка, однако в случае с **CButtons** необходимо для вывода изображений вызывать методы **CButton::SetIcon()** и **CButton::SetBitmap()**.

Упоминания заслуживают три дополнительных новых стиля кнопки:

- Стиль **BS_MULTILINE** позволяет создавать кнопки с текстом, распространяющимся на несколько строк. Можно вручную создавать разрыв строки, вставляя управляющую последовательность (\n) или разрешить автоматический перенос строки. Стиль **BS_MULTILINE** обладает некоторым дезориентирующим эффектом: множество строк не центрируются в индивидуальном порядке при использовании стиля **BS_CENTER** (хотя весь текстовый блок выровнен по центру).
- Стиль **BS_FLAT** позволяет применять флажки нового стиля и переключатели со страницы опций Internet Explorer. Этот стиль можно также использовать с кнопками.

- Стиль **BS_PUSHLIKE** не оказывает влияние на кнопки. Однако он заставляет переключатели и флажки вести себя обычным образом, но при этом походить на кнопки. Если щелкнуть на флажке со стилем **BS_PUSHLIKE**, кнопка утапливается точно так же, как обычная кнопка. Вместо обратного выпрямления утопленное положение кнопки остается до тех пор, пока не щелкнуть на ней снова. Когда используются переключатели, стиль работает точно так же, но щелчок на одной кнопке в группе вызывает возврат в исходное положение других кнопок.

Большинство пользователей, узнающих о том, что в их системе установлено более 200 шрифтов, отнюдь не должны использовать каждый из них при написании какого-либо документа. Аналогично, все рассмотренные опции форматирования предоставляют расширенные возможности управления завершенным интерфейсом, но необязательно все их использовать. Некоторые из различных стилей кнопки, привязанные к кнопкам, показаны на рис. 6.7, но в реальной жизни рекомендуется выбирать для кнопок единый стиль и придерживаться его во всех своих приложениях.

Быстрое и простое создание кнопок BS_ICON и BS_BITMAP

Несмотря на появление новых и простых в использовании свойств Developer Studio, создание кнопок со стилями BS_ICON или BS_BITMAP все еще требует выполнения значительного объема работы. Имеются слишком много частей, "болтающихся" вокруг. Ниже приведен совет, помогающий отслеживать все части.

Предположим, что необходимо создать новую кнопку в стиле BS_BITMAP, вызывающую программу проверки грамматики. Назовем кнопку IDC_SPELL_BTN. Для отображения растра потребуются четыре дополнительных порции информации:

1. Ресурс растрового изображения.
2. Статический элемент управления для удерживания растра.
3. Управляющая переменная, позволяющая выбирать растр из статического элемента управления.
4. Управляющая переменная для кнопки IDC_SPELL_BTN, позволяющая установить растр для кнопки.

Используя непротиворечивое соглашение по именованию, легко сообщить сразу, какая "часть" головоломки рассматривается. Если вы добавляете ресурс растрового изображения, дайте ему имя IDB_SPELL_BM и поименуйте статический элемент управления, содержащий ресурс, как IDC_SPELL_BM. Если вы создаете управляющие переменные, поименуйте статический элемент управления как m_SpellBM, а кнопку — как m_SpellBtn.

Если необходимо назначить кнопке растровое изображение в функции InitDialog(), просто запишите следующую строку кода:

```
m_SpellBtn.SetIcon(m_SpellBm.GetBitmap());
```

Стили кнопки флажка CButton

При создании флажка **CButton** можно осуществить выбор из четырех различных стилей: **BS_CHECKBOX**, **BS_AUTOCHECKBOX**, **BS_3STATE** или **BS_AUTO3STATE**. Существует практически стопроцентная вероятность, что вы просто используете стиль **BS_AUTOCHECKBOX**. Вы используете два стиля **3STATE**, если флажок может иметь промежуточное состояние: да, нет или возможно. Эти стили не будут применяться часто, поскольку обработка случая "возможно" усложняет остальную логику программы. Если промежуточное состояние "не

применимо", отключение элемента управления — это лучше, чем применение стиля **3STATE**.

В случае использования стиля **BS_AUTOCHECKBOX** Windows заботится об установке и очистке вашего флажка. Если применяется стиль **BS_CHECKBOX**, то эти операции должны выполняться непосредственно пользователем. В обоих случаях можно выбирать режим уведомления, если состояние флажка изменяется, поэтому применение стиля **BS_CHECKBOX** себя не оправдывает.

Конечно, в Visual C++ нет причин волноваться по поводу этих различных стилей. Просто используется диалоговое окно Check Box Properties, которое показано на рис. 6.8. Обратите внимание, что можно использовать элементы, которые выглядят одинаково как в случае флажков, так и в случае кнопок. На рис. 6.9 показаны некоторые стили флажка.

Стили кнопки переключателя CButton

При создании переключателя **CButton** в вашем распоряжении имеется только половина из возможностей выбора, которыми вы наслаждались в случае со стилями флажка: два стиля **BS_RADIOBUTTON** и **BS_AUTORADIOBUTTON**. Однако если вам доступно меньшее количество стилей, то имеется двойная причина использовать **BS_AUTORADIOBUTTON** и сторониться **BS_RADIOBUTTON**.

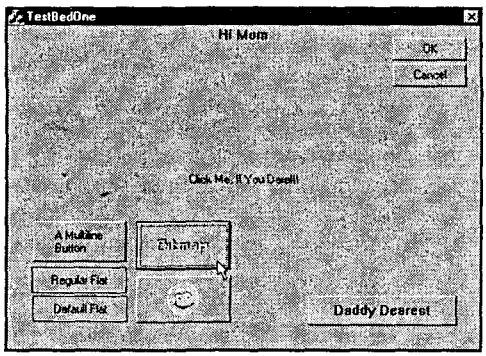


РИСУНОК 6.7 *Отображение различных стилей CButton.*

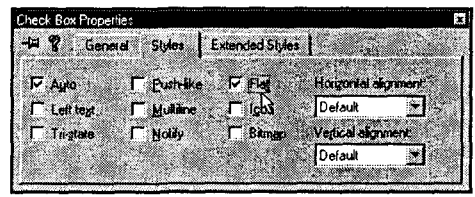


РИСУНОК 6.8 *Использование диалогового окна Check Box Properties.*

Как и в случае с флажками, стиль **AUTO** оставляет установку и снятие переключателей самой Windows. При использовании стиля **BS_RADIOBUTTON** все эти задачи потребуется выполнять самостоятельно. Но самостоятельное выполнение этих задач означает больше, чем текущий контроль одиночного переключателя и его очистка или установка по мере необходимости. Вместо этого придется контролировать целый набор переключателей. Если пользователь щелкает на одном из них, вам нужно циклически пройти по всем другим, чтобы отменить выбор каждого из них, в дополнение к выбору необходимого переключателя. Это достаточно большой объем совершенно ненужной работы.

Для того чтобы сделать Windows волшебником, устанавливающим и очищающим создаваемые вами кнопки **BS_AUTORADIO**, следует изучить еще один стиль: **WS_GROUP**. Windows использует этот стиль для определения принадлежности переключателей специфической группе. В конце концов, вам не понравится, если при выборе конкретного переключателя Windows станет очищать каждый неактив-

ный переключатель во всем диалоговом окне; желательно, чтобы в данном случае очистились только переключатели, относящиеся к определенной группе.

Если **CButtons** создаются программным способом, стиль **WS_GROUP** применяется к первому переключателю в группе переключателей и к первому элементу управления, находящемуся после группы переключателей. Убедитесь в том, что никакой другой переключатель в группе не имеет стиль **WS_GROUP**.

Как обычно в Visual C++, диалоговое окно используется для установки реквизитов переключателей. Выбор стилей, а также корректировка стиля **WS_GROUP** выполняется в диалоговом окне Radio Button Properties, которое показано на рис. 6.10.

На рис. 6.11 продемонстрированы несколько стилей переключателей, доступных для класса **CButton**.

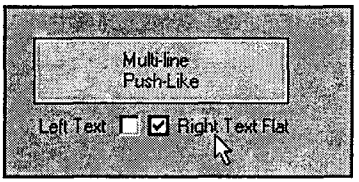


РИСУНОК 6.9 *Различные стили флажков в действии.*

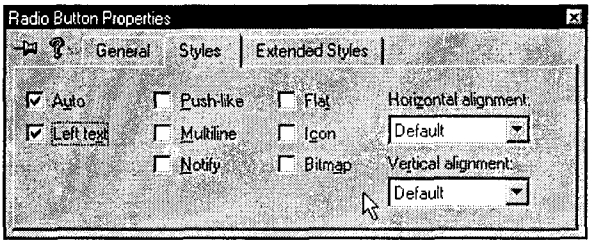


РИСУНОК 6.10 *Использование диалогового окна Radio Button Properties.*

Стили группы CButton

И напоследок рассмотрим еще один стиль: **BS_GROUPBOX**. Ранее уже упоминалось о том, что элемент управления кнопки в стиле **BS_GROUPBOX** не особенно утруждает себя работой. Группа (groupbox) появляется в виде пустого окна с заголовком в левом верхней части.

СОВЕТ

WS_GROUP и табуляторы

Если два переключателя находятся рядом, из этого вовсе не следует, что они принадлежат одной группе. Группирование зависит не от размещения на экране, но от порядка, в котором компоненты добавлялись в диалоговое окно. При создании диалоговых окон можно сбрасывать порядок обхода элементов управления, воспользовавшись опцией Layout | Tab Order в главном меню.

Необходимо всегда выполнять эту операцию перед установкой элементов WS_GROUP, если только элементы управления не добавлялись строго последовательно.

Как и следует из имени, группы обычно группируют переключатели и флажки вместе, формируя визуальное целое. Однако важно помнить, что только по причине помещения переключателей в группу, они не станут автоматически частью группы. Кнопки в стиле Groupbox позволяют визуально группировать элементы управления, но необходимо использовать стиль WS_GROUP для логического группирования ваших переключателей и флажков.

Подобно всем остальным объектам в Visual C++, даже самая непритязательная группа имеет собственное диалоговое окно. Диалоговое окно Group Box Properties показано на рис. 6.12.

Работа с элементами управления CButton

Как вы уже видели в приложении FourUp, работа с кнопками не является трудной, особенно, если воспользоваться услугами ClassWizard. Когда пользователь щелкает на кнопке (отличной от кнопки группы), генерируется событие **BN_CLICKED**. (Более точно, это не событие, а *уведомление* — своего рода событие, сгенерированное элементом управлением. В данный момент времени это различие несущественно.)

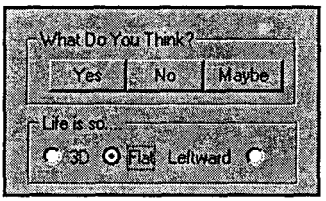


РИСУНОК 6.11 *Стили переключателей, доступные для класса CButton.*

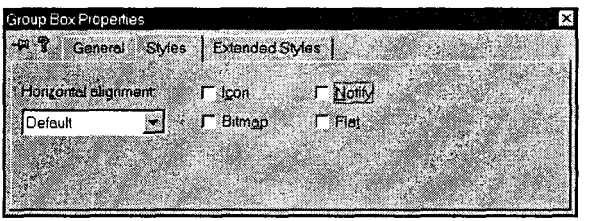


РИСУНОК 6.12 *Диалоговое окно Group Box Properties.*

В Visual C++ можно создать метод обработки события **BN_CLICKED**, просто дважды щелкнув на кнопке в Dialog Editor. ClassWizard откроет диалоговое окно Add Member Function и предложит создать новый метод. При этом даже поддерживается заданное по умолчанию имя. (Как было показано, имя записывается в форме **OnCtrlName()**). Все, что нужно добавить — это текст кода.

Если при создании кнопки используется стиль **BS_NOTIFY** (либо выполняется щелчок на флажке Notify в Developer Studio), кнопка генерирует три дополнительных события:

- **BN_DOUBLECLICKED**, когда на кнопке был произведен двойной щелчок.
- **BN_SETFOCUS**, когда пользователь переходит к кнопке при помощи клавиши Tab или ее выборе мышью.
- **BN_KILLFOCUS**, когда клавиатурный фокус покидает элемент управления, поскольку пользователь выбирает другой элемент управление при помощи клавиатуры или мыши.

Методы CButton

CButton не только уведомляют вас о происходящем, они также имеют несколько методов, которые могут оказаться полезными. Обычно наиболее часто используются методы **GetCheck()** и **SetCheck()**. Последние применяются прежде всего для групп переключателей и флажков.

Метод **SetCheck()** принимает один параметр, либо **BST_CHECKED**, либо **BST_UNCHECKED**.

(Флажки **3STATE** также могут принимать значение параметра **BST_INDETERMINATE**.) Чтобы выяснить, установлен ли флажок, можно использовать функцию **GetCheck()**, которая возвращает те же самые значения. Например, чтобы принять участие в установке пользователем элемента управления флажком **m_check**, можно записать следующую строку:

```
if (m_check.GetCheck() == BST_CHECKED) {...}
```

Если вы не хотите выполнять специфическое действие, когда выбран или не выбран флажок, то обычно не нужно отвечать на уведомления, полученные от флажков. Однако переключатели являются другим предметом. Для определения того, какой переключатель выбран, используется метод **GetCheck()** для каждого индивидуального переключателя. Невозможно непосредственно обнаружить, какой переключатель выбран из всех переключателей в группе. По этой причине для переключателей часто приходится перехватывать сообщение **BN_CLICKED**.

Заключительное слово

В этой главе вы получили множество информации, касающейся окон, а также элементов управления **CStatic** и **CButton**. Не волнуйтесь, если не сможете запомнить такой объем информации — просто разберитесь в ней. Воспользуйтесь оперативной документацией Visual C++ для получения информации относительно конструкторов, методов и их параметров. Продолжайте сосредоточивать внимание на "большой картине", и вы преуспеете.

В следующей главе начинает рассматриваться новый проект — программа рисования. Рассмотрение этого вопроса займет несколько глав. Однако программа с ограниченной функциональностью будет разработана сразу же. При работе с проектом вы изучите интерфейс графических устройств Windows (GDI — Graphical Device Interface) и порядок его использования для рисования линий, форм и отображения текста.

Глава 7

Компьютерная графика: создание графического приложения

Мастера граффити рисуют на стенах, а программисты — в окнах.

Популярная наклейка на бамперах в 1960-тых годах гласила: "Будь человеком: не сгибай, не скручивай, не искажай" (Human Being: Do Not Fold, Spindle, Or Mutilate). Эти слова были напечатаны на лицевой стороне вездесущих перфокарт IBM, в те времена знакомых практически каждому американцу. Результаты правительственных проверок, сервисные счета и даже заметки относительно просроченных библиотечных книг находились на аналогичных серых картах.

Более чем что-либо еще, перфокарты воплощают безличный компьютер и настойчивость, с которой люди пытаются подчинить свою жизнь внешним требованиям, а не наоборот. И, люди, естественно восстали против такого положения дел. Революция в лице персонального компьютера победила силы тирании и темноты, и красочное, дружественное программное обеспечение воцарилось во всей стране. Более или менее.

В свое время Джордж Оруэлл (George Orwell) отметил в своей новелле *Звероферма*, что чем больше происходит изменений, тем дольше все остается неизменным. Перфокарты ушли в прошлое, но автоматизированные формы их духовных потомков живы до сих пор. Благодаря программным продуктам, подобным Visual Basic, были созданы компьютерные приложения, основанные на формах, что свело количество необходимой работы к минимуму, и в результате (к восхищению *лишенных воображения бюрократов, соривших перфокартами во всем мире*), почти идентичные формы распространены повсюду. Однако несмотря на то что нам скорее всего придется воспринять эти приложения, основанные на формах, как "хлеб и масло" разработки программного обеспечения Windows, хотя можно найти творческую отдушину благодаря душе программирования — *графическим приложениям*.

Возникает вопрос, что такое графическое приложение? Популярная игра для PC "Myst" — это графическое приложение, как и большинство игр, разработанных, начиная с момента упадка текстовых приключенческих игр. Adobe Photoshop и Windows Paint — это графические приложения. Таковыми же являются Lotus Freelance и Microsoft PowerPoint. Хранители экрана, настольные издательские системы, текстовые процессоры WYSIWYG и системы электронных таблиц — все это примеры графических приложений. Свойства графических приложений зависят от познаний программиста, создающего код, рисующего изображение на экран — то ли это захватывающие трехмерные лабиринты Doom или более мирные гистограммы 1-2-3.

В этой главе вы научитесь рисованию на компьютерном экране. Эта задача является намного более трудной, чем размещение меток или кнопок. Для облегчения процесса обучения не будем торопиться, начнем с рисования некоторых простых строк и форм.

И так, возьмите берет, блузу и мольберт и приготовьтесь немного развлечься.

Графика в одной линии

По традиции первая текстовая программа, созданная программистами, изучающими любой язык программирования — это приложение "Hello World", которое выводит вышеупомянутые слова на компьютерном дисплее. Когда программисты перешли от текстовых к графическим программам, традиция "Hello World" начала отмирать. Единственная проблема с этой специфической традицией заключается в том, что она является назойливой. Простое рисование набора произвольных строк было бы более интересным. Так, с намерением быть *немного* более интересными, приступим к делу!

Проект LineOne

Программа LineOne представляет собой приложение, основанное на диалоговых окнах, которое рисует на экране 50 линий случайного размера. Несмотря на простоту, программа LineOne представляет сгусток фундаментальных понятий, которыми потребуется овладеть перед переходом к более сложным графическим приложениям. Если это поможет, то думайте о LineOne как о "Компьютерном рисовании 100". Для дальнейшей работы необходимо правильно ответить на следующие вопросы:

- Что такое *контекст устройства* и для чего он используется?
- Каков характер сообщения **WM_PAINT** и как оно обрабатывается?
- Какие используются функции для рисования простых строк на экране?

Формирование каркаса проекта

Для начала работы над проектом LineOne используйте AppWizard, который создаст каркас будущей программы. Выполните следующие шаги:

1. Используйте команду File | Close Workspace (Файл | Закрыть рабочую область) для закрытия любых открытых проектов. Щелкните на Yes (Да) при выдаче запроса относительно закрытия всех окон документов.
2. Выберите команду File | New (Файл | Открыть) для открытия диалогового окна New. Выберите закладку Projects (Проекты) из списка доступных страниц, затем выберите пункт MFC AppWizard (exe) из списка типов проектов. Назовите свой проект LineOne. Отображаемый экран должен походить на показанный на рис. 7.1.

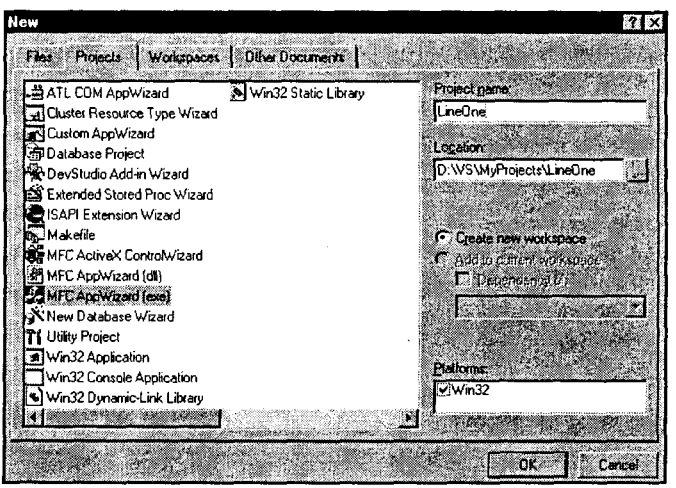


РИСУНОК 7.1
Начало работы с новым приложением LineOne.

3. Щелкните на кнопке OK для запуска AppWizard. В диалоговом окне MFC AppWizard — Step 1 выберите пункт Dialog-based Application и щелкните на кнопке Next (Далее).
4. В диалоговом окне MFC AppWizard — Step 2 отмените установку всех флажков. Оставьте LineOne в качестве заголовка. Диалоговое окно должно похо-

дить на изображенное на рис. 7.2. Теперь щелкните на кнопке Finish. По открытию диалогового окна New Project Information (Информация о новом проекте) щелкните на кнопке OK.

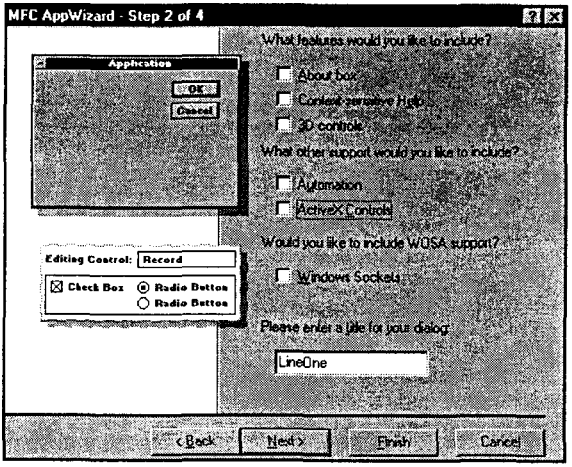


РИСУНОК 7.2
Опции MFC AppWizard Step 2.

5. В Dialog Editor, который запускается автоматически, выберите и удалите метку TODO: label, равно как и кнопки OK и Cancel. Ваше диалоговое окно не должно содержать никаких элементов управления, а экран должен походить на показанный на рис. 7.3.

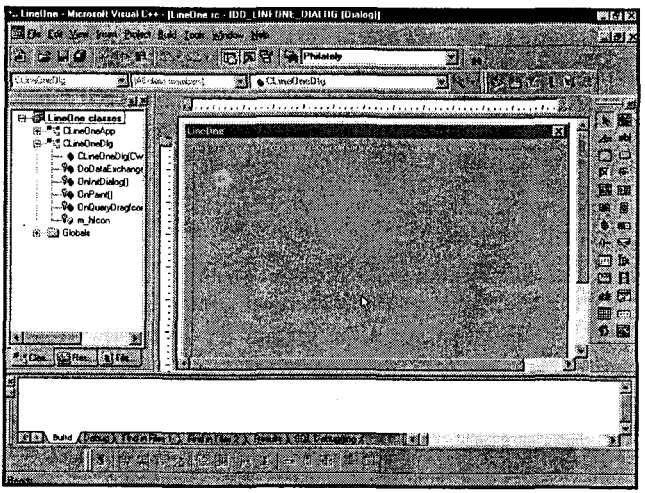


РИСУНОК 7.3
Удаление ненужных элементов управления диалогового окна.

6. Скомпилируйте и запустите на выполнение приложение, дабы убедиться, что этот "каркас" работает. По завершению исследования полученных весьма ограниченных возможностей закройте приложение, выполнив щелчок на поле Close в заголовке.

Добавление нескольких линий

В ходе наблюдения за работой базового приложения LineOne вы, наверное, заметили, что в нем уже присутствует некоторый код рисования. В конце концов, оно выглядит и ведет себя подобно простому, серому холсту. Однако простой, серый холст — именно то место, где вы собираетесь рисовать линии.

Для добавления собственных рисунков в LineOne выполните следующим шагам:

1. Выберите панель ClassView в окне Project Workspace. Расширьте класс **CLineOneDlg** и дважды щелкните на методе **OnPaint()**, который находится там. Здесь будет помещен код вычерчивания линий.
2. Обратите внимание, что метод **OnPaint()** уже включает небольшой объем кода. Если вы ознакомитесь с комментариями, вставленными AppWizard, вы увидите, что этот код просто рисует пиктограмму, соответствующую минимизированному приложению. Поскольку в приложении нет кнопки Minimize, код, описывающий эту кнопку, является лишним. Удалите код **OnPaint()** и замените его кодом, приведенным в листинге 7.1.

Листинг 7.1 Метод CLineOneDlg::OnPaint().

```
void CLineOneDlg::OnPaint()
{
    // 1. Создание контекста устройства для экрана
    CPaintDC dc(this);

    // 2. Измерение клиентской области
    CRect rect;
    GetClientRect(&rect);

    // 3. Рисование 50 линий
    for (int line = 0; line < 50; line++)
    {
        // 4. Позиционирование начала линии
        dc.MoveTo(rand() % rect.right, rand() % rect.bottom);

        // 5. Рисование до конечной точки
        dc.LineTo(rand() % rect.right, rand() % rect.bottom);
    }
}
```

3. Скомпилируйте и запустите на выполнение приложение. О! Вы достаточно искусны! Результаты вашей работы должны выглядеть как на рис. 7.4.

Быстрый взгляд внутрь LineOne

Программа LineOne не содержит большого количества кода. Конечно, если вы не знаете, что выполняют эти строки кода, или то, что они собой представляют, то наличие только пяти или шести строк кода не принесет облегчения. На самом деле наиболее тяжелые загадки часто самые короткие.

К счастью, в этом случае ситуация не столь грустная. Точно так же как исследование камня Розетты открыло значение древних египетских иероглифов, функция **OnPaint()** из LineOne открывает секреты интерфейса графических устройств Windows, или GDI — Graphical Device Interface. Если только вы поймете, как работает GDI, то все остальное — лишь простая "компиляция".

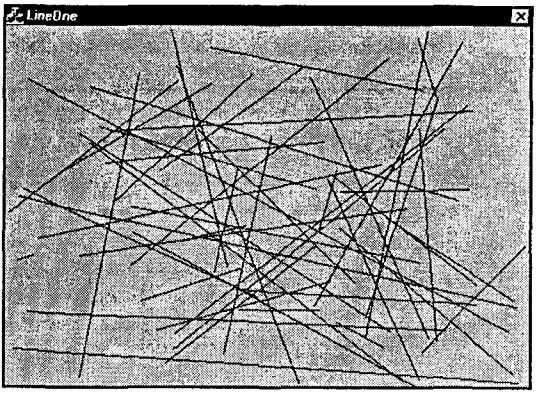


РИСУНОК 7.4
Результат выполнения приложения LineOne.

Для облегчения начала работы объяснение разбивается на две части. Сначала будет рассмотрена каждая строка в функции **OnPaint()**, сопровождаемая комментариями. Затем в следующей главе будет предпринята попытка более глубокого исследования GDI.

Что такое OnPaint()?

В Windows каждое окно разделено на две части: неклиентскую область (которая включает название окна и его границы) и клиентскую область (область "между строк", как принято говорить). Иллюстрацию этих понятий можно видеть на рис. 7.5.

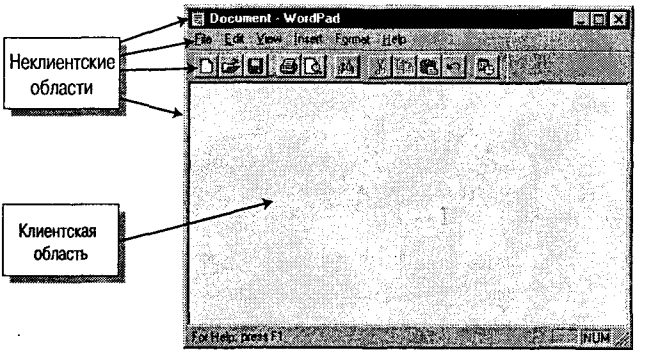


РИСУНОК 7.5 *Клиентские и не клиентские области окна.*

Windows рисует в неклиентской области; при этом нет необходимости предпринимать что-либо специальное. С другой стороны, вы отвечаете за рисование внутри клиентской области. Всякий раз когда Windows желает окрашивать клиентскую область окна, она посылает окну сообщение **WM_PAINT**. Для обработки этого сообщения обычно программируется метод **OnPaint()**.

Windows посылает сообщение **WM_PAINT** при первом отображении окна; это сообщение также посылается в том случае, когда часть клиентской области окна открывается из-под другого окна. В любом случае, когда окно получает сообщение **WM_PAINT**, должна быть осуществлена подготовка для выполнения перерисовки его клиентской области.

Как вы понимаете, код, написанный для отображения файла в формате GIF, лишь немного отличается от кода, отображающего гистограмму. Несмотря на подобные различия, большая часть кода в **OnPaint()** выполняет следующих три задачи:

1. Получение холста или поверхности рисования. В Windows используемая поверхность рисования называется *контекстом устройства.*
2. Установка среды. Это понятие включает сбор всех требуемых перьев и кистей, а также измерение рабочей поверхности, что позволит правильно распределить изображения по поверхности.
3. Окрашивание окна при помощи функции, доступных в графической библиотеке Windows, GDI.

Давайте рассмотрим LineOne и попытаемся идентифицировать каждый из этих шагов.

Получение холста

Точно как и в реальной жизни, если вы собираетесь рисовать, то необходима поверхность, на которой выполняется собственно рисование. Windows-эквивалент настоящего холста живописца — это *контекст устройства.* Перед созданием любого вывода в Windows необходимо получить контекст устройства.

Функция **OnPaint()** из LineOne использует класс **CPaintDC** для создания объекта контекста устройства, именуемого **dc**. Параметр, передаваемый конструктору **CPaintDC** — это указатель на окно, в котором будет происходить рисование. Поскольку LineOne рисует в главном окне, то передается параметр **this**.

```
// Создать контекст устройства для экрана
CPaintDC dc(this);
```

Далее будет показано, что класс **CPaintDC** — только один из нескольких классов, которые можно использовать для создания контекста устройства. Класс **CPaintDC** используется только в методе **OnPaint()**, и он обеспечивает вас холстом, установленным по размеру клиентской области окна.

Подготовка среды

Для гарантирования того, что нарисованные линии появляются внутри клиентской области (где они будут видимы), программа LineOne должна измерить клиентскую область. Чтобы зафиксировать размерности клиентской области, создается экземпляр **rect** класса **CRect**:

```
CRect rect;
```

Затем экземпляр вызывает функцию **GetClientRect()**:

```
GetClientRect(&rect);
```

Функция **GetClientRect()** заполняет **rect** размерностями клиентской области, сохраняя их в общедоступных элементах данных **top**, **left**, **bottom** и **right**.

Рисование линий

Поскольку программа LineOne рисует 50 линий, требуется, чтобы функция **OnPaint()** выполнялась в цикле. Каждая итерация цикла рисует новую линию.

Функции вычерчивания линий Windows рисуют, начиная с текущей позиции "пера" GDI по направлению к определенной позиции. Следовательно, рисование линии в Windows — это двухступенчатый процесс. Сначала вызывается функция **MoveTo()**, которая перемещает перо в местоположение, откуда начнется процесс

рисования. Программа LineOne использует функцию **MoveTo()** следующим образом:

```
// Начать линию в произвольной позиции
Dc.MoveTo(rand() % rect.right, rand() % rect.bottom);
```

LineOne использует функцию **rand()** и значения полей **rect.right** и **rect.bottom** для гарантирования того, что каждая линия начинается где-то в клиентской области окна.

На втором шаге двухступенчатого процесса рисования линии вызывается метод контекста устройства **LineTo()**. Когда вызывается **LineTo()**, передается конечная позиция линии. **LineTo()** рисует линию между текущей позицией пера и новой конечной позицией, которую вы определяете. В качестве побочного эффекта, эта окончательная позиция становится новой текущей позицией пера.

Подобно **MoveTo()**, функция **LineTo()** использует **rand()** для генерации случайной конечной точки, которая попадает внутрь клиентской области. Ниже приведен код вычерчивания линии:

```
// Рисование в направлении второй точки, расположенной случайным
   образом
Dc.LineTo(rand() % rect.right, rand() % rect.bottom);
```

Поскольку функция **LineTo()** обновляет текущую позицию пера, на самом деле нет необходимости в вызове функции **MoveTo()** перед вызовом **LineTo()**. Всякий раз создается новый контекст устройства, текущая позиция пера, заданная по умолчанию, устанавливается в значение 0,0 — размещенное в левом верхнем углу клиентской области. Если линии создаются без использования функции **MoveTo()** в приложении LineOne, то каждая новая линия начинается в точке окончания предыдущей линии (см. рис. 7.6).

СОВЕТ

Включаемый/исключаемый, или что происходит с последним пикселом?

Начинающие программисты в среде Windows часто удивляются и досадуют, когда линия, проведенная из точки с координатами (0,0) в точку с координатами (100,100) не окрашивает пиксел по координатам (100,100). При рисовании линий Windows использует метод под названием включаемый/исключаемый (inclusive/exclusive) — окрашивается начальный пиксел линии и не окрашивается конечный пиксел линии. Поскольку функция LineTo() переустанавливает текущую позицию пера контекста устройства, исключая конечную позицию, убедитесь в том, что серия вызовов LineTo() приводит к однократному окрашиванию каждого пиксела.

Парадокс с LineTwo

Прежде чем оставить в покое рисование линий, давайте рассмотрим другой короткий пример программы, которая использует повторяющиеся вызовы **LineTo()** без использования вызовов **MoveTo()**. Приложение LineTwo работает практически так же, как и LineOne, только вместо 50 произвольных линий рисуется серия ромбов, каждый из которых немного сдвинут и повернут относительно предыдущего. В результате получается довольно замысловатая конструкция.

Формирование LineTwo

Инструкции для построения LineTwo весьма похожи на те, которые использовались для LineOne:

1. Используйте MFC AppWizard для создания нового приложения под названием LineTwo.
2. Выберите пункт Dialog-Based Application из диалогового окно MFC AppWizard Step 1 и щелкните на кнопке Next.
3. Снимите отметки со всех флажков в диалоговом окне MFC AppWizard Step 2, а затем щелкните на Finish. Щелкните на кнопке OK при появлении окна New Project Information.
4. Удалите из главного диалогового окна все элементы управления.
5. Откройте диалоговое окно Dialog Properties (Свойства диалогового окна), щелкнув правой кнопкой мыши на главном диалоговом окне в Dialog Editor. Выберите закладку Styles, затем выберите пункт Resizing (Изменение размера) из раскрывающегося списка Border (Рамка). Эта операция проиллюстрирована на рис. 7.7.

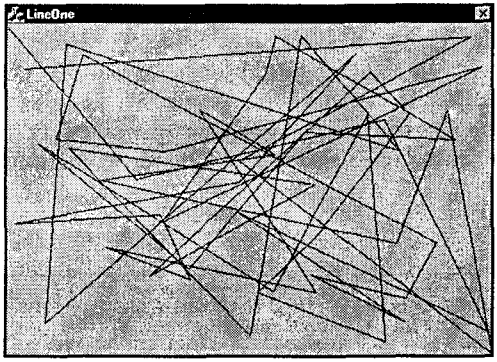


РИСУНОК 7.6 *Запуск приложения LineOne без использования функции MoveTo().*

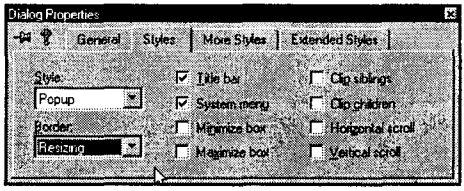


РИСУНОК 7.7 *Установка свойств диалогового окна LineTwo.*

6. Выберите панель ClassView в окне Project Workspace. Расширьте класс **CLineTwoDlg** и дважды щелкните на функции **OnPaint()**. Замените находящийся там код выделенным кодом из листинга 7.2.

Листинг 7.2 Метод CLineTwoDlg::OnPaint().

```
void CLineTwoDlg::OnPaint()
{
        // Рисование линий при помощи LineTo
        // 1. Получить контекст устройства клиентской области
        CPaintDC dc(this);

        // 2. Измерить клиентскую область
        CRect rect;
        GetClientRect(&rect);

        // 3. Начать рисование с точки, находящейся вверху в центре
        dc.MoveTo(rect.right / 2, 0);
```

```
    // 4. Рисование 50 ромбов, со смещением относительно друг друга
      for (int lines = 0; lines < 50; lines++)
      {
            Dc.LineTo(rect.right - lines, rect.bottom / 2 + lines);
            Dc.LineTo(rect.right / 2 - lines, rect.bottom - lines);
            dc.LineTo(lines, rect.bottom / 2 - lines);
            dc.LineTo(rect.right / 2 + lines, lines);
      }
}
```

7. Скомпилируйте и запустите на выполнение приложение. Результаты выполнения показаны на рис. 7.8.

Как работает LineTwo?

Как и в случае с LineOne, все действия в LineTwo происходят в методе **OnPaint()**. Приложение LineOne использовало размерности окна клиентской области (сохраненные в объекте **rect** из **CRect**) для гарантирования того, что каждая линия попадает внутрь клиентской области. Приложение LineTwo использует те же самые размерности для вычисления новой позиции окончания для каждого набора четырех линий, составляющих один ромб.

Вот как это работает:

- Сначала (на шаге 3 в листинге 7.2) LineTwo перемещает текущую позицию пера в центр верхней строки. Координаты центра вычисляются по формуле **rect.right / 2**, при этом верхняя строка — 0. Эти значения передаются в функцию **MoveTo()**.
- Затем LineTwo использует тот же самый цикл, который применялся в LineOne. Цикл выполняется 50 раз с использованием локальной переменной **lines**, принимающей значения от 0 до 49.
- Цикл включает четыре вызова **LineTo()**. Первый вызов рисует в направлении от середины верхней строки к середине правого столбца, второй вызов рисует от середины правого столбца к середине нижней строки, третий — от середины нижней строки к середине левого столбца, а последний — обратно к середине верхней строки.
- Каждый раз при прохождении итерации цикла отметка конца каждой линии сдвигается по часовой стрелке на величину, сохраненную в переменной **lines**. Во время первой итерации цикла величина **lines** равна 0, поэтому концы каждой линии находятся посередине — никакого смещения нет. На второй (и каждой последующей) итерации происходят смещения конечной точки. Например, правые вершины ромбов смещаются вправо на один пиксел и вниз на один пиксел, и со временем LineTwo достигает такой строки кода:

```
dc.LineTo(rect.right - lines, rect.bottom / 2 + lines);
```

Точно такое же преобразование происходит с каждой из остающихся вершин.

Чего не делает LineTwo?

Теперь наступило время для проведения небольшого эксперимента. Когда формировалось приложение LineTwo, стиль рамки диалогового окна был изменен на

Resizing. Давайте рассмотрим, что происходит при изменении размеров окна программы. Для выполнения этого эксперимента переместите диалоговое окно в левый верхний угол экрана, а затем тяните правый угол диалогового окна до тех, пока оно практически не заполнит весь экран. Когда вы оставите в покое рамку, вызывается функция **OnPaint()** из LineTwo и быстро повторно окрашивает диалоговое окно, как показано на рис. 7.9.

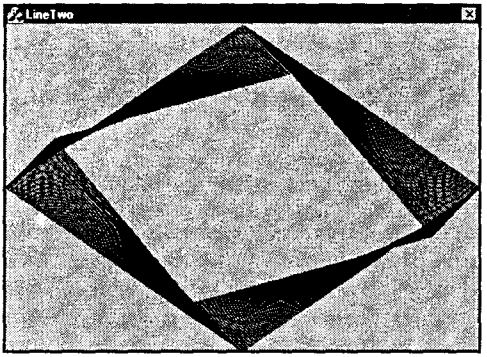


РИСУНОК 7.8 *Результаты работы приложения LineTwo.*

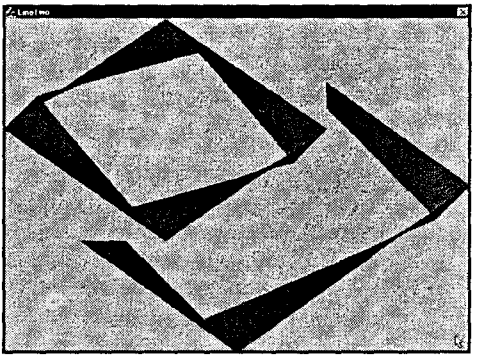


РИСУНОК 7.9 *Изменение размеров приложения LineTwo.*

Хм. Это не выглядит слишком красиво, не так ли? Вместо того чтобы перерисовывать все окно, метод **OnPaint()** перерисовывает только новые области, даже при том, что **OnPaint()** перерисовывает каждую линию. В главе 8 будет детально объяснено, как и почему это происходит. Здесь же приведем лишь краткие пояснения.

Ваш метод **OnPaint()** вызывается *только* Windows, и только тогда, когда Windows считает, что часть вашего окна должна быть перерисована. Когда создается объект **CPaintDC** в обработчике **OnPaint()** (не забывайте, что это единственное место, где можно использовать объект **CPaintDC**), Windows услужливо предоставляет вам *усеченный* контекст устройства.

Усеченный контекст устройства позволяет окрашивать только часть вашего окна — Windows чувствует характер изменения потребностей. Это похоже на то, как будто Windows вручает вам холст, но сначала размещает на нем трафарет. Независимо от того, где применяется окрашивание, краска ложится на холст только через отверстия трафарета.

Очевидно, что поведение Windows не соответствует задачам нашего приложения. Существует ли способ сообщить Windows, что нужно повторно окрашивать целое окно? Да, существует, — просто вызовите функцию **CWnd::Invalidate()**. Эта функция сообщает Windows, что должно быть окрашено целое окно, а не его часть.

Поскольку функция **OnPaint()** из LineTwo подсчитывает линии, основываясь на размере окна, то нужно полностью перерисовывать экран всякий раз, когда размер окна изменяется. Давайте сейчас выполним необходимые изменения в LineTwo.

Добавление обработчика сообщений Windows при помощи ClassWizard

Добавление кода рисования в приложения LineOne и LineTwo оказалось несложным, поскольку метод **OnPaint()** уже был; все, что нужно было добавить — это новый код. Однако если открыть проект LineTwo и просмотреть панель ClassView в окне Project Workspace, там не будут присутствовать методы, изменяющие размеры. Потребуется добавить собственные методы.

При изменении размеров окна Windows посылает сообщение **WM_SIZE**, сообщающее окну новые размеры. (Это происходит после завершения изменение размеров, а не непрерывно.) Для обработки сообщения **WM_SIZE** добавляется обработчик сообщений Windows при помощи ClassWizard. Для этого необходимо проделать следующее:

1. Откройте проект LineTwo и выберите панель ClassView в окне Project Workspace. Найдите класс **CLineTwoDlg** и щелкните правой кнопкой мыши на имени класса.
2. Из появившегося контекстного меню выберите пункт Add Windows Message Handler (Добавить обработчик сообщений Windows).
3. Открывается диалоговое окно с громоздким заголовком "New Windows Message And Event Handlers For Class CLineTwoDlg" (Новые обработчики сообщений Windows и событий для класса CLineTwoDlg). Из списка New Windows Messages/Events выберите **WM_SIZE**, а затем щелкните на элементе Add And Edit. (Это диалоговое окно можно видеть на рис. 7.10.)
4. Добавьте выделенный код из листинга 7.3 к новому методу **OnSize()**, затем повторно откомпилируйте программу. После запуска программы вы увидите, что теперь выполняется корректное окрашивание при каждом изменении размеров главного окна.

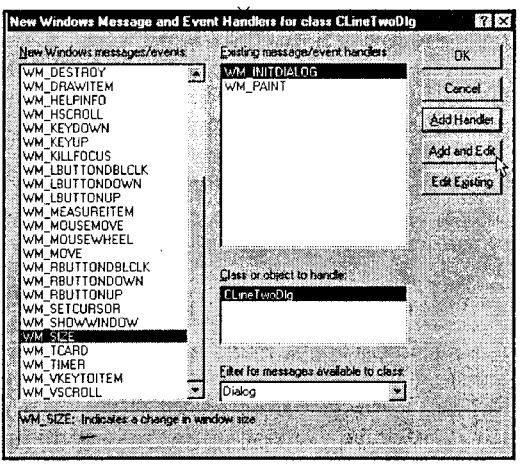


РИСУНОК 7.10 *Диалоговое окно New Windows Message And Event Handlers.*

Листинг 7.3 Метод CLineTwoDlg::OnSize().

```
void CLineTwoDlg::OnSize(UINT nType, int cx, int cy)
{
      CDialog::OnSize(nType, cx, cy);

      // ЧТО СДЕЛАТЬ: Добавьте здесь код обработчика
      // сообщений
      Invalidate();
}
```

Рисование изображений в Windows

Теперь, когда вы уже знаете, как рисовать базовые линии, давайте расширим набор, изучив рисование замкнутых фигур — простых форм, подобных прямоугольникам, кругам, эллипсам и многоугольникам.

Замкнутые фигуры отличаются от линий двумя характеристиками:

- При рисовании замкнутых фигур используется рисующее перо, точно так же, как и в случае линий, но при этом также используется "кисть" для закрашивания цветом внутренностей фигур. Позже в главе можно увидеть, как изменяются встроенное перо и кисть, используемые для рисования линий и форм.
- Большинство замкнутых фигур рисуются при помощи мнимого прямоугольника, называемого *ограничивающим прямоугольником.*

Давайте начнем с рассмотрения работы ограничивающего прямоугольника. Например, для рисования прямоугольника используется функция **Rectangle()** контекста устройства:

```
CPaintDC dc(this); // Получить контекст устройства
dc.Rectangle(0, 0, 50, 50);
```

Четыре параметра, передаваемые функции **Rectangle()**, представляют координаты верхнего левого и нижнего правого углов ограничивающего прямоугольника. Если рисуется прямоугольник, используется текущее перо, рисующее четыре линии, начинающиеся в позиции 0,0 и идущие вверх, но не включающие строку и столбец с номером 50. (При рисовании замкнутых фигур Windows использует ту же самую философию включения/исключения, которая имела место для случая простых линий.)

Область, заключенная между этими линиями, закрашивается при помощи текущей кисти контекста устройства. По умолчанию Windows использует перо с 1-пиксельной шириной для рисования контура рисунка и сплошную белую кисть для окрашивания внутренних частей форм.

Рисование фигур: SquaresAndCircles

Давайте начнем наше исследование рисования фигур, рисуя некоторые произвольные квадраты и круги. (Фактически будут рисоваться прямоугольники и эллипсы, но SquaresAndCircles (Квадраты и круги) в качестве имени программы звучит лучше.) Будем придерживаться базовых схем действий, используемых в LineOne, с которыми вы уже ознакомились к настоящему времени:

- Получение контекста устройства при помощи класса **CPaintDC**.
- Измерение клиентской области.
- Использование цикла для рисования 50 прямоугольников и эллипсов, имеющих случайные размеры. Приложение SquaresAndCircles использует ту же самую логику, которая применялась в LineOne для обеспечения того, чтобы каждый рисунок полностью попадал внутрь клиентской области окна.

Далее руководствуйтесь следующими инструкциями:

1. Создайте приложение, основанное на диалоговых окнах, под именем SquaresAndCircles, используя AppWizard, точно так же, как это было в случае с LineOne. Не забудьте отменить установку всех опций в диалоговом окне MFC AppWizard Step 2.

2. Удалите все элементы управления из главного диалогового окна в Dialog Editor. Не изменяйте на Resizing стиль рамки диалогового окна.
3. Найдите метод **CSquaresAndCirclesDlg::OnPaint()** в окне Project Workspace ClassView.

Читайте документацию, но держите порох сухим

Несмотря на огромное количество документации, доступной для Windows API и MFC, невозможно всецело полагаться на нее в плане достоверности или полноты информации. Функция CDC::Rectangle(), наряду со всеми другими графическими функциями, использующими ограничивающий прямоугольник, в документации описана не вполне корректно.

В документации, поставляемой вместе с Visual C++, присутствует следующий прототип функции Rectangle().

```
BOOL Rectangle( int x1, int y1, int x2, int y2);
```

Документация идентифицирует x1, y1 как местоположение левого верхнего угла, а x2, y2 — как местоположение правого нижнего угла. Однако фактически координаты x1, y1 не представляют левый верхний угол прямоугольника — эта точка с таким же успехом может быть правым нижним, левым нижним или правым верхним углом. Вы просто передаете два набора координат, и MFC рисует прямоугольник, связанный с ними. Нет необходимости тратить время, убеждаясь в том, что точка x1, y1 лежит левее и выше x2, y2, поскольку так и будет, если x1, y1 является на *самом деле* левым верхним углом.

Мораль сей басни такова — читайте документацию, но не полагайтесь на нее во всем. Напишите программу и посмотрите, как на самом деле работает MFC.

Дважды щелкните мышью для открытия функции, затем замените существующий код на код из листинга 7.4.

Листинг 7.4 Метод CSquaresAndCirclesDlg::OnPaint().

```
void CSquaresAndCirclesDlg::OnPaint()
{
      // Рисование фигур с использованием функций Rectangle() и
      // Ellipse()
      // 1. Получение контекста устройства клиентской области
      CPaintDC dc(this);

      // 2. Измерение клиентской области
      CRect rect;
      GetClientRect(&rect);

      // 3. Рисование 50 прямоугольников и 50 эллипсов
      int middle = rect.right / 2;
      for (int shapes = 0; shapes < 50; shapes++)
      {
            dc.Rectangle(rand() % middle, rand() % rect.bottom,
                         rand() % middle, rand() % rect.bottom);
            dc.Ellipse(middle + rand() % middle, rand() %
  rect.bottom,
                       middle + rand() % middle, rand() %
  rect.bottom);
      }
}
```

4. Откомпилируйте и запустите на выполнение программу. На рис. 7.11 показана выполняющаяся программа SquaresAndCircles.

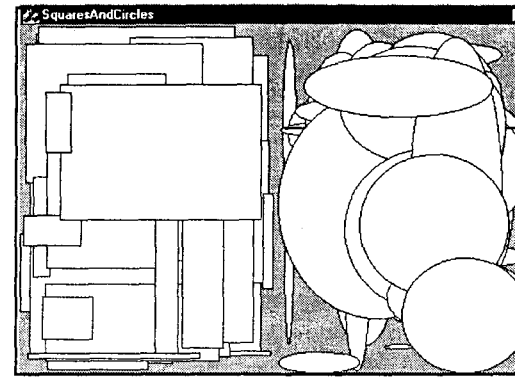


РИСУНОК 7.11 *Выполнение программы SquaresAndCircles.*

Внутри SquaresAndCircles

Программа SquaresAndCircles не очень сильно отличается от LineOne. Основные различия (выделенные в листинге 7.4) таковы:

- Программа рисует прямоугольники слева и эллипсы справа, не смешивая их. Для осуществления подобной методики создается локальная переменная **middle** и инициализируется значением **rect.right/2**.
- На каждой итерации цикла рисуется один прямоугольник и один эллипс. Подобно LineOne, SquaresAndCircles использует случайные числа для вычисления границ замкнутой фигуры. Когда программа вызывает функцию **Rectangle()**, оба горизонтальных параметра (**x1** и **x2**) вычисляются модулем **middle**, что гарантирует их попадание в левую половину диалогового окна. Когда программа вызывает **Ellipse()**, горизонтальные параметры также вычисляются модулем **middle** для гарантирования того, что каждый эллипс выводится в правой половине диалогового окна.

Инструменты рисования

Наверное, вы обратили внимание, что при работе приложения LineOne (рис. 7.4) все линии видимы, но многие из фигур в SquaresAndCircles невидимы (рис. 7.11). Вместо этого появляются какие-то непонятные изображения сверху, нераспознаваемые из-за непрозрачного белого цвета, который заполняет каждую фигуру.

Windows вызывает цвет, используемый для рисования линии или контура фигуры при помощи *пера(pen)*. Цвет, используемый для окрашивания фигуры или фона окна, называется *кистью(brush)*. Windows поддерживает несколько встроенных перьев и кистей, называемых *набором объектов (stock objects)*; давайте рассмотрим порядок их использования. (Windows позволяет определять пользовательские кисти и перья — они рассматриваются в главе 8.)

Наборы перьев и кистей

Windows поддерживает три встроенных пера, которые могут использоваться для рисования контуров фигур или линий. Каждое из перьев набора имеет ширину в 1 пиксел.

Перо по умолчанию, которое до сих пор использовалось, называется **BLACK_PEN**. Если вы хотите рисовать белые линии, то можете использовать перо **WHITE_PEN**. Третье перо, **NULL_PEN** (которое не рисует ничего), на первый взгляд может показаться ненужным — в конце концов, зачем создавать себе дополнительные проблемы, рисуя невидимые линии?

Перо **NULL_PEN** может оказаться малопригодным для рисования линий, но оно вступает в игру, когда необходимо рисовать сплошные окрашенные фигуры. Всякий раз при использовании функции **Rectangle()** или **Ellipse()** рисуется окрашенная фигура с контуром. Если вы хотите получить окрашенную фигуру без окружающего контура, используйте встроенное перо **NULL_PEN**.

В дополнение к набору из трех перьев Windows поддерживает набор из семи кистей. Они называются мнемоническими именами: **BLACK_BRUSH**, **DKGRAY_BRUSH**, **GRAY_BRUSH**, **HOLLOW_BRUSH**, **LTGRAY_BRUSH**, **NULL_BRUSH** и **WHITE_BRUSH**.

И **NULL_BRUSH**, и **HOLLOW_BRUSH** по существу делают то же самое: они представляют собой кисть, которая ничего не делает. Эти кисти используются при рисовании *неокрашенных фигур*.

Использование наборов перьев и кистей

Использование наборов перьев или кистей весьма просто. Когда рисуется линия или форма, используется метод класса контекста устройства, типа **LineTo()** или **Rectangle()**. Вы не можете рисовать линию или прямоугольник без первоначального получения объекта контекста устройства.

Затем объект контекста устройства использует текущее перо и кисть для рисования требуемой линии или формы. Чтобы рисовать различными пером или кистью, необходимо просто сообщить контексту устройства, какое перо или кисть вы желаете, обращаясь к методу **SelectStockObject()** контекста устройства.

Для примера ниже приводится набор инструкций для контекста устройства, используемый для рисования черных прямоугольников с белыми контурами:

```
CPaintDC dc(this);
dc.SelectStockObject(WHITE_PEN);
dc.SelectStockObject(BLACK_BRUSH);
// Здесь рисуются черные
// формы, окруженные
// белыми контурами
```

Для того чтобы увидеть, как это работает, откройте программу SquaresAndCircles и внесите в нее изменения, выделенные в листинге 7.5. Перед рисованием каждого прямоугольника код устанавливает кисть контекста устройства в **HOLLOW_BRUSH**; перед рисованием каждого эллипса код устанавливает кисть в **DKGRAY_BRUSH**. Все формы нарисованы при помощи пера из набора **WHITE_PEN**. Результаты отображены на рис. 7.12.

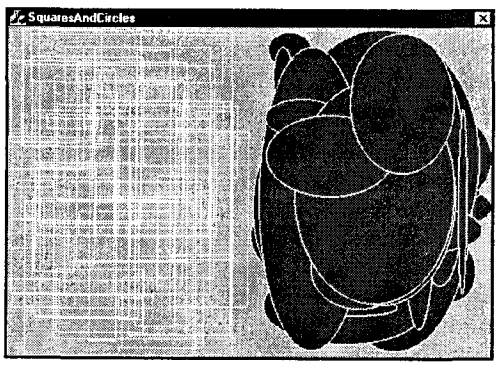


РИСУНОК 7.12 *Работающая программа SquaresAndCircles, использующая наборы перьев и кистей.*

Листинг 7.5 Добавление наборов перьев и кистей к методу CSquaresAndCirclesDlg::OnPaint().

```
void CSquaresAndCirclesDlg::OnPaint()
{
      // Рисование фигур при помощи функций Rectangle() и Ellipse()
      // 1. Получение контекста устройства клиентской области
      CPaintDC dc(this);
```

```
        // 2. Измерение клиентской области
        CRect rect;
        GetClientRect(&rect);

        // 3. Рисование 50 прямоугольников и эллипсов
        int middle = rect.right / 2;
        dc.SelectStockObject(WHITE_PEN);
        for(int shapes = 0; shapes < 50; shapes++)
        {
            dc.SelectStockObject(HOLLOW_BRUSH);
            dc.Rectangle(rand() % middle, rand() % rect.bottom,
                         rand() % middle, rand() %  rect.bottom);

            dc.SelectStockObject(DKGRAY_BRUSH);
            dc.Ellipse(middle + rand() % middle, rand() %
    rect.bottom,
                       middle + rand() % middle, rand() %
    rect.bottom);
        }
}
```

Непрерывное рисование

Все программы, которые рассматривались в этой главе, выполняют рисование при помощи метода **OnPaint()**. Каждая программа также рисует фиксированное число линий или фигур. Для большинства программ эти возможности являются достаточными. Однако в хранителях экрана и других программах, включающих анимацию, потребуется выполнять рисование непрерывно.

Имеются несколько способов реализации непрерывного рисования. Один из самых простых заключается в помещении оператора

```
Invalidate(FALSE);
```

в последнюю строку метода **OnPaint()**. Например, добавьте этот оператор в конец программы SquaresAndCircles, чтобы увидеть, как он работает.

При добавлении свойства изменения размера в приложении LiteTwo использовался метод **CWnd::Invalidate()**. Если найти в документации описание функции **Invalidate()**, несложно заметить, что она принимает единственный параметр **BOOL**, именуемый **bErase**. (В Windows API **BOOL** — это определение типа (**typedef**) для **int**, используемое ввиду того, что встроенный тип данных **bool** — это тип данных C++, предшествующий Windows.) Параметр **bErase** определяет, должен ли фон автоматически уничтожаться перед рисованием нового изображения. MFC присваивает параметру значение по умолчанию **TRUE**, поэтому параметр при вызове можно не указывать. Однако когда выполняется непрерывное рисование, каждый раз уничтожать фон не требуется, поэтому функция **Invalidate()** вызывается с параметром **FALSE**.

Если вы являетесь опытным программистом на языках C/C++, то использование функции **Invalidate()** в конце метода **OnPaint()**, вероятно, будет вас немного нервировать. В конце концов, данная технология кажется опасно близкой к рекурсивным вызовом метода **OnPaint()**. Возможно, вы думаете, что лучше бы использовать бесконечный цикл. Если у вас появились подобные мысли, то примите поздравления! Потребуется лишь точно понять различие между традиционным приложением, основанным на обращениях к функциям, и системой передачи сообщений, управляемой событиями.

При вызове функции **Invalidate()** вы не вызываете (даже косвенно) метод **OnPaint()**. Вместо этого вы указываете Windows, что нужно перерисовывать окно. Windows сама решит, когда нужно выполнить перерисовку.

Если вы попытаетесь изменить способ, в соответствие с которым работает Windows, добавив собственный бесконечный цикл внутри метода **OnPaint()**, то компьютер заблокируется, поскольку Windows никогда не получит возможности обработать другие сообщения. Windows выставляет очень низкий приоритет для обработки сообщений **WM_PAINT**, генерируемых обращениями к **Invalidate()**, что означает возможность обработки Windows любых других отложенных сообщений перед возвратом к перерисовке окна.

Встречайте: таймеры Windows

Непрерывное рисование обычно выполняется за счет использования таймера Windows. В случае применения метода **Invalidate()** нет никакого способа управления тем, насколь часто Windows должна перерисовывать изображение. Однако если используется таймер, можно организовать перерисовку изображения каждую секунду или каждую минуту — по вашему выбору. При использовании таймера можно даже рисовать, не применяя метод **OnPaint()**, тем самым высвобождая его для других нужд.

Для использования таймера в Windows необходимо знать, как:

- Создать таймер.
- Уничтожить таймер.
- Ответить на сообщение таймера.

Создание таймера

Для создания таймера используется метод **SetTimer()** из **CWnd**, принимающий три параметра:

- *Идентификатор таймера* — целочисленный параметр, который отличает один экземпляр таймера от другого при ответе на их сообщения. Можно использовать любое положительное число.
- *Интервал таймера* — интервал времени ожидания между сообщениями таймера, выраженный в миллисекундах. Если для второго аргумента используется значение 1000, ваш таймер "тикает" примерно каждую секунду. Таймеры Windows имеют максимальную дискретность (т.е. минимальный интервал) 55 миллисекунд, а это означает, что в лучшем случае можно получать приблизительно 18 обращений к таймеру в секунду.
- *Функция обратного вызова таймера* — адрес специальной *функции обратного вызова*, которой посылаются сообщения таймера. Если вы передаете **NULL**, как будет сделано ниже, Windows будет посылать сообщение **WM_TIMER**. Это простой выход из положения, поскольку сообщение **WM_TIMER** можно посылать обычной функции Windows, обрабатывающей сообщения.

Поскольку **SetTimer()** начинает посылать сообщения немедленно, необходимо использовать **SetTimer()** только после создания главного окна. В приложениях, основанных на диалоговых окнах, функция **OnInitDialog()** — это превосходное место для создания таймера. Если все произошло успешно, метод **SetTimer()** возвращает тот же самый идентификатор таймера, который передавался в качестве параметра. Таймеры относятся к ограниченным ресурсам, и поэтому может случить-

ся так, что доступных таймеров больше не окажется. Если **SetTimer()** не может создать таймер, возвращается 0.

Уничтожение таймера

После завершения создания таймера необходимо вызвать метод **KillTimer()** из **CWnd**, передавая идентификатор таймера, используемый при его создании. Поскольку таймеры — ограниченный глобальный ресурс, вызов **KillTimer()** по завершении использования таймера — просто общие правила хорошего тона.

Вызывать **KillTimer()** следует прежде, чем окно будет разрушено. Для приложений, основанных на использовании диалоговых окон, AppWizard не генерирует подходящей функции для разрешения вызова **KillTimer()**. Однако эта оплошность легко исправляется при помощи ClassWizard, используемого для добавления функции обработчика сообщений **WM_DESTROY** и размещения в ней вызова **KillTimer()**.

Ответ на сообщения таймера

Windows использует системные часы для "подсчета" значений интервалов таймера, которые передаются функции **SetTimer()**. Когда интервал времени заканчивается, Windows помещает сообщение **WM_TIMER** в очередь событий программы и сбрасывает значение таймера. Если интервал времени таймера заканчивается прежде, чем программа обрабатывает последнее сообщение, другое сообщение **WM_TIMER** не генерируется. (На самом деле все происходящее немного сложнее, но конечный результат — тот же.)

Для обработки сообщений таймера просто используйте ClassWizard для написания обработчика сообщений **WM_TIMER**, обычно называемого **OnTimer()**. Если планируется выполнять окрашивание изображения в функции **OnTimer()**, можно использовать практически тот же код, который применялся для **OnPaint()**. Поскольку рисование выполняется без использования метода **OnPaint()**, в качестве контекста устройства вместо **CPaintDC** необходимо будет задействовать объект **CClientDC**.

PaintItGray

Прежде чем перейти к следующей главе, давайте создадим приложение, использующее таймер для непрерывного обновления экрана интересным шаблоном. Используемый алгоритм основан на Pascal-программе TVCure Дуга Купера (Doug Cooper). Здесь используются таймеры, новая форма контекста устройства и даже новый режим рисования. В знак уважения к старой песне Роллинг Стоунз программа называется PaintItGray. (Авторы этой книги — люди среднего возраста.)

Для начала формирования приложения PaintItGray создайте приложение, основанное на диалоговых окнах, использующее те же самые опции, которые применялись в предыдущих программах в этой главе. После удаления ненужных элементов управления из главного окна следуйте приведенным ниже инструкциям для завершения приложения:

1. Используйте ClassWizard для добавления к классу **CPaintItGrayDlg** новой функции обработчика сообщений **WM_TIMER**. Добавьте код, выделенный в листинге 7.6, в функцию **OnTimer()**, созданную ClassWizard.

Листинг 7.6 Метод CPaintItGrayDlg::OnTimer().

```
void CPaintItGrayDlg::OnTimer(UINT nIDEvent)
{
```

```
        // ЧТО СДЕЛАТЬ: Добавьте здесь свой код обработчика сообщений
        // и/или вызовите обработчик по умолчанию
        CClientDC dc(this);

        CRect rect;
        GetClientRect(&rect);

        dc.SetROP2(R2_XORPEN);
        dc.SelectStockObject(WHITE_PEN);

        for(int row = rect.bottom - m_boxes; row > m_boxes; row--)
        {
              dc.MoveTo(m_boxes, row);
              dc.LineTo(rect.right - m_boxes, rect.bottom - row);
        }
        for(int col = m_boxes; col < rect.right - m_boxes; col++)
        {
              dc.MoveTo(col, m_boxes);
              dc.LineTo(rect.right - col, rect.bottom - m_boxes);
        }

        m_boxes++;
        m_boxes %= 5;
}
```

2. Откройте метод **OnInitDialog()** и создайте новый таймер. Присвойте таймеру ID, равный 1, и интервал, равный 120. Передайте **NULL** в качестве функции обратного вызова таймера. В листинге 7.7 показан модифицированный метод **OnInitDialog()** с выделенной новой строкой.

Листинг 7.7 Метод CPaintItGrayDlg::OnInitDialog().

```
BOOL CPaintItGrayDlg::OnInitDialog()
{
   CDialog::OnInitDialog();

   // Установить пиктограмму диалогового окна.
   // Каркас делает это автоматически, если главное окно приложения
   // не является диалоговым
   SetIcon(m_hIcon, TRUE);      // Установить большую пиктограмму
   SetIcon(m_hIcon, FALSE);     // Установить малую пиктограмму

   // ЧТО СДЕЛАТЬ: Здесь добавить дополнительный код
   // инициализации
   SetTimer(1, 120, NULL);
   return TRUE;      // вернуть TRUE, если не установлен фокус
                     // на какой-то элемент управления
}
```

3. Используйте ClassWizard снова, добавьте другую функцию обработчика сообщений Windows, на сей раз для сообщения **WM_DESTROY**. В методе **OnDestroy()** добавьте строку, необходимую для уничтожения таймера, как показано в листинге 7.8.

Листинг 7.8 Метод CPaintItGrayDlg::OnDestroy().

```
void CPaintItGrayDlg::OnDestroy()
{
      CDialog::OnDestroy();
```

```
        // ЧТО СДЕЛАТЬ: Добавьте здесь свой код обработчика сообщения
        KillTimer(1);
}
```

4. В окне ClassView щелкните правой кнопкой мыши на **CPaintItGrayDlg** и выберите Add Member Variable. В открывшемся диалоговом окне создайте переменную **m_boxes**. Тип переменной должен быть **int**, а вид доступа к ней — Private.
5. В окне ClassView дважды щелкните на конструкторе **CPaintItGrayDlg**. Добавьте строку, инициализирующую новый элемент данных, которая выделена в листинге 7.9.

Листинг 7.9 Конструктор CPaintItGrayDlg.

```
CPaintItGrayDlg::CPaintItGrayDlg(CWND* pParent /*=NULL*/)
    : CDialog(CPaintItGrayDlg::IDD, pParent)
{
      //{{AFX_DATA_INIT (CPaintItGrayDlg)
            // ЗАМЕЧАНИЕ: Здесь ClassWizard добавляет
            // инициализацию элемента данных
      //}}AFX_DATA_INIT
      // Обратите внимание, что в Win32 LoadIcon не требует
      // последующего вызова DestroyIcon
      M_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
      m_boxes = 0;
}
```

6. Откомпилируйте и запустите на выполнение приложение; постарайтесь не попасть под воздействие гипноза. Выполняющаяся программа показана на рис. 7.13.

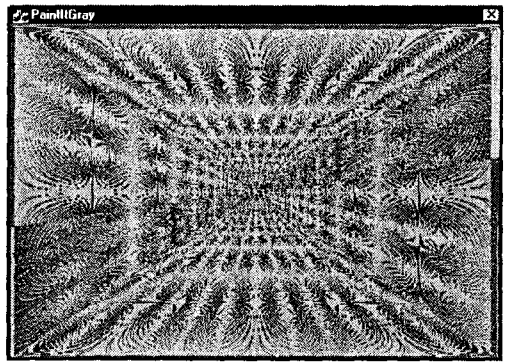


РИСУНОК 7.13 *Выполняющееся приложение PaintItGray.*

Вперед и вверх

В этой главе был пройден длинный путь, однако впереди еще более длинная дорога. В конце концов, мир не состоит из оттенков серого цвета.

В главе 8 некоторое время будет уделено на более глубокое исследование механизма рисования Windows, GDI и классов контекста устройств CDC. Вы узнаете, как создавать пользовательские кисти и перья, и даже рисовать в цвете. Как и было обещано, в главе 8 приобретенные навыки рисования будут использоваться для создания нескольких хранителей экрана, которые можно подключить через Control Panel (Панель управления) в Windows.

В путь!

Глава 8

Графика и текст

Для компьютерных программ необходим как ввод, так и вывод. Ввод позволяет "разговаривать" с программой; вывод же дает программе возможность отвечать пользователям.

Во времена безраздельного господства DOS, программы должны были запрашивать ввод. Выполнялся вызов специальной функции, например, **getch()** для получения ввода с клавиатуры. В среде Windows ситуация совсем другая. Программе больше не надо *запрашивать* ввод — вместо этого пользовательский ввод *поступает* в программу в зависимости от конкретного события, к которому он привязан. Пользователи щелкают на определенных кнопках мыши, нажимают клавиши или перемещают указатель мыши. Каждое такое действие порождает событие, которое Windows автоматически пересылает в программу.

Как и следовало ожидать, в среде Windows вывод также осуществляется иначе. Во времена DOS вывод выполнялся в двух видах: в виде текста и графики. В текстовом режиме можно было использовать такие функции, как **printf()** или **putch()** для вывода символов на экран. В задачу аппаратной части компьютера входило фактическое создание символов и вывод их на экран или принтер. В графическом режиме можно было вычерчивать на экране линии или различные фигуры, а также отображать тексты с использованием разнообразных шрифтов. Для работы в графическом режиме необходимо было использовать графическую библиотеку, содержащую функции, подобные **putpixel()** и **drawline()**. Однако программисты, работающие в среде DOS, часто вынуждены были отказываться от использования графических библиотек общего назначения, поскольку это приводило к снижению производительности, и писать программы, которые непосредственно управляли работой видеоадаптеров и мониторов.

В среде Windows нужда в текстовом режиме отпадает, поскольку весь вывод графический. Однако чтобы выполнить графический вывод, приложение должно воспользоваться библиотекой интерфейса графических устройств (GDI) системы Windows, поскольку другого способа доступа к экрану или принтеру нет.

За кулисами Windows и GDI

GDI — суть графическая библиотека, *не зависящая от устройств*. Под Windows любой вывод — графический, поскольку используется одна и та же библиотека (GDI) независимо от того, направляется ли вывод на экран, принтер, графопостроитель или на какое-то другое экзотическое устройство вывода. Каждая буква, каждая строка, каждый знак, отображаемые программой, пропускаются через утилиты библиотеки GDI, обеспечивающие корректное отображение вывода. Не имеет значения, используется при этом дисплей VGA с 640×480 элементов изображения (пикселей) или суперсовременное фотонаборное устройство с высокой разрешающей способностью.

Чтобы такая гибкость оказалась возможной, GDI использует три механизма:

- *Контекст устройства* — Контекст устройства (device context, или DC) — это *логический холст*, который используется для вывода в системе Windows. Вместо того чтобы выводить данные непосредственно на экран или принтер, вы неизменно пользуетесь контекстом устройства. Более подробно контексты устройств рассматриваются в следующем разделе.
- *Драйвер устройства* — DC (логическое выходное устройство) соединен с физическим устройством вывода через драйвер устройства Windows. Драйвер устройства транслирует команды GDI в физические команды, которые устройство выполняет с целью отображения выходных данных. Этот процесс не требует при написании программы знания конечного выходного устройства. С другой стороны, вы иногда можете потребовать от устройства вывода выполнить то, чего оно сделать не может — например, монитор легко

может воспроизвести на экране желтый круг, в то время как черно-белый лазерный принтер этого сделать не может. Соответствующий драйвер устройства Windows изящно выполняет подобного рода запросы, делая то, что в его возможностях. Например, если вы потребуете от черно-белого лазерного принтера распечатать цветной вывод, драйвер этого устройства может использовать для таких целей оттенки серого цвета или какой-либо другой способ нанесения рисунка.

- *Режим отображения* — Физические устройства вывода выпускаются с различными разрешающими способностями, от стандартной электронно-лучевой трубки с разрешением 72 dpi (dots per inch — точек на дюйм) до фотонаборного устройства с разрешением 4000 dpi. Следовательно, независимая от устройств графическая библиотека должна выполнять преобразование координат. Воспроизведенный на экране квадрат из 100 пикселей занимает немного больше места, чем один квадратный дюйм, но тот же квадрат, отображенный фотонаборным устройством, выглядит бесконечно малым. GDI решает эту проблему посредством реализации различных *режимов логического отображения*. Выполняя графический вывод, программист может указать масштаб (пикселы/дюймы/миллиметры), а также начало координат и направление масштабирования. Использование режимов масштабирования рассматривается в главах 9 и 10.

В дополнение к этим трем свойствам GDI включает в себя все "навороты", какие только можно ожидать от полномасштабной графической библиотеки — от примитивных графических функций типа **SetPixel()** до сложных функций управления растром и цветом.

Что такое контекст устройства и для чего он нужен?

Если мы вернемся к рассмотрению функции **OnPaint()**, употребляемой в программах, написанных в главе 7, мы увидим, что первым делом она получает контекст устройства. В простейшем случае DC представляет собой "холст", используемый для вывода. Тем не менее, DC — это не только просто холст, пассивно воспринимающий все ваши художества, но и активный участник создания изображения — своего рода художник-соавтор.

Когда дело доходит до создания изображения, DC играет три различных роли:

- Он обеспечивает логическую связь с физическим устройством (через драйвер устройства)
- Он предоставляет пользователю набор изобразительных инструментов и атрибутов.
- Он играет роль "регулировщика уличного движения", назначение которого состоит в том, чтобы все ваши попытки нарисовать изображения остались "между строк".

Прежде чем переходить к изучению самих классов DC, рассмотрим вкратце каждую из этих ролей.

Логическая связь

Если вы какое-то время программировали на языках C или C++, вам, несомненно, пришлось читать и записывать дисковые файлы. Чтобы открыть файл в программе на языке C, сначала требовалось создать указатель файла:

```
FILE* fp;
```

Затем этот указатель привязывался к конкретному файлу или устройству посредством функции **fopen()**, например:

```
fp = fopen("myfile.txt", "w");
```

Для записи в файл символа, хранящегося в переменной **ch**, применяется функция **fputc()**:

```
fputc(ch, fp);
```

Это код достаточно элементарен и кажется исключительно простым прежде всего в силу того, что он хорошо известен. Тем не менее, в действительности он *весьма* показателен. Обратите внимание на следующие моменты — команду **FILE*** можно использовать, не зная многих деталей:

- О различных полях, на которые она указывает. Вместо этого вы просто используете ее в качестве *дескриптора (handle)*, обеспечивающего доступ к конкретному устройству вывода.
- О дорожках и секторах, физической организации, которые применяет диск для хранения информации.
- Какой тип устройства находится на другом конце. Например, для вывод на принтер не требуется никакой информации о дорожках и секторах. Вместо этого символы просто печатаются.

В Windows роль контекста устройства аналогична роли **FILE***. Тем не менее, вместо того чтобы предоставить дескриптор носителя для хранения информации, DC предоставляет дескриптор конкретного выходного устройства отображения.

Набор изобразительных средств

Если вы используете функцию **fopen()**, чтобы привязать **FILE*** к конкретному физическому файлу, поля структуры **FILE** инициализируются информацией о конкретном файле. Чтобы получить доступ к этой информации, нужно воспользоваться функциями из стандартной библиотеки ввода/вывода — такими как **ftell()**, которая возвращает адрес дескриптора текущего файла.

Аналогично, DC определяет набор изобразительных инструментальных средств, включая текущее перо для вычерчивания линий, текущую кисть для заполнения областей изображения и текущий шрифт для отображения текста. Вы можете использовать любое количество различных перьев, кистей и шрифтов, однако в каждый конкретный момент DC может определить только одно перо, кисть или шрифт.

Дополнительно к этим инструментальным изобразительным средствам каждый DC содержит набор, содержащий более 20 атрибутов, определяющих режим функционирования этих средств. Среди упомянутых атрибутов содержатся цвет фона рисунка, фоновый режим (непрозрачный и прозрачный), выравнивание текста, цвет, используемый для вывода текста и текущий режим отображения.

Делая эти атрибуты частью DC, GDI упрощает использование инструментальных средств. Например, функция **TextOut()** интерфейса GDI, которая отображает текст, воспринимает только три аргумента: отображаемый текст и координаты x, y, по которым текст должен быть воспроизведен. Все другие необходимые атрибуты — цвет текста, цвет фона, преобразование координат x, y и шрифт — являются частью DC. Без атрибутов GDI такие простые функции, как **TextOut()**, потребуют гораздо более пространного списка параметров.

Отсекающая связь

И последняя важная роль контекста устройства связана с его функцией регулировщика движения, которая заключается в посредничестве в конфликтах между программами, соперничающих за использование одного и того же участка экрана. Поскольку Windows обеспечивает одновременное выполнение нескольких программ, ни одной из них нельзя предоставить неограниченный доступ ко всему дисплею. Представьте себе, какая возникнет путаница, если вы решите воспроизвести изображение в том месте экрана, в которое другое приложение уже поместило свой вывод.

Чтобы выполнять функцию посредничества между программами, GDI использует стратегию, именуемую *отсечением*. В момент получения DC, он содержит *область отсечения* — прямоугольник, в котором вам предоставляется возможность рисовать все, что вам угодно. Если ваше окно активно, то область отсечения обычно заключена в границах окна — в так называемой *клиентской области*. Однако часть окна клиентской области может быть заслонена другим окном, которое может частично перекрывать первое окно. Когда такое случается, область отсечения и клиентская область не идентичны. В подобного рода случаях область отсечения содержит в себе только видимую часть клиентской области.

Итак, что же происходит, когда вы рисуете за допустимыми пределами? А ничего не происходит, и в этом-то и заключается одно из достоинств принципа отсечения. Вместо того чтобы отчитать вас за невнимательность, Windows просто игнорирует вашу просьбу. Но все это происходит за кулисами, так беспокоиться не о чем. Вы просто рисуете, как если бы все окно было видимым, и возлагаете все остальные заботы на Windows.

Понятие семейства CDC

Теперь, когда вы знаете, что такое контекст устройства, вопрос заключается в том, как его получить. В MFC контекст устройства (DC) есть пример одного из четырех потомков класса **CDC**: **CPaintDC**, **CClientDC**, **CWindowDC** и **CMetaFileDC** (см. рис 8.1). И хотя существуют исключения, большинство рисунков создаются с использованием класса **CPaintDC** или класса **CClientDC**. Каждый из них позволяет рисовать в любом месте в пределах окна.

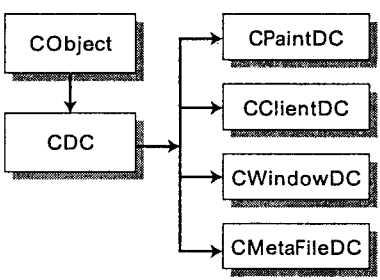


РИСУНОК 8.1. *Классы контекстов устройств Windows.*

Чтобы построить объект **CPaintDC** или объект **CClientDC**, следует вызвать конструктор класса, передавая ему в качестве аргумента указатель на окна, в котором требуется рисовать. Программный код аналогичен кодам, реализованным на протяжении главы 7:

```
CPaintDC dc(this);
```

Вы можете воспользоваться объектом **dc** типа **CPaintDC**, чтобы рисовать в любом месте клиентской области главного окна. Однако вы не можете использовать его для заполнения рисунками участков, лежащих за пределами клиентской

области, таких как полоса заголовка или обрамление окна. Если используются объекты **CPaintDC** или **CClientDC**, позиция 0,0 представляет собой верхний левый угол клиентской области, но не фактическое положение на экране. Если вы хотите рисовать на всем окне, включая полосу заголовка и обрамление, необходимо создать объект **CWindowDC**, а не объект **CPaintDC**. Для объекта **CWindowDC** позиция 0,0 относится к верхнему левому углу окна, но не к верхнему левому углу клиентской области.

Сообщение WM_PAINT и класс CPaintDC

Оба объекта **CPaintDC** и **CClientDC** строят контекст устройства, область отсечения которого включает клиентскую область окна. И хотя это может показаться странным, будьте уверены, чтобы иметь два класса, существуют веские причины: вы используете **CPaintDC**, только когда отвечаете на сообщение **WM_PAINT**. Обычно это делается в рамках метода, именуемого **OnPaint()**.

Windows генерирует сообщение **WM_PAINT** всякий раз, когда требуется воспроизвести часть клиентской области окна. Необходимость в этом может возникнуть после того, как окно было минимизировано, или после того, как другое окно, которое заслоняло частично (или полностью) клиентскую область первого окна, будет закрыто либо перемещено в другое место.

Вместо того чтобы воспроизводить рисунок во всей клиентской области, Windows следит за той его порцией, которую нужно нарисовать заново; такая порция называется *недействительной областью* (*invalid region*). Дабы известить Windows о том, что получено сообщение **WM_PAINT** и выполняется соответствующий ответ, потребуется вызвать методы **CWnd BeginPaint()** и **EndPaint()** из набора доступных изобразительных функций. Если этого не сделать, Windows посчитает, что в окне все еще имеется недействительная область и будет неопределенно долго посылать сообщения **WM_PAINT**.

Тем не менее, внимательно изучив функции **OnPaint()** в главе 7, вы не найдете в них вызовов **BeginPaint()** или **EndPaint()**. Конструктор класса **CPaintDC** автоматически вызывает функцию **BeginPaint()**, а его деструктор — функцию **EndPaint()**. Однако **CClientDC** не выполняет эти автоматические вызовы, так что вы никогда не сможете воспользоваться объектом **CClientDC** в функции, предназначенной для манипулирования сообщениями **WM_PAINT**. В противоположность этому, вы должны использовать класс **CPaintDC** только для функции **OnPaint()**.

Комплект изобразительных инструментальных средств интерфейса GDI

Как было показано в главе 7, нет необходимости использовать по умолчанию черное перо или белую кисть каждый раз, когда создается DC. Вместо этого можно прибегнуть к функции **GetStockObject()**, чтобы изменить, например, **WHITE_PEN** или **BLACK_BRUSH**. Последние представляют собой встроенные или используемые по умолчанию графические объекты для рисования в том смысле, что они уже созданы системой Windows для вашего использования. К сожалению, палитра встроенных инструментальных средств относительно бедна. Практически для каждой

серьезной работы потребуется создавать собственные кисти, перья, шрифты и другие инструментальные средства.

В MFC средства, используемые для рисования в DC, являются элементами класса **CGdiObject**. Как и в случае класса **CDC**, объекты **CGdiObject** не создаются непосредственно. Вместо этого применяется один из шести подклассов **CGdiObject**, реализованных в MFC: **CBitmap**, **CBrush**, **CFont**, **CPalette**, **CPen** или **CRgn** (см. рис. 8.2). В данной главе мы поближе познакомимся с **CBrush**, **CFont** и **CPen**.

Начнем с того, что рассмотрим такие атрибуты, как перья и цвета.

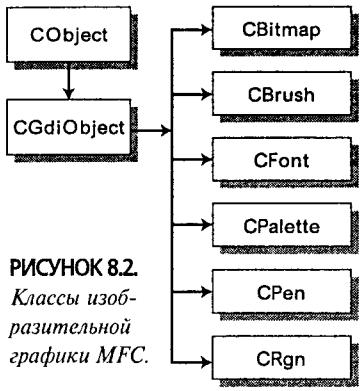


РИСУНОК 8.2. *Классы изобразительной графики MFC.*

Класс CPen, цвет и другие атрибуты CDC

В Windows объекты, ответственные за вычерчивание линий и кривых, — суть перья GDI, представленные в Win32 API типом данных **HPEN** (дескриптор пера). В MFC совсем не обязательно работать с указателями пера (хотя можно, если подобное желание возникает) — вместо этого используется класс **CPen**.

Перо интерфейса GDI имеет три характеристики:

- *Стиль (Style)* — Тип линии, которую вычерчивает перо: сплошная, пунктирная, точечная и пр.
- *Ширина (Width)* — Какой должна быть ширина штриха. При построении пера задается его ширина в логических единицах. Сейчас в качестве логической единицы используется пиксел, однако скоро будет показано, как работать с другими, более удобными логическими единицами.
- *Цвет (Color)* — Цвет вычерчиваемой линии.

Не все эти атрибуты доступны в каждом контексте. Например, *стилизованные линии* (линии, вычерчиваемые с использованием тире, точек и прочего) возможны, только если они имеют ширину в один пиксел. Другой тип пера, именуемый *геометрическим пером*, допускает более точную настройку, зато его труднее создавать и использовать.

Построение пера произвольной конфигурации

Перо произвольной конфигурации с применением класса **CPen** можно построить тремя способами.

Во-первых, можно воспользоваться трехаргументным конструктором, который позволит задать аргументы, определяющие стиль пера, ширину и цвет. Ниже представлен пример построения зеленого пера с шириной в 10 единиц (вскоре вы научитесь определять цвет пера и его стиль):

```
CPen greenPen(PS_SOLID, 10, RGB(0, 255, 0));
```

Это простейший метод, однако он страдает существенным недостатком: если выполнение функции завершится неудачей, например, ввиду нехватки ресурсов Windows, конструктор **CPen** возбудит исключение, что приведет к завершению работы программы (если только для обработки исключений не используется **try-**

catch). Во избежание возможного исключения, Microsoft рекомендует выполнять *двухшаговое создание*.

Двухшаговое создание, которое обычно применяется для всех рисованных объектов GDI, а не только перьев, разбивает процесс создания на этап собственно создания объекта и на этап его инициализации. Чтобы выполнить двухшаговое создание для класса **CPen**, сначала создается объект **CPen** при помощи конструктора по умолчанию:

```
CPen greenPen;
```

Затем вызовом метода **CPen::CreatePen()** инициализируются атрибуты пера, например, так:

```
greenPen.CreatePen(PS_SOLID, 10, RGB(0, 255, 0));
```

Конструктор по умолчанию обеспечивает успешное выполнение операции создания, так что вам не придется иметь дело с возможными исключениями. Функция **CreatePen()** просто возвращает **FALSE (0)**, если ее выполнение завершается неудачно.

Создать экземпляр класса **CPen** можно также посредством создания экземпляра структуры **LOGPEN** и заполнения значениями каждого из ее трех полей, после чего **CreatePenIndirect()** вызывается следующим образом:

```
LOGPEN lp;
lp.lopnStyle = PS_SOLID;
lp.lopnWidth = 10;
lp.lopnColor = RGB(0, 255, 0);
CPen greenPen;
greenPen.CreatePenIndirect(&lp);
```

Иметь дело с этой программой намного труднее, чем просто пользоваться функцией **CreatePen()**, однако имеет смысл освоить эту программу, например, в тех случаях, когда требуется создать несколько перьев одного и того же стиля, но с разной шириной. Функцию **CreatePenIndirect()** потребуется вызвать для каждого объекта **CPen**, однако при этом многократно используется одна и та же структура **LOGPEN**:

```
CPen pens[10];
lp.opnWidth = 1;
for (int i = 0; i < 10; i++)
{
      pens[i].CreatePenIndirect(&lp);
      lp.opnWidth++;
}
```

Использование объектов CPen опасно для вашего здоровья

Как программист, работающий на языке C++, вы знаете, что когда создается объект, вызывается его конструктор. После того как объект создан, его можно создать *повторно*. Создание обычно требует решения двух задач: распределения и инициализации. В процессе распределения резервируется память для объекта, инициализация же устанавливает начальные (исходные) значения каждого поля объекта. Каждая из упомянутых задач может быть выполнена *только один раз*.

Выполняя распределение и инициализацию в два приема, в виде двухшагового метода создания в GDI, программисты часто путают инициализацию с присвоением значений.

В процессе присвоения в существующие поля помещаются новые значения, в то время как в процессе инициализации устанавливаются начальные значения. Функция CPen::CreatePen() выполняет инициализацию, но не присвоение значений; таким образом, для каждого создаваемого объекта CPen CreatePen() можно вызвать только один раз.

Стили перьев

Первым аргументом функции **CreatePen()** является стиль пера. На выбор предоставляются семь стилей: **PS_SOLID**, **PS_DASH**, **PS_DOT**, **PS_DASHDOT**, **PS_DASHDOTDOT**, **PS_NULL**, и **PS_INSIDEFRAME**.

Стиль **PS_NULL** создает перо, аналогичное шаблону пера **NULL_PEN**, за исключением того, что перо **PS_NULL** можно создавать с шириной более одного пиксела.

Стили **PS_DASH**, **PS_DOT**, **PS_DASHDOT** и **PS_DASHDOTDOT** создают перья с шириной в один пиксел и представляют собой комбинацию тире и точек. Нет возможности создавать стилизованные перья с большей шириной. Во время черчения стилизованным пером пространство между каждым тире или точкой окрашивается в цвет фона контекста устройства, принятый по умолчанию. Это может быть тот же цвет, что и цвет кисти, используемой для закрашивания фона окна. Если вы хотите, чтобы цвет фона окна проявлялся между сегментами линий, задайте прозрачный режим фона DC:

```
dc.SetBkMode(TRANSPARENT);
```

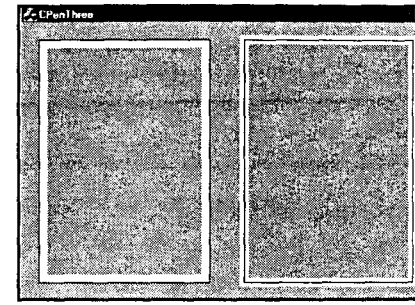
Для установки конкретного цвета в качестве фонового можно воспользоваться функцией **SetBkColor()** контекста устройства.

Каждый из оставшихся двух стилей **PS_SOLID** и **PS_INSIDEFRAME** создает сплошное перо. **PS_SOLID** и **PS_INSIDEFRAME** различаются в двух аспектах. Во-первых, **PS_INSIDEFRAME** — единственный стиль, который может использовать *цвет оттенка (dithered color)* для широких перьев. Цвет оттенка появляется тогда, когда графические аппаратные средства не могут воспроизвести требуемый цвет, так что Windows генерирует образец цвета, максимально приближенный к заданному цвету в конкретных условиях. С выбором цвета оттенка трудностей не возникает, если видеокарта поддерживает 24-разрядные цвета, однако в режимах с низким разрешением подбор цвета оттенка превращается в сложную проблему.

Другое различие между этими двумя стилями пера проявляется, только когда вы рисуете фигуру, использующую ограничивающий прямоугольник. В таких случаях перо **PS_SOLID** с шириной более одного пиксела пытается охватить ограничивающий прямоугольник, при этом одна половина толщины линии попадает за пределы прямоугольника, а другая — внутрь него. Однако когда вы пользуетесь стилем **PS_INSIDEFRAME**, вся толщина линии оказывается внутри прямоугольника.

Упомянутые различия можно наблюдать на рис. 8.3. С левой стороны рисунка изображены два прямоугольника с одними и теми же координатами. Черный прямоугольник начерчен черным пером шириной в 1 пиксел, в то время как белый прямоугольник — белым пером **PS_INSIDEFRAME** шириной в 10 пикселов. (Прямоугольник, нарисованный черным пером, находится поверх прямоугольника, нарисованного белым пером, что позволяет четко видеть первый прямоугольник.) Обратите внимание, что вся толщина белого прямоугольника появляется внутри рамки, начерченной черным пером.

Два прямоугольника в правой части рис. 8.3 отличаются от прямоугольников в левой части только одним моментом: белому перу, использованному для вычерчивания линии шириной в 10 пикселов, был назначен стиль **PS_SOLID**. Обратите внимание на то, как черная линия ложится на белый прямоугольник.



РИСУНКИ 8.3. *Стили пера PS_INSIDEFRAME и PS_SOLID*

Цвета и ширина пера

Второй аргумент метода **CPen()::CreatePen()** — это толщина пера. Если задать толщину пера в логических единицах, то ее фактическое значение зависит от режима отображения. Например, если в качестве режима отображения выбран **MM_LOENGLISH**, толщина в 10 логических единиц составляет примерно $^1/_{10}$" независимо от разрешающей способности дисплея. В случае использования режима отображения по умолчанию (**MM_TEXT**) толщина составляет 10 пикселов. О режимах отображения будет рассказано более подробно в нескольких следующих главах.

Иногда может возникнуть необходимость создать перо, ширина которого составляет ровно 1 пиксел, даже если активен режим отображения, отличный от **MM_TEXT**. В таких случаях значение ширины устанавливается равным 0, а Windows затем создает перо шириной в 1 пиксел, независимо от того, каким является текущий режим.

Третьим аргументом, передаваемым в функцию **CreatePen()** (или трехаргументный конструктор **CPen**) является цвет пера. Windows представляет цвет в виде 32-разрядного числа, получившего название **COLORREF**, в котором фактически используются только 24 разряда. Для получения цвета придется воспользоваться макрокомандой **RGB()**, которая принимает три аргумента: интенсивность красной, интенсивность зеленой и интенсивность синей составляющей, определяющие выбранный цвет. Для каждой из указанных выше трех составляющих выбирается число в диапазоне от 0 до 255; чтобы подавить какую-либо составляющую, соответствующее значение устанавливается равным 0. Число **COLORREF**, представленное как

```
RGB(255, 0, 0);
```

генерирует максимально насыщенный красный цвет, поскольку первый аргумент (*R*ed — красный) принимает максимально возможное значение, а два других аргумента (*G*reen — зеленый и *B*lue — синий) подавлены.

Задавая равные интенсивности каждой составляющей цвета, можно получать оттенки серого цвета от чистого черного (**RGB(0, 0, 0)**) до чистого белого цветов (**RGB(255, 255, 255)**).

Использование объектов CPen

После того как вы построили объект **CPen** или любой другой **CGdiObject** для этой цели, его потребуется активизировать с использованием метода **SelectObject()** класса **CDC**. Поведение функции **SelectObject()** во многом совпадает с поведением функции **SelectStockObject()**, которая применялась в главе 7. Однако вместо передачи ей одного из предопределенных идентификаторов, ей следует передать адрес вновь созданного объекта **CPen**, примерно так:

```
SelectObject(&greenPen);
```

Поскольку DC может манипулировать одновременно только одним пером, функция **SelectObject()** возвращает указатель на ранее выбранное перо. Деструктор пера произвольной конфигурации автоматически возвращает ресурсы операционной системе, когда перо выходит из области видимости. Когда это произойдет, обязательно убедитесь, что DC содержит исходное перо или одно из стандартных перьев. Исходное перо можно выбрать следующим образом:

```
CPen * oldPen, bluePen(PS_SOLID, 5, RGB(0, 0, 255));
oldPen = dc.SelectObject(&bluePen);
// Здесь выполняется рисование
dc.SelectObject(oldPen);
```

Либо же можно выбрать стандартное перо:

```
CPen bluePen(PS_SOLID, 5, RGB(0, 0, 255));
dc.SelectObject(&bluePen);
// Рисование пером голубого цвета
dc.SelectStockObject(BLACK_PEN);
```

Приложение CPenOne: различные стили пера

Иногда наиболее простой способ узнать, как функционирует объект, заключается в написании простейшей тестовой программы, которая демонстрирует его возможности. Рассмотрим функцию **OnPaint()** для CPenOne — приложения, которое можно найти на сопровождающем CD-ROM, и отметим следующие моменты:

- Функция создает шесть черных перьев шириной в один пиксел, по одному для каждого стиля (за исключением **PS_NULLPEN**):

```
// Создать шесть объектов CPen
CPen p1, p2, p3, p4, p5, p6;
p1.CreatePen(PS_SOLID,       1, RGB(0, 0, 0));
p2.CreatePen(PS_DASH,        1, RGB(0, 0, 0));
p3.CreatePen(PS_DOT,         1, RGB(0, 0, 0));
p4.CreatePen(PS_DASHDOT,     1, RGB(0, 0, 0));
p5.CreatePen(PS_DASHDOTDOT,  1, RGB(0, 0, 0));
p6.CreatePen(PS_INSIDEFRAME, 1, RGB(0, 0, 0));
```

- Один за другим эти перья выбираются в DC. Затем приложение CPenOne чертит горизонтальную линию, используя для этого вычисленные ранее значения **x**, **y**, **width** и **height**. После того как все линии будут начерчены, программа увеличивает вертикальную координату с целью подготовки места для следующей линии. Вот как используется первое перо:

```
dc.SelectObject(&p1);
dc.MoveTo(x, y);
dc.LineTo(x + width, y);
y += height;
```

- После того как все линии будут начерчены, CPenOne выбирает **BLACK_PEN** в DC, ожидая истечения срока действия (или, по меньшей мере, завершения функции **OnPaint()**):

```
dc.SelectStockObject(BLACK_PEN);
```

Вы можете оценить, как работает эта программа, обратив взор на рис. 8.4. Посмотрите на то, как стилизованные перья **PS_DOT**, **PS_DASH**, **PS_DASHDOT**

и **PS_DASHDOTDOT** используют белую кисть для раскрашивания пространств между сегментами их линий, даже если в качестве фонового цвета диалогового окна выбран серый. Если вы найдете это нежелательным, положение можно исправить, воспользовавшись функцией **SetBkMode()** или **SetBkColor()** класса **CDC**, как было указано выше. Если вместо этого рисуется диалоговое окно с белым фоном, можно добавить следующую строку в метод **InitInstance()** приложения:

```
SetDialogBkColor(RGB(255,255,255));
```

Приложение CPenTwo: различная ширина пера

Как и приложение CPenOne, CPenTwo представляет собой простую программу, которая демонстрирует методику использования различных объектов **CPen**. Программа CPenTwo создает пять перьев **PS_SOLID**, размер которых изменяется от 1 до 32 единиц:

```
// 4. Создать пять объектов CPen
CPen p1, p2, p3, p4, p5;
p1.CreatePen(PS_SOLID,    1,    RGB(0, 0, 0));
p2.CreatePen(PS_SOLID,    4,    RGB(0, 0, 0));
p3.CreatePen(PS_SOLID,    8,    RGB(0, 0, 0));
p4.CreatePen(PS_SOLID,    16,   RGB(0, 0, 0));
p5.CreatePen(PS_SOLID,    32,   RGB(0, 0, 0));
```

Помните, что используются логические единицы, которые (поскольку режим отображения не изменялся) соответствуют пикселам.

Оставшаяся часть программного кода приложения CPenTwo — такая же, как и у приложения CPenOne. Каждое из пяти перьев несложно заметить на рис. 8.5.

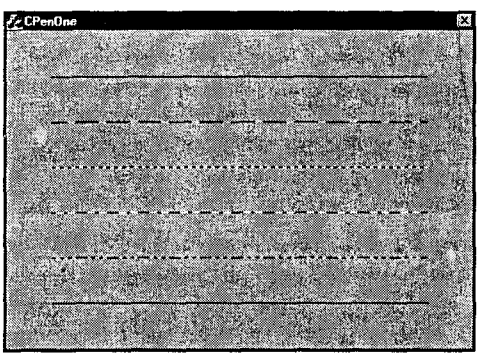


РИСУНОК 8.4. *Различные стили объектов CPen.*

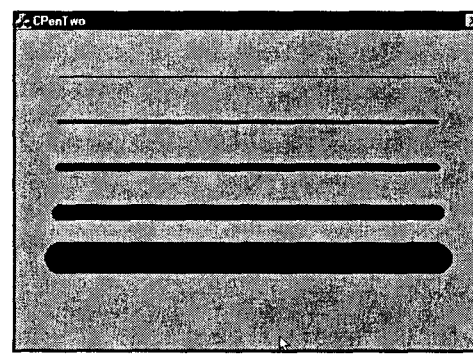


РИСУНОК 8.5. *Различные размеры перьев в приложении CPenTwo.*

Класс CBrush интерфейса GDI

Для раскраски сплошных областей применяются кисти. Работа с кистями во многом напоминает работу с пером — как и в классе **CPen**, вы передаете аргументы конструктору **CBrush** или прибегаете к помощи конструктора по умолчанию **CBrush**, а затем вызываете одну из функций инициализации **CBrush**.

CBrush существуют в трех модификациях: сплошные, штриховые и узорчатые кисти. Сплошные кисти наносят сплошной цвет либо цвет оттенка, что зависит от используемых аппаратных средств. Штриховые кисти используют один из шести

возможных стилей закраски, подобных штриховке, применяемой в архитектурных чертежах. Узорчатые кисти закрашивают области повторяющимися растровыми рисунками в виде квадратов размером в 8 пикселов. Узорчатые кисти рассматриваются в последующих главах, сейчас же приступим к исследованию сплошных и штриховых кистей.

Создание сплошных кистей

За счет использования функций инициализации **CBrush** можно построить два типа сплошных кистей: кисти, цвет которых задается через **COLORREF**, и кисти, цвет которых задается при помощи цветовой системы Windows.

Создание стандартной сплошной кисти во многом напоминает процесс, которому вы следовали, создавая перо в стиле **PS_SOLID**: сначала с помощью конструктора по умолчанию создается объект **CBrush**, затем вызывается функция **CreateSolidBrush()**, в которую передается единственный аргумент, определяющий желаемый цвет. Вот пример построения кисти светло-розового цвета:

```
CBrush pinkBrush;
pinkBrush.CreateSolidBrush(RGB(255, 192, 192));
```

Определив собственные значения **COLORREF**, можно дополнительно использовать один из 31 цвета, перечисленных в табл. 8.1. Панель управления Windows предоставляет пользователям возможность выбирать предпочитаемые цвета для графических элементов во всей системе Windows. Используя в приложениях кисти с системными цветами (отдавая им предпочтение перед жесткозакодированными сплошными кистями), вы делаете свое приложение более удобным для пользователей.

Таблица 8.1. Системные цвета Windows

Идентификатор цвета	*Описание*
COLOR_3DDKSHADOW	Темная тень для трехмерных элементов
COLOR_3DFACE, COLOR_BTNFACE	Поверхности трехмерных элементов
COLOR_3DHILIGHT COLOR_3DHIGHLIGHT COLOR_BTNHILIGHT COLOR_BTNHIGHLIGHT	Выделение трехмерных элементов
COLOR_3DLIGHT	"Свет" для трехмерных элементов
COLOR_3DSHADOW COLOR_BTNSHADOW	Тень для трехмерных элементов
COLOR_ACTIVEBORDER	Граница активного окна
COLOR_ACTIVECAPTION	Заголовок активного окна
COLOR_APPWORKSPACE	Фон главного окна MDI
COLOR_BACKGROUND COLOR_DESKTOP	Рабочий стол
COLOR_BTNTEXT	Текст на кнопках
COLOR_CAPTIONTEXT	Текст заголовка окна
COLOR_GRAYTEXT	Запрещенный (обесцвеченный, или серый) текст

Таблица 8.1. (Продолжение)

Идентификатор цвета	*Описание*
COLOR_HIGHLIGHT	Фон выбранных элементов в управляющих блоках, таких как окно списка
COLOR_HIGHLIGHTTEXT	Текст выбранных элементов в управляющих блоках, таких как окно списка
COLOR_INACTIVEBORDER	Граница неактивного окна
COLOR_INACTIVECAPTION	Заголовок неактивного окна
COLOR_INACTIVECAPTIONTEXT	Текст неактивного заголовка
COLOR_INFOBK	Фон элемента управления подсказкой
COLOR_INFOTEXT	Текст элемента управления подсказкой
COLOR_MENU	Фон меню
COLOR_MENUTEXT	Текст в меню
COLOR_SCROLLBAR	Серая область линейки прокрутки
COLOR_WINDOW	Фон окна
COLOR_WINDOWFRAME	Рамка окна
COLOR_WINDOWTEXT	Текст в окнах
HS_BDIAGONAL	Линии под углом в 45° с нижнего левого угла в верхний правый
HS_CROSS	Взаимно перпендикулярные, горизонтальные и вертикальные линии
HS_DIAGCROSS	Взаимно перпендикулярные линии под углом в 45° относительно горизонтали
HS_FDIAGONAL	Линии под углом в 45° с верхнего левого угла в нижний правый
HS_HORISONTAL	Горизонтальные линии
HS_VERTICAL	Вертикальные линии

Чтобы воспользоваться системными цветами, вызовите функцию **CreateSysColorBrush()**, например, следующим образом:

```
CBrush bkgrndBrush;
bkgrndBrush.CreateSysColorBrush(COLOR_WINDOW);
```

Создание штриховых кистей

Для создания кисти со штриховым стилем следует воспользоваться функцией **CreateHatchBrush()**, которая требует задания двух аргументов: штрихового стиля, который выбирается из табл. 8.2, и цвета штриховых линий (представленных как **RGB COLORREF**). Текущий цвет фона появляется между штрихами. Если вы хотите изменить их, используйте метод **CDC::SetBkColor()**.

Ниже представлен пример, в рамках которого производится построение голубой решетки как фоновой кисти:

```
CBrush blueGrid;
blueGrid.CreateHatchBrush(HS_CROSS, RGB(192, 192, 255));
```

Пример использования приложения CBrushOne: применение штриховых кистей

Прежде чем мы оставим тему кистей и перьев, рассмотрим короткие примеры использования штриховых кистей. Как и в случаях примеров использования перьев, программный код приложения CBrushOne можно найти на сопровождающем CD-ROM. Программа CBrushOne создает шесть кистей штрихового стиля с различными цветами:

```
CBrush b1, b2, b3, b4, b5, b6;
b1.CreateHatchBrush(HS_BDIAGONAL,    RGB(255, 0, 0));
b2.CreateHatchBrush(HS_FDIAGONAL,    RGB(0, 255, 0));
b3.CreateHatchBrush(HS_CROSS,        RGB(0, 0, 255));
b4.CreateHatchBrush(HS_DIAGCROSS,    RGB(255, 255, 0));
b5.CreateHatchBrush(HS_HORIZONTAL,   RGB(255, 0, 255));
b6.CreateHatchBrush(HS_VERTICAL,     RGB(0, 255, 255));
```

Как только программа построит шесть кистей, она разбивает главное окно на две части, причем в каждой части нарисованы три прямоугольника, а для закраски каждого прямоугольника используются различные кисти. Программа раскраски первого прямоугольника имеет вид:

```
// 4. Раскрасить с использованием всех
// шести кистей
dc.SelectObject(&b1);
dc.Rectangle(0, 0, rect.right/3, rect.bottom/2);
```

Программный код для других прямоугольников аналогичен — меняются только кисти и координаты. Результат выполнения рассматриваемого приложения представлен на рис. 8.6.

Режимы рисования CDC

Какого цвета вы ожидаете видеть на дисплее линию, если вы создадите красное перо, а затем проведете саму линию? Как и большинство людей, вы можете ответить: "красного". Но ответ не обязательно будет таким.

Рисование в Windows имеет много общего с рисованием акварельными красками. Когда вы раскрашиваете акварельными красками, цвет вашей кисти есть один из факторов, определяющих окончательный цвет, который появляется на картине. Окончательный же эффект определяет все: влажность бумаги, краски, положенные ранее и относительная влажность кисти.

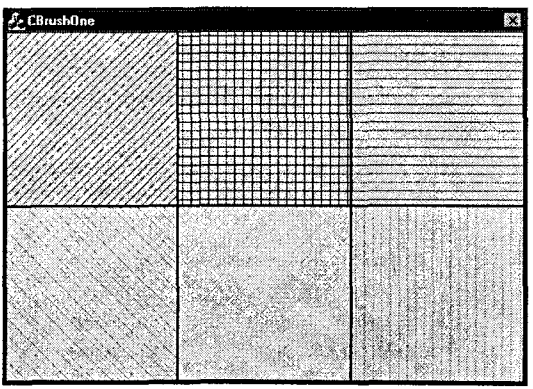


РИСУНОК 8.6. *Результат выполнения приложения CBrushOne.*

При выполнении рисования в Windows *режим рисования* определяет, как цвет пера (и кисти) сочетается с цветами, уже нанесенными на экран, чтобы дать окончательный результат. Режим рисования устанавливается при помощи функции **CDC::SetROP2()**. (**ROP** оз-

начает *R*aster *Op*eration, т.е. растровая операция, и определяет метод отображения цветов на экране электронно-лучевой трубки.) На выбор предлагаются 256 растровых операций, но только 16 из них используются достаточно часто, в силу чего им даны имена (которые начинаются с ***R2_***). Чаще других применяются следующие растровые операции:

- ***R2_COPYPEN*** — Наносит цвет пера прямо на поверхность. В случае выбора красного пера получатся красные пиксели. Этот режим подобен рисованию сухой кистью на сухой бумаге.
- ***R2_BLACK***, ***R2_WHITE*** и ***R2_NOP*** — Перо, выбранное в контексте устройства, игнорируется, и рисование всегда выполняется, соответственно, черным, белым цветом или вообще не производится.
- ***R2_NOT*** и ***R2_NOTCOPYPEN*** — Рисование с помощью "инверсии", соответственно, цвета экрана и цвета пера. Эти режимы полезны, когда требуется, чтобы линия обязательно была видимой, независимо от цвета других линий или фигур, расположенных под ней.
- **R2_XORPEN** — Выполняет операцию "исключающее или", иначе **XOR**, с цветом пера и цветом пикселов экрана. Если вы используете **R2_XORPEN**, проводя одну и ту же линию дважды, эта линия будет стерта.

Создание собственной программы хранителя экрана

В следующей главе вы узнаете, как реагировать на сообщения, поступающие от мыши или клавиатуры, а также как использовать их для создания интерактивных программ рисования. Воспользуемся тем, что мы на этот момент почерпнули из этой главы о цветах, перьях и кистях, для написания программы хранителя экрана.

Если посмотреть раздел "хранители экрана" в документации по Visual C++, можно найти в ней несколько разделов, которые помогут справиться с любыми осложнениями. Наш хранитель экрана будет несколько проще — нужно будет, например, нажать клавишу или щелкнуть кнопкой мыши, чтобы погасить экран, а во всех других отношениях она будет работать, как прочие программы такого типа.

Каркас SuperSaver

Назовем наше приложение SuperSaver. Для его создания потребуется выполнить следующие шаги:

1. Для создания приложения, основанного на диалоговых окнах, получившего имя SuperSaver, воспользуйтесь AppWizard.
2. Удалите из диалогового окна все элементы управления. Откройте диалоговое окно Dialog Properties и установите в поле Border (Обрамление) значение None (см. рис 8.7).

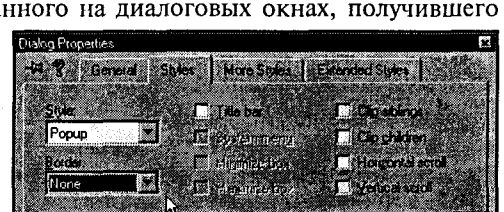


РИСУНОК 8.7. *Установка значения None для параметра Border.*

3. Измените функцию **CSuperSaverApp::InitInstance()** таким образом, чтобы она приняла вид, представленный в листинге 8.1. Вот как работает код. Если хранитель экрана запускается через Control Panel, в него обеспечивается передача аргумента командной строки: "c" для конфигурирования хранителя экрана или "s" для запуска в режиме сохранения экрана. Поскольку вы не намерены создавать каких-либо конфигураций, остается только проверить наличие ключа "s" и выполнять свою программу. Для проверки наличия ключа воспользуйтесь стандартной функцией **strcmpi()** языка C, проверяющей значение командной строки, которую система MFC автоматически сохраняет в **m_lpCmdLine**. (Условные операторы можно поместить в комментарии, пока не станет ясно, что программа работает правильно.) Вызов функции **SetDialogBkColor()** обеспечивает установку черного фона. Поскольку значение, возвращаемое функцией **DoModal()**, не используется, нет необходимости в его сохранении.

Листинг 8.1. Функция CSuperSaverApp::InitInstance().

```
BOOL CSuperSaverApp::InitInstance()
{
      if (!strcmpi(m_lpCmdLine, "/s") ||
            !strcmpi(m_lpCmdLine, "-s") ||
            !strcmpi(m_lpCmdLine, "s" ))
      {
            SetDialogBkColor(RGB(0,0,0));

            CSuperSaverDlg dlg;
            m_pMainWnd = &dlg;
            dlg.DoModal();
      }
      return FALSE;
}
```

4. В рамках метода **OnInitDialog()** диалоговых окон выберите максимальный размер главного диалогового окна, чтобы оно заполняло весь экран и располагалось поверх всех других окон. Листинг 8.2 показывает, как это сделать. Для помещения диалогового окна сверху воспользуйтесь функцией **SetWindowPos()**, передавая ей в качестве аргумента **&wndTopMost**. Для установки соответствующего размера окна можно вызвать функцию **GetSystemMetrics()** интерфейса API. Установите также флаг **SWP_SHOWWINDOW**, что обеспечит немедленное отображение окна на экране. Помимо выбора размеров окна, потребуется инициализировать таймер, который выполнит раскраску отображения, а также инициализировать поле **m_boxes** значением 0.

Листинг 8.2. Метод CSuperSaverDlg::OnInitDialog().

```
BOOL CSuperSaverDlg::OnInitDialog()
{
   CDialog::OnInitDialog();
   // Установить пиктограмму диалогового окна.
   // Каркас делает это автоматически, если главное окно приложения
   // не является диалоговым
   SetIcon(m_hIcon, TRUE);       // Установить большую пиктограмму
   SetIcon(m_hIcon, FALSE);      // Установить малую пиктограмму

   // ЧТО СДЕЛАТЬ: Здесь добавить дополнительный код инициализации
   SetWindowPos(&wndTopMost, 0, 0,
```

```
            ::GetSystemMetrics(SM_CXSCREEN),
            ::GetSystemMetrics(SM_CYSCREEN),
            SWP_SHOWWINDOW);

   SetTimer(1, 250, NULL);
   m_boxes = 0;
   return TRUE;        // вернуть TRUE, если не установлен фокус
                       // на какой-то элемент управления
}
```

5. Добавьте **m_boxes** в класс **CSuperSaverDlg** как элемент данных **private**, **int**.
6. Щелкните правой кнопкой мыши на классе **CSuperSaverDlg** в окне ClassView и выберите элемент Add Windows Message Handler из контекстного меню. Добавьте программный код, закрывающий хранитель экрана при щелчке кнопкой мыши или нажатии клавиши. Начнем с реализации метода обработки нажатия клавиши. В диалоговом окне выберите сообщение **WM_KEYDOWN**, затем щелкните на Add and Edit. В рамках этого метода заменим программный код одним оператором:

```
CDialog::EndDialog(IDOK);
```

Повторим этот процесс для сообщений **WM_LBUTTONDOWN** и **WM_RBUTTONDOWN**. В конечном итоге код должен оказаться таким, как представленный в листинге 8.3.

Листинг 8.3. Методы, обеспечивающие закрывание окна хранителя экрана.

```
void CSuperSaverDlg::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)
{
      CDialog::EndDialog(IDOK);
}

void CSuperSaverDlg::OnLButtonDown(UINT nFlags, CPoint point)
{
      CDialog::EndDialog(IDOK);
}

void CSuperSaverDlg::OnRButtonDown(UINT nFlags, CPoint point)
{
      CDialog::EndDialog(IDOK);
}
```

7. Добавьте обработчик сообщений **WM_TIMER**. Программный код аналогичен коду приложения PaintItGray, написанному в предыдущей главе. Поскольку вы намерены использовать цвет, а не оттенки серого, внесите изменения, показанные на листинге 8.4.

Листинг 8.4. Функция OnTimer() для хранителя экрана SuperSaver.

```
void CSuperSaverDlg::OnTimer(UINT nIDEvent)
{
      CClientDC dc(this);

      CRect rect;
      GetClientRect(&rect);

      dc.SetROP2(R2_XORPEN);

      Cpen randomPen;
      randomPen.CreatePen(PS_SOLID, 1, RGB(rand() % 255,
                                           rand() % 255,
```

```
                                        rand() % 255));
    dc.SekectObject(&randomPen);

    for (int row = rect.bottom - m_boxes; row > m_boxes; row--)
    {
        dc.MoveTo(m_boxes, row);
        dc.LineTo(rect.right - m_boxes, rect.bottom - row);
    }
    for (int col = m_boxes; col < rect.right - m_boxes; col++)
    {
        dc.MoveTo(col, m_boxes);
        dc.LineTo(rect.right - col, rect.bottom - m_boxes);
    }
    m_boxes++;
    m_boxes %= 5;
    dcSelectStockObject(BLACK_PEN);
}
```

8. Добавьте заключительный обработчик сообщений **WM_DESTROY**. В метод **OnDestroy()** с целью разрушения таймера добавьте строку

```
KillTimer(1);
```

9. Убедитесь в том, что Control Panel признает вашу программу в качестве хранителя экрана. Выберите в меню Insert|Resource, а затем String Table в диалоговом окне Insert Resources. Откройте диалоговое окно String Properties, дважды щелкнув на первом вхождении строковой таблицы. Измените идентификатор этой строки на **IDS_DESCRIPTON** и убедитесь в том, что **IDS_DESCRIPTION** принимает значение 1 (см. рис. 8.8).
10. Поместив в комментарий условные операторы в начале метода **CSuperSaverApp::InitInstance()**, откомпилируйте программу и убедитесь в ее корректном функционировании. Когда вы будете удовлетворены ее функционированием, скопируйте выполняемый файл в каталог Windows\System и переименуйте программу так, чтобы она получила расширение .scr. Затем щелкните правой кнопкой мыши на рабочем столе системы, чтобы открыть диалоговое окно Display Properties (Свойства | Экран). Выберите страницу Screen Saver (Заставка) и добавьте новый хранитель экрана. Программа SuperSaver в работе показана на рис. 8.9.

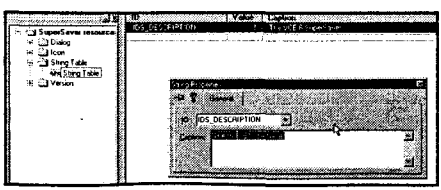


РИСУНОК 8.8. *Добавление ресурса строковой таблицы.*

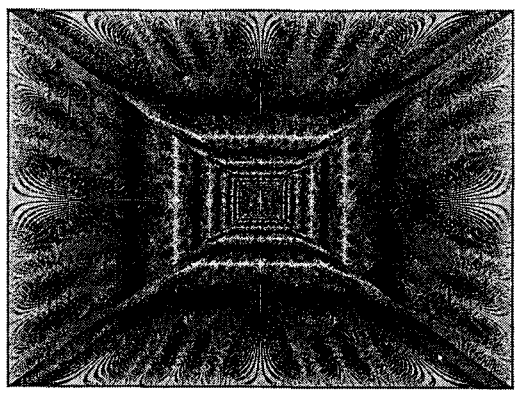


РИСУНОК 8.9. *Выполнение хранителя экрана SuperSaver.*

Что же дальше?

Как вы сами можете убедиться, библиотека Windows GDI обеспечивает все возможности, необходимые для генерации впечатляющих графических эффектов. MFC и Visual C++ упрощают задачу задействования всей мощи интерфейса API за счет реализации большого объема деталей, которые отвлекали программистов предыдущего поколения от решения основных задач. Тем не менее, как и в случаях других устройств, обеспечивающих снижение трудозатрат, возникают также и потенциальные опасности. Пока вы не уделите должного внимания анализу функциональности инструментальных средств, вы никогда полностью не овладеете искусством программирования в среде Windows. Апробирование возможностей — дело полезное, однако следует уделять время и на размышления.

Как было показано в процессе разработки приложения SuperSaver, включение программных кодов, распознающих нажатия клавиш и щелчки кнопками мыши, требует в Visual C++ лишь незначительных усилий. В следующей главе у вас будет шанс приобщиться к этим мощным средствам и сделать рисунки интерактивными, предоставив пользователям возможность вносить требуемые изменения самостоятельно.

Глава 9

Кошмар Пикассо: Построение интерактивной программы рисования

Неподвижная графика хороша, подвижная графика еще лучше, а интерактивная графика — лучше всех. Компьютер обладает особой магией в силу того, что в отличие от кассетного видеомагнитофона, пользователь может управлять событиями. Это отнюдь не воспроизведение, это настоящая игра.

Помните ли вы тот момент, когда впервые увидели компьютерную игру? Я помню. Летом 1969 года, когда мне было 10 лет, мои родители (а они были учителями в средней школе) скопили немного денег и взяли меня и моего семилетнего брата в автомобильную туристическую поездку по Европе. Родители приобрели новый VW Beetle, и мы вместе с братом провели 6 недель на заднем сидении автомобиля. (Мне кажется, что такая поездка показалась моим родителям превосходной идеей, когда они ее задумали, однако она оказалась единственной за довольно таки продолжительное время.)

Поскольку машина была очень маленькой, мы были вынуждены часто останавливаться и посещать различные музеи и аттракционы. Причиной одной из таких остановок был Музей науки и промышленности (Museum of Science and Industry) в Женеве, Швейцария, и вот тут я и увидел первый компьютер — машину, которая могла играть в "крестики и нолики".

Я полагаю, что это была достаточно примитивная машина, но этого чудовища, непринужденно разгрызающего числовые задачки оказалось вполне достаточно, чтобы поразить воображения десятилетнего мальчика. Это было нечто, с чем я мог *играть*. И поскольку эта штука была интерактивной, она была больше, чем игрушка. Я был очарован. К сожалению, она выглядела довольно громоздкой и, я не сомневаюсь, достаточно дорогой. Я убеждал моих родителей, что игрушка займет в машине нисколько не больше места, чем мой брат, но они меня и слушать не хотели. Только почти 20 лет спустя я стал обладателем собственной машины "крестики и нолики".

В главе 8 вы научились проводить линии, строить фигуры и раскрашивать их, т.е. всем необходимым элементам качественной интерактивной программы. Но эти элементы используются в вашей программе только на стадии вывода. А как насчет ввода?

В Windows пользователи могут генерировать ввод двумя способами: за счет использования мыши либо клавиатуры. Большую часть времени вы не имеете дело с "сырыми" входными сообщениями, посылаемыми мышью или клавиатурой — вместо этого вы используете кнопки, элементы управления текстом или меню с тем, чтобы захватить пользовательские щелчки кнопками мыши и нажатия на клавиши. Элементы управления сообщают вашей программе, когда случается что-либо интересное.

В этой главе будет разработана интерактивная программа рисования, которая выполняет обработку "сырого" ввода пользователя. Вы научитесь реагировать на щелчки и перемещения мыши. Вы также узнаете, как сочетать навыки, приобретенные вами при изучении нескольких последних глав, с новыми приемами запроса пользовательского ввода, благодаря чему у вас появится столь необходимый опыт:

Версия 1 программы PaintORama

Прежде чем мы погрузимся в детали, наметим несколько целей, которые необходимо достичь. В этой главе мы будем иметь дело с пятью версиями программы PaintORama. Каждая последующая версия добавляет в предыдущую набор новых возможностей. На сопровождающем книгу CD-ROM вы найдете все упомянутые версии, каждую в собственном каталоге, так что можете воспользоваться любой из них по своему усмотрению.

Ниже описана функциональность каждой версии:

- Версия 1 создает главный экран, затем добавляет к нему холст, на котором располагаются ваши рисунки, и кнопку Clear для их стирания. Эта версия позволяет наносить от руки базовый рисунок, воспользовавшись черным пером шириной в один пиксел, предоставляемым по умолчанию. После изучения этого раздела станет понятно, как обрабатывать *сообщения мыши* в Windows.
- Версия 2 включает в программу PaintORama перья переменной ширины. Помимо этого, вы узнаете как создавать новые *счетчики*.
- Версия 3 начинается с добавления цветных перьев, выполняя экскурс в *общие диалоговые окна*. Далее эта версия выходит за пределы рисунка от руки и вовлекает в процесс линии и формы. Вы узнаете, как воспользоваться классом **CComboBox** для добавления в программу выпадающих списков. Также будет показано, как включить в графические программы визуальную обратную связь, используя для этой цели *метод резиновой нити*.
- Поскольку вы уже умеете пользоваться перьями, версия 4 реализует поддержку кистей. Кроме того, мы объясним, как пользоваться элементом управления **CListBox** из Windows.
- Версия 5 возвращает вас к основам — к обработке сообщений **WM_PAINT** программы PaintORama за счет записи и воспроизведения всех ваших графических операций. Для этого используются классы **CMetaFileDC** (такими как **CPaintDC** и **CClientDC**), каждый из которых — суть подкласса всевидящего класса **CDC**.

Дабы у читателя при этом не возникало нервных стрессов, этот амбициозный проект был разбит на две главы. В текущей главе разрабатываются две первых версии программы PaintORama. Как и в случае хорошего телевизионного сериала, вам придется научиться ждать следующей главы, чтобы увидеть, что из этого получится.

Все версии PaintORama в этой и в следующей главах — это приложения, в основу которых положены диалоговые окна. В главе 11 в проект PaintORama вносятся дальнейшие усовершенствования за счет привлечения архитектуры "документ-представление".

Построение программы PaintORama

Начнем с того, что построим базовый каркас программы PaintORama. Для этого потребуется выполнить следующие шаги:

1. Чтобы начать разработку новой программы, основанной на диалоговых окнах, воспользуйтесь AppWizard. В окне AppWizard — Step 2 позаботьтесь о том, чтобы флажок 3D Controls был включен, а все остальные флажки — сброшены, как показано на рис. 9.1. Присвойте проекту имя PaintORama и щелкните на кнопке Finish. Когда на экране появится диалоговое окно New Project Information, щелкните на кнопке OK.
2. В окне ResourceView выберите Dialog Editor и шаблон **IDD_PAINTORAMA_DIALOG**. Когда появится Dialog Editor, удалите метку "TODO:", а также кнопки OK и Cancel. Оставьте заголовок диалогового окна установленным в значение PaintORama.

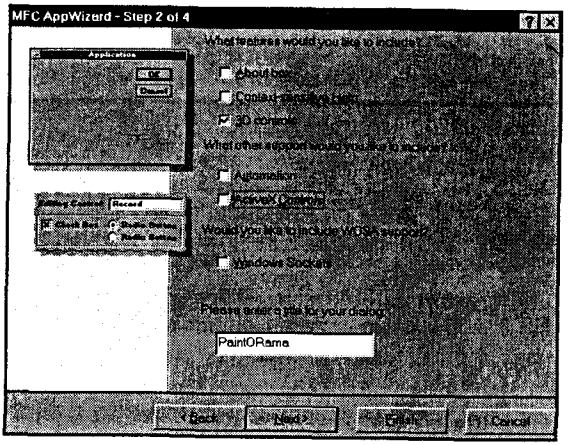


РИСУНОК 9.1.
Окно AppWizard — Step 2. Приложение PaintORama

3. Перетащите нижний правый угол диалогового окна таким образом, чтобы размеры окна составили 460 единиц в ширину и 275 единиц в высоту, руководствуясь значениями координат, отображаемыми в строке состояния в нижней правой части строки состояния. В случае возникновения трудностей с установкой размеров диалогового окна из-за того, что размеры окна Dialog Editor недостаточно велики, выберите в меню View | Full Screen, а затем раскройте окно Dialog Editor. Для возврата в IDE нажмите Esc или щелкните на небольшой пиктограмме в верхнем левом углу. Полноэкранный режим обладает только одним существенным недостатком: строка состояния больше не показывает размеров диалогового окна, в чем можно убедиться из рис. 9.2.

4. Перетащите элемент управления изображением из панели Control в диалоговое окно. Замените его идентификатор на **IDC_CANVAS**, тип на Frame и цвет на Gray (см. рис. 9.3). На странице Extended Styles диалогового окна Picture Properties, активизируйте флажок Client Edge (Края клиентской области) и сбросьте все остальные флажки. На странице Styles выберите только Sunken.

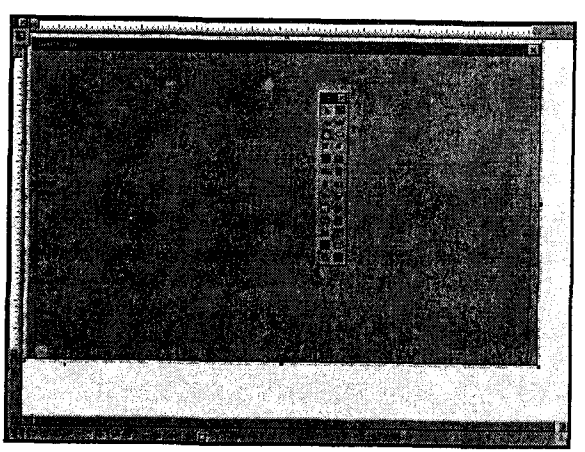


РИСУНОК 9.2. *Использование Dialog Editor в полноэкранном режиме*

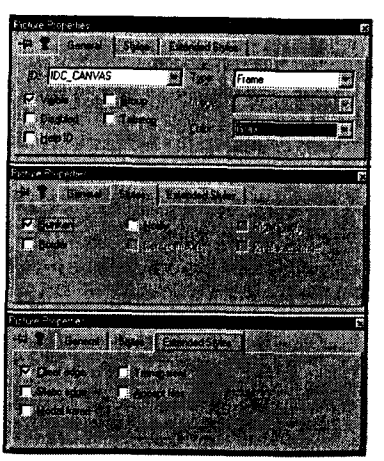


РИСУНОК 9.3. *Свойства элемента управления IDC_CANVAS*

5. Расположите и выберите размеры элемента **IDC_CANVAS** управления рамкой так, чтобы он занимал правую сторону диалогового окна. Размеры этого элемента должны иметь ширину порядка 295 единиц и высоту 253 единиц, верхний левый угол должен иметь координаты 150, 7. Разумеется, вовсе не обязательно абсолютно точно выдерживать эти размеры — просто зарезервируйте примерно 150 единиц с левой стороны диалогового окна для элементов управления приложения PaintORama. Возвращаясь к рис. 9.2, вы найдете там измерительные линейки.
6. Добавьте в верхний левый угол кнопку. Присвойте этой кнопке идентификатор **IDC_CLEARBTN** и установите ее заголовок в "Clear" (Очистить). Эта кнопка используется для стирания рисунков всякий раз, когда требуется очистить холст.

Это все, что касается данной части интерфейса. При желании можно откомпилировать программу просто для того, чтобы убедиться, что вы до сих пор не допустили ошибок.

Добавление элементов данных

В версии 1 приложения PaintORama при помощи мыши будут выполняться только рисунки от руки. Вот как это делается:

- Если щелкнуть мышью в области, отведенной под холст, т.е. в области, ограниченной рамкой **IDC_CANVAS**, программа будет вести себя так, как если бы перо опустилось на холст.
- Рисунок наносится методом перетаскивания — перемещения мыши при нажатой левой кнопке.
- Отпускание кнопки мыши приводит к прекращению рисования.

Для завершения этой операции потребуется помощь трех элементов данных: один из них отслеживает положение границ холста, другой определяет, в каком месте был начерчен последний фрагмент линии, а третий фиксирует тот факт, что пользователь на самом деле намерен начать вычерчивание линии. После краткого напоминания о том, как производится добавление элементов данных все упомянутые элементы будут подробно рассмотрены.

Кодирование программы начинается с добавления трех элементов данных в класс **CPaintORamaDlg** в соответствии с таким способом:

1. Щелкните правой кнопкой мыши на имени класса **CPaintORamaDlg** в окне ClassView и выберите Add Member Variable из контекстного меню (см. рис. 9.4).
2. Когда откроется диалоговое окно Add Member Variable, введите "**CRect**" в качестве типа и "**m_Canvas**" — в качестве имени переменной. Для доступа выберите значение **private**, воспользовавшись группой переключателей. На рис. 9.5 показан окончательный вариант диалогового окна Add Member Variable. Переменная **m_Canvas** содержит размеры области холста, в которой можно рисовать. На эти значения следует ориентироваться, чтобы не выходить за пределы "допустимой области".

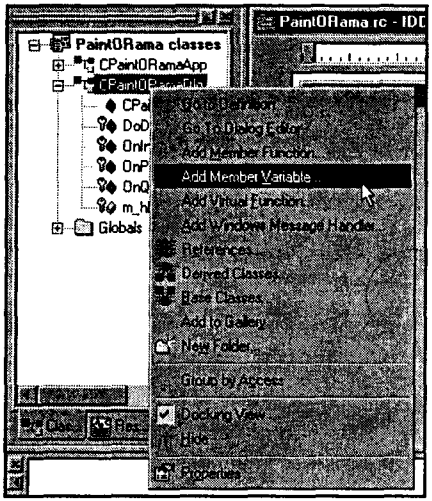


РИСУНОК 9.4. *Добавление элемента данных в класс CPaintORamaDlg*

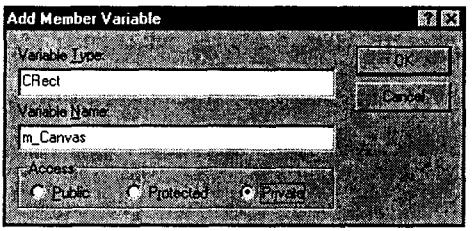


РИСУНОК 9.5. *Использование диалогового окна Add Member Variable*

Многие программисты считают полезным сопровождать переменные комментариями, однако при работе в диалоговом окне Add Member Variable такое сделать не удастся. По этой причине многие предпочитают включать элементы данных в определение класса вручную, тем самым имея возможность форматировать и комментировать каждый элемент по своему усмотрению. Как при использовании диалогового окна, так и при вводе переменных вручную, Visual C++ сообщает об изменениях и отображает переменную в панели ClassView, как только ее ввод с клавиатуры завершен. В случае удаления переменной (эта операция выполняется вручную), последняя исчезает из окна ClassView, как только будет удалено ее объявление.

Воспользуйтесь любым из рассмотренных методов для включения в класс **CPaintORamaDlg** двух оставшихся элементов данных:

- ***m_LineStart*** — **private CPoint**, содержащий координаты начальной позиции сегмента линии, вычерчиваемой в данный момент.
- ***m_IsDrawing*** — **private bool**, который позволяет проводить различие между правильными и неправильными движениями мыши. Когда пользователь нажимает левую кнопку мыши в области холста, **m_IsDrawing** устанавливается в **true**, а положение мыши запоминается в **m_LineStart**. Затем, когда обнаруживается событие движения мыши, можно будет сказать, выполняется рисование или нет, а если да — то где должна начинаться вычерчиваемая линия.

Инициализация элементов данных

Если вы достаточно долго занимаетесь программированием, то наверняка усвоили, что неудачная инициализация переменных — одна из главных причин сбоев в работе программного обеспечения (программных ошибок, если быть более точным). Как программист, работающий на языке C++, вы, по всей вероятности, хорошо усвоили, что цель конструктора состоит в обеспечении уверенности, что

объекты "построены корректно". Вы знаете, что конструктор должен везде, где это возможно, присвоить каждой переменной правильное значение.

Ключевой фразой здесь является, однако, "где это возможно". При программировании в Windows и MFC часто это *не* представляется возможным. В случае приложения, основанного на диалоговых окнах, элементы управления главным окном во время выполнения конструктора еще не созданы. Поэтому большинство классов MFC включают вспомогательную функцию, обеспечивающую возможность проинициализировать все элементы данных. Для диалоговых приложений таковой является виртуальная функция **OnInitDialog()**. В шаблоне MFC-приложения функция **OnInitDialog()** вызывается сразу *после* построения всех дочерних элементов управления приложения. Для целей приложения PaintORama функция **OnInitDialog()** — именно то, что вам нужно.

Инициализировать поле **m_LineStart** не потребуется, поскольку оно будет инициализировано, как только выполнить щелчок кнопкой мыши, начиная рисунок. Однако необходимость инициализировать переменную **m_IsDrawing** все же возникнет по причине того, что когда вы первый раз переместите указатель мыши на холст, обнаружится, что линии чертятся, хотя этого не должно быть. Для инициализации переменной **m_IsDrawing** потребуется добавить такие строки

```
// 1. Предположить, что мы не рисуем
m_IsDrawing = false;
```

в функцию **CPaintORama::OnInitDialog()**, в то место, где находится комментарий

```
// ЧТО СДЕЛАТЬ: Здесь добавить дополнительный код инициализации
```

Правда о переменных типа BOOL, bool, int

Каждому программисту, работающему на языке C, известно, что "все истинно, ничего нет ложного". Но в прекрасной новой среде Standard C++, даже это утверждение неверно.

В C++ и C с давних пор для представления булевских значений (true/false) используется целочисленный тип. Когда принимается решение или проверяется условие выхода из цикла, целое значение 0 означает ложь, а любое другое — истину. Для представления булевских величин Windows API использует typedef BOOL наряду с буквальными константами *TRUE* и *FALSE*.

В ANSI Standard C++ (который все еще находится на стадии окончательного уточнения) вводится новый встроенный булевский тип *bool*, который был представлен в Visual C++, начиная с версии 5. Переменные типа *bool* могут принимать значения *true* и *false*. Значение переменной *bool* можно инкрементировать, благодаря чему она получит значение *true*, однако нельзя декрементировать ее значение.

Большинство программистов не заметят перемен при переходе от *BOOL* к *bool*. Тем не менее, если вы переходите от версии Visual C++ 4.2 к более новой, следует иметь представление об одной ловушке. В версии 4.2 Standard C++ файлы заголовков используют *typedef* для типа *bool*, приравнивая его к типу *int*. В этой связи в Visual C++ 4.2 *sizeof(bool)* выдавала четыре байта. В Visual C++ 5.0 и последующих версиях значение *sizeof(bool)* составляет один байт. Упомянутое изменение затронет вас только в случае наличия структур, содержащих компоненты данных типа *bool*. В таких случаях файлы данных, записанные в более ранних версиях Visual C++, могут оказаться непригодными.

Вычисление значения m_Canvas

Элемент данных **m_Canvas** — это **CRect**, где хранится положение холста. Холст представляет собой внутреннюю область статической рамки, образованной элементом управления **IDC_CANVAS**. Поскольку элементы управления статической рамкой в своей основе являются обычными окнами **CWnd** под полотном, координаты и размеры клиентской области можно получить с помощью метода **GetClientRect()**, как это делалось ранее. Единственное различие заключается в том, что требуется получить клиентскую область для элемента управления **IDC_CANVAS**, а не клиентскую область для диалогового окна **CPaintORama**. Чтобы выполнить это, в своем распоряжении необходимо иметь сам объект **IDC_CANVAS CWnd**.

Поскольку у программистов Windows часто возникает необходимость в объекте **CWnd**, это нетрудно сделать. Функция **GetDlgItem()**, примененная для диалогового окна, с переданным ей идентификатором элемента управления, возвращает указатель **CWnd** на этот элемент управления. **GetDlgItem()** работает только в окнах, порожденных от **CDialog**; таким окном является **CPaintORamaDlg**. Таким образом, для получения окна **IDC_CANVAS** следует записать

```
CWnd* pCanvas = GetDlgItem(IDC_CANVAS);
```

Теперь, используя указатель **pCanvas**, можно легко найти на экране местоположение элемента управления. Вместо применения функции **GetClientRect()** вызывается функция **GetWindowRect()**, которая возвращает местоположение элемента управления в экранных координатах:

```
pCanvas->GetWindowRect(&m_Canvas);
```

После этого **m_Canvas** содержит местоположение окна **IDC_CANVAS** на экране. Поскольку требуются координаты относительно клиентской области **CPaintORamaDlg**, возвращенное значение должно быть преобразовано. Функция **ScreenToClient()** принимает адрес объекта **CRect**, в данном случае это **m_Canvas**, и заменяет каждую координату на эквивалентную в клиентской области. Вот как это выглядит:

```
ScreenToClient(&m_Canvas);   // Сохранить координаты относительно
   клиентской области
```

И наконец, вы получаете координаты вашего холста — почти. И хотя элементы управления Windows представляют собой настоящие объекты **CWnd**, у них нет границ, которые имеются у окон верхнего уровня и всплывающих окон. В данном случае все окно представляет собой клиентскую область. Вы оставляете на месте обрамление шириной в 2 пиксела, начерченное элементом управления "вдавленной" статической рамкой. Следовательно, потребуется передвинуть поле **CRect**, называемое **left**, на два пиксела вправо, а поле, называемое **right**, — на два пиксела влево и т.д. Метод **DeflateRect()** класса **CRect** служит именно для этих целей. Поскольку функции включающего-исключающего рисования, которыми вы пользуетесь для очистки холста, фактически не рисуют правый или нижний ряд пикселов, ширину правого и нижнего обрамления придется уменьшить только на один пиксел, используя функцию **DeflateRect()** следующим образом:

```
m_Canvas.DeflateRect(2, 2, 1, 1);
```

Завершенная функция **OnInitDialog()** показана в листинге 9.1.

Листинг 9.1. Функция CPaintORamaDlg::OnInitDialog().

```
BOOL CPaintORamaDlg::OnInitDialog()
{
      CDialog::OnInitDialog();
      // Установить пиктограмму диалогового окна.
      // Каркас делает это автоматически, если главное окно
      // приложения не является диалоговым
      SetIcon(m_hIcon, TRUE);  // Установить большую пиктограмму
      SetIcon(m_hIcon, FALSE); // Установить малую пиктограмму

      // ЧТО СДЕЛАТЬ: Здесь добавить дополнительный код инициализации
      // 1. Предположим, что рисование не выполняется
      m_IsDrawing = false;

      // 2. Вычислить истинное местоположение холста
      // Получить указатель на IDC_CANVAS
      CWind* pCanvas = GetDlgItem(IDC_CANVAS);
      // Найти его местоположение на экране
      pCanvas->GetWindowRect(&m_Canvas);
      // Сохранить координаты клиентской области
      m_Canvas.DeflateRect(2, 2, 1, 1);

      return TRUE;  // вернуть TRUE, если не установлен фокус
                    // на какой-то элемент управления
}
```

Добавление кнопки Clear

Чтобы убедиться в том, что все вычисления, выполненные для **m_Canvas**, правильны, приведем кнопку Clear в активное состояние. После выполнения щелчка на кнопке Clear приложение PaintORama должно закрасить белым цветом всю область, ограниченную объектом **m_Canvas**. Для активизации кнопки выполните следующие шаги:

1. Откройте главное диалоговое окно **CPaintORamaDlg** в Dialog Editor.
2. Дважды щелкните на кнопке **IDC_CLEARBTN**. В открывшемся диалоговом окне Add Member Function примите предлагаемую им функцию **OnClearbtn()** и щелкните на OK.
3. Включите программный код, представленный в листинге 9.2, в функцию **OnClearbtn()**.

Листинг 9.2. Функция CPaintORamaDlg::OnClearbtn().

```
void CPaintORamaDlg::OnClearbtn()
{
      // ЧТО СДЕЛАТЬ: Добавить код обработчика уведомлений элементов
      // управления
      // В момент запуска очистить холст
      CClientDC dc(this);
      dc.SelectStockObject(NULL_PEN);
      dc.Rectangle(m_Canvas);
}
```

Функция **OnClearbtn()** получает контекст устройства, связанный с диалоговым окном. Затем она выбирает **NULL_PEN** и включает его в контекст устройства;

таким образом, появятся только заполняющие мазки белой кисти, выбранной по умолчанию. И наконец, в **OnClearbtn()** используется функция **Rectangle()**, с которой вы сталкивались на протяжении нескольких последних глав, для очистки области холста. Обратите внимание, что функция **Rectangle()** перегружена — ей можно передать либо различные углы ограничивающей рамки, либо объект **CRect**, содержащий эти координаты. В данном случае передается объект **m_Canvas**.

Откомпилируйте программу и щелкните на кнопке Clear несколько раз, чтобы убедиться, что она работает. А теперь можно что-нибудь нарисовать.

Как срабатывают сообщения, поступающие от мыши

При перемещении мыши или выполнении щелчков на ее кнопках Windows генерирует одно или большее число сообщений и пересылает их в окно, на котором в данный момент находится курсор мыши. Существует более 20 типов сообщений мыши, и столь большое их количество вызывает некоторое беспокойство. Однако сообщения мыши не являются таким и сложными, какими они кажутся на первый взгляд. Windows проводит различие между действиями мыши, совершаемыми ею в клиентской области, и действиями, которые она совершает в области, отличной от клиентской, наподобие заголовка окна или обрамления. Половина типов сообщений мыши соответствует событиям, имеющим место в неклиентской области — за исключением нескольких специальных ситуаций, их можно игнорировать.

10 оставшихся сообщений, относящихся к клиентской области, включают **WM_MOUSEMOVE**, которое посылается в окно при перемещении указателя мыши в клиентскую область окна. Вы скоро научитесь использовать это сообщение, чтобы определить, где чертить линии. Если переместить указатель мыши на элемент управления Windows, например, на какую-либо кнопку, Windows генерирует сообщения **WM_MOUSEMOVE**, однако эти сообщения не пересылаются в окно приложения, они пересылаются на кнопку, которая осуществляет их внутреннюю обработку.

Каждое из девяти оставшихся сообщений мыши, относящихся к клиентской области, генерируется по причине нажатия кнопки мыши. При нажатии на кнопку, отпускании кнопки и двойном щелчке на кнопке генерируются различные сообщения. С каждой из трех кнопок мыши ассоциированы три упомянутых сообщения, так что всего получается девять сообщений.

В случае нажатия какой-либо кнопки мыши генерируется сообщение **WM_LBUTTONDOWN**, **WM_RBUTTONDOWN** или **WM_MBUTTONDOWN**, в зависимости от того, какая кнопка нажата — **L**eft (левая), **R**ight (правая) или **M**iddle (средняя). (Если у вашей мыши нет средней кнопки, вы никогда не получите набора сообщений, начинающихся с "**WM_M**".)

При отпускании кнопки мыши генерируется сообщение **WM_LBUTTONUP**, **WM_RBUTTONUP** или **WM_MBUTTONUP**. Если дважды быстро нажать и отпустить кнопку мыши, ожидаемая пара сообщений **DOWN/UP** не возникнет. Предположим, что этот эксперимент проводится с левой кнопкой мыши. Windows заменит второе сообщение **WM_LBUTTONDOWN** на сообщение **WM_LBUTTONDBLCLK**. Программа получит такую последовательность сообщений:

- **WM_LBUTTONDOWN**
- **WM_LBUTTONUP**
- **WM_LBUTTONDBLCLK**
- **WM_LBUTTONUP**

WM_LBUTTONDOWN: начинаем рисовать

Когда пользователь программы PaintORama нажимает кнопку мыши на холсте, это означает желание нарисовать линию. Для этого потребуется перехватить сообщение **WM_LBUTTONDOWN**. Выполните следующие шаги:

1. В панели ClassView выберите класс **CPaintORamaDlg**. Щелкните правой кнопкой для открытия контекстного меню, затем выберите Add Windows Message Handler. В результате открывается диалоговое окно New Windows Message And Event Handlers, показанное на рис. 9.6. (Подобный результат можно получить несколькими способами: использовать выпадающее меню Wizard Bar, упоминаемое в главе 1, или главное окно ClassWizard. Целесообразно эти методы изучить; в конечном итоге, скорее всего, будет отдано предпочтение какому-то одному из них.)

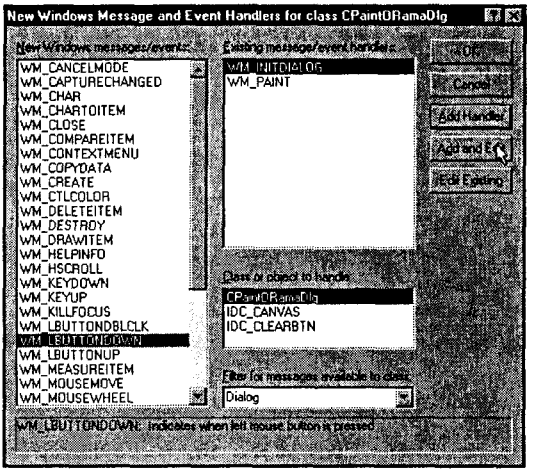


РИСУНОК 9.6. *Добавление обработчика сообщений Windows*

2. Прокрутите список доступных сообщений, пока не найдете сообщения **WM_LBUTTONDOWN**. Выберите его и щелкните на кнопке Add and Edit.
3. Замените код функции **OnLButtonDown()**, сгенерированной Visual C++, кодом, выделенным в листинге 9.3.

Листинг 9.3. Функция CPaintORamaDlg::OnLButtonDown().

```
void CPaintORamaDlg::OnLButtonDown(UINT nFlags, CPoint point)
{
  if (m_Canvas.PtInRect(point))      // Если точка расположена на
  {
       m_IsDrawing = true;           // холсте, то начинаем рисовать
       m_Line_Start = point;         // Это есть новая исходная точка
       SetCapture();                 // Убедитесь в получении
  WM_LBUTTONTP
  }
  // Иначе ничего не делать
}
```

Когда Windows вызывает функцию **OnLButtonDown()**, она передает ей два аргумента. Первый аргумент, **nFlags**, определяет, была ли нажата клавиша Shift или Ctrl в момент выполнения щелчка на кнопке. Например, если необходимо выполнить какое-либо действие при нажатии Ctrl+щелчок, потребуется записать такой программный код:

```
if ( nFlags & MK_CONTROL ) { /* Обработка Ctrl+щелчок */ }
```

Другие константы таковы: **MK_SHIFT**, **MK_LBUTTON**, **MK_RBUTTON** и **MK_MBUTTON**. Например, если должно выполняться какое-то действие, когда одновременно нажаты левая и правая кнопки, воспользуйтесь **MK_RBUTTON**.

Второй аргумент, **point**, — это объект **CPoint**, содержащий данные о местоположении курсора мыши в момент выполнения щелчка.

Функция **CPaintORama::OnLButtonDown()** реализует четыре действия:

1. Проверяет, был ли выполнен щелчок внутри прямоугольника **m_Canvas**. Функция **PtInRect()** класса **CRect** возвращает значение **true**, если указанная точка находится внутри прямоугольника, и значение **false** в противном случае.
2. Если щелчок выполнен внутри прямоугольника **m_Canvas**, это означает, что пользователь намерен начать рисование линии. Таким образом, **m_IsDrawing** устанавливается в **true**.
3. Поскольку функция **OnLButtonDown()** отмечает начало линии, элемент данных **m_LineStart** инициализируется с использованием аргумента **point**.
4. Сообщения мыши пересылаются в окно, на котором находится курсор мыши. Следовательно, если нажать кнопку мыши, находясь на холсте, а отпустить ее после выхода из окна приложения PaintORama, логика, управляющая началом и завершением рисования, будет нарушена. Функция **SetCapture()** гарантирует, что программа получит завершающее сообщение **WM_LBUTTONUP** независимо от того, где будет отпущена кнопка.

WM_LBUTTONUP: Завершение линии

Через некоторое время мы вернемся к программному коду, который фактически выполняет рисование. Сейчас же мы рассмотрим сообщение **WM_LBUTTONUP** — сообщение, останавливающее то, что начала функция **WM_LBUTTONDOWN**.

Включите в приложение обработчик сообщений **OnLButtonUp()**, воспользовавшись той же процедурой, которая применялась для создания функции **OnLButtonDown()**. Функция **OnLButtonUp()** содержит только две строки: одна для установки поля **m_IsDrawing** в **false** (поскольку рисование прекращено) и еще одна для освобождения мыши от захвата, осуществленного в функции **OnLButtonDown()**. Полностью метод представлен в листинге 9.4.

Листинг 9.4. Функция CPaintORamaDlg::OnLButtonUp().

```
void CPaintORamaDlg::OnLButtonUp(UINT nFlags, CPoint point)
{
  m_IsDrawing = false;   // Установить режим рисования в false
  ReleaseCapture();      // Освободить мышь
}
```

Если вы забыли освободить мышь от захвата, вы обнаружите, что не можете закрыть приложение после начала рисования. Не поддавайтесь панике, просто переключитесь на какое-нибудь другое приложение, и Windows освободит мышь от захвата.

WM_MOUSEMOVE: все пикселы в один ряд

Чтобы начертить пикселы на экране, добавьте функцию **OnMouseMove()** для перехвата события **WM_MOUSEMOVE**. Замените программный код, сгенерированный ClassWizard, программным кодом, выделенным в листинге 9.5.

Листинг 9.5. Функция CPaintORamaDlg::OnMouseMove().

```
void CPaintORamaDlg::OnMouseMove(UINT nFlags, CPoint point)
{
  // Рисовать, если m_IsDrawing имеет значение true, указатель мыши
  // находится на холсте и левая кнопка мыши нажата
  if (m_IsDrawing && (nFlags & MK_LBUTTON) &&
  m_Canvas.PtInRect(point))
  {
      CClitntDC dc(this);          // Получить контекст устройства
      dc.MoveTo(m_LineStart);      // Переместиться в начало линии
      dc.LineTo(point);            // Рисовать в текущей позиции
      m_LineStart = point;         // Скорректировать текущее
                                   // положение мыши
  }
}
```

Как это работает

Прежде чем вы начнете рисовать линию, должны быть удовлетворены три предварительных условия:

- Мышь должна находиться в пределах области холста. Проверьте это с помощью функции **PtInRect()**.
- Переменная **m_IsDrawing** должна иметь значение **true**. Если это так, то вы знаете, что пользователь щелкнул в прямоугольнике **m_Canvas** и что исходная позиция для рисования выбрана правильно.
- Пользователь *перетаскивает*, а не *перемещает* мышь. В Windows не предусмотрено отдельного сообщения наподобие **WM_MOUSEDRAG**, так что вы должны различать перетаскивание и перемещение самостоятельно. Однако это совсем не трудно сделать. Просто воспользуйтесь аргументом **nFlags** (который может содержать те же возможные значения, что и для сообщения **WM_LBUTTONDOWN**), а также константой **MK_LBUTTON**. Такая комбинация проверяет, была ли нажата левая клавиша при перемещении мыши — именно тот критерий, посредством которого можно определить, имеет ли место перетаскивание мыши. Комбинация **nFlags** и **MK_LBUTTON** с использованием поразрядного оператора **AND** (**&**) формирует выражение, которое возвращает **0**, если левая кнопка мыши не нажата, и ненулевое значение, если она нажата.

Если удовлетворены все три условия, несложно вычертить сегмент линии. Достаточно выполнить следующие шаги:

1. Получите контекст устройства с помощью функции **GetClientDC()**.
2. Переместитесь в точку, сохраненную в переменной **m_LineStart**, используя для этой цели функцию **MoveTo()**. Обратите внимание на то, что **MoveTo()** перегружена, так что она может принять **CPoint** (что здесь и сделано) или пару целых координат (как в предыдущей главе).

3. Передайте данные о текущей позиции мыши, выступающие ранее как аргумент **point**, в функцию **LineTo()** — другой перегруженный метод контекста устройства.
4. Обновите значение элемента данных **m_LineStart** новым положением мыши, так чтобы следующий раз, когда произойдет перемещение мыши, оно начиналось с новой позиции. (Если хотите немного поразвлечься, закомментируйте эту строку и запустите программу на выполнение. В конечном итоге вы получите довольно любопытные результаты.)

Возможно, возникнет вопрос, почему используется функция **LineTo()** вместо, скажем, функции **SetPixel()**. В случае применения **SetPixel()** вместо **LineTo()** обнаруживается, что вместо рисования линий получается экран, засоренный "пиксельной пылью". Это происходит ввиду того, что сообщение **WM_MOUSEMOVE** не посылается всякий раз, когда мышь перемещается на один пиксел — такое количество сообщений **WM_MOUSEMOVE** просто переполнит систему. Фактическое количество посланных сообщений зависит от аппаратных средств, а также от того, насколь далеко и быстро перемещается мышь. Применение функции **LineTo()** устраняет данную проблему.

После того как все это будет сделано, откомпилируйте и запустите приложение. Как работает приложение PaintORama, можно видеть на рис. 9.7 — мы воспользовались им, чтобы нарисовать очаровательный морской пейзаж. (Если вы считаете, что PaintORama не идет ни в какое сравнение с Photoshop, смею заверить, что изображение, полученное с помощью PaintORama ни в чем не уступает изображению, нарисованному в Photoshop. Плохие изображения получаются не по причине низкого качества инструментов, а из-за неумения того, кто ими манипулирует.)

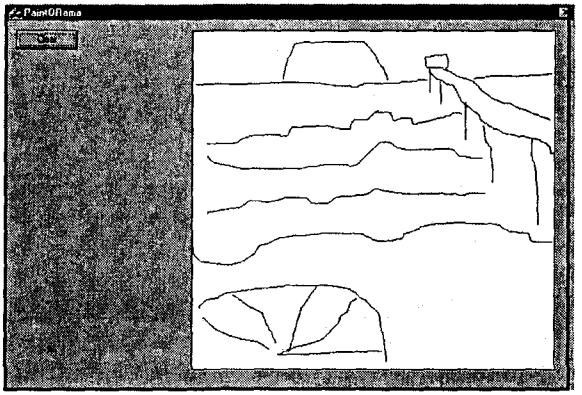


РИСУНОК 9.7. *Выполнение версии 1 приложения PaintORama*

Версия 2 приложения PaintORama: перья на заказ

Программе PaintORama присущ один существенный недостаток: она во всех случаях применяет одно и то же черное перо. Вы уже знаете, как создавать сплошные и штриховые перья, а также перья, использующие специальные стили. Дабы применить приобретенный опыт при разработке программы PaintORama, нужно просто добавить интерфейс, который позволит *пользователям* выбрать желаемое перо.

Напоминаем, что перо характеризуется тремя параметрами: стиль, ширина и цвет. Для изменения этих параметров будут применяться различные элементы управления Windows. Давайте будем менять значения толщины пера при помощи счетчика и сопровождающего его элемента управления.

Счетчик выполняет только то, что определил ему "партнер"

Счетчик представляет собой один из 15 новых элементов управления, реализованных в Windows 95. Эти новые элементы управления называются *общими элементами управления Win32*. Последнее предпринято для того, чтобы отличить их от общих элементов управления, реализованных как часть базовых библиотек Windows. Новые элементы управления реализованы в файле Comctl32.dll, и в MFC они используются наравне с более старыми элементами.

Счетчик подобен очень маленькой линейке прокрутки, у которой удалена центральная часть. Он применяется в сочетании со вторым элементом управления, именуемым *партнером счетчика* (spin buddy), отвечающим за прием и отображение целых чисел. Счетчик может быть вертикальным (его стрелки направлены вверх и вниз) или горизонтальным (стрелки направлены вправо и влево). Когда вы щелкаете на одной из стрелок, счетчик увеличивает или уменьшает целочисленное значение, отображенное его партнером. Обычно в качестве партнера используется элемент управления редактированием или элемент управления статическим текстом.

Перья в группе

Прежде чем включать в PaintORama новые элементы управления, целесообразно добавить группу, дабы внести в программу организационный момент. Вот что надо сделать:

1. Откройте окно Dialog Editor и убедитесь в том, что отображается главное диалоговое окно программы PaintORama.
2. Перетащите группу из панели Control. Элемент управления группой показан на рис. 9.8.
3. Расположите группу под кнопкой Clear и расширьте ее, используя рис. 9.9 в качестве шаблона для размещения элементов управления. (Индикатор положения в строке состояния должен показывать что-то типа 7, 31, а индикатор размеров должен принимать значение примерно 128, 87.) Заголовок группы должен выглядеть как "Стили пера". (Если перед "Р" поставить амперсанд, Windows подчеркнет символ и позволит без проблем переключиться на элементы управления внутри группы при помощи клавиатуры.)

Добавление счетчика и его партнера

После размещения группы перетащите в диалоговое окно элемент управления редактированием и счетчик и разместите их так, как показано на рис. 9.10. (Возможно, эту операцию будет выполнить проще, если включить компоновочную сетку.) Важно добавить элемент управления редактированием первым, а уж за ним счетчик — счетчик будет искать элемент управления, добавленный непосредственно перед ним, и воспримет его в качестве своего партнера. Если же первым добавить счетчик, он примет группу "Стили пера" за элемент управления редактированием — абсолютно не то, что требовалось.

Расположите элемент управления редактированием по своему усмотрению; во время выполнения счетчик самостоятельно "перепрыгнет" внутрь элемента управления редактированием и будет отображаться там. Если сделать элемент управле-

ния редактированием чрезмерно малым, появляется риск, что счетчик его полностью закроет.

Установка параметров счетчика

Установка параметров счетчика выполняется в диалоговом окне Spin Properties (Параметры счетчика). Имя **IDC_SPIN1**, принятое по умолчанию, можно оставить, но обязательно потребуется изменить следующие свойства:

- Отметьте флажок Auto Buddy (Автоматический выбор партнера). Если этого не сделать, придется явно назначить партнера счетчику в коде.
- Отметьте флажок Set Buddy Integer (Целые значения для партнера). Обычно применяются счетчики с целыми значениями. Установка этого флажка автоматизирует процесс. Если необходимо использовать другие типы значений, флажок не должен отмечаться. В случае других типов придется вручную реализовать код, обеспечивающий установку и получение значений партнера.
- Отметьте флажок Arrow Keys (Кнопка со стрелками). Это позволит оперировать счетчиком при помощи как клавиатуры, так и мыши. Все другие флажки должны остаться неотмеченными.
- В выпадающем списке Orientation (Ориентация) выберите Vertical (Вертикальная), чтобы маленькие стрелки указывали вверх и вниз.
- В выпадающем списке Alignment (Выравнивание) выберите Right (Вправо). Эта опция заставляет счетчик перемещаться внутрь своего партнера во время перерисовки экрана. В случае выбора Left (Влево) счетчик переместится внутрь своего партнера и выровняется по его левой стороне.

Диалоговое окно Spin Properties показано на рис. 9.11.

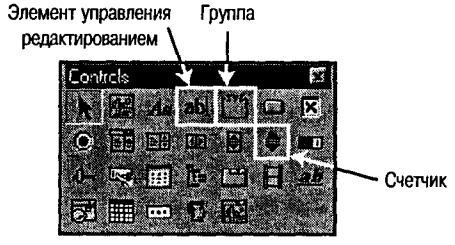


РИСУНОК 9.8. *Группа, счетчик и элемент управления редактированием*

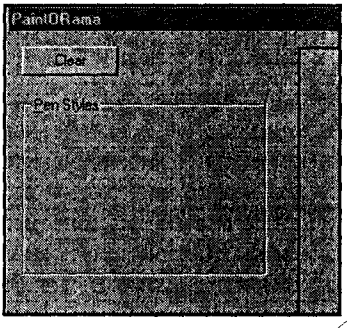


РИСУНОК 9.9. *Размещение группы "Стили пера"*

Установка параметров элемента управления редактированием

После успешного решения всех вопросов, связанных со счетчиком, наступило время позаботиться о его партнере. Visual C++ IDE существенно упрощает эту задачу. Потребуется выполнить следующие шаги:

1. Присвойте элементу управления редактированием имя **IDC_PENWIDTH**.
2. Откройте окно ClassWizard и выберите страницу Member Variables. Отыщите и выберите **IDC_PENWIDTH**, после чего щелкните на кнопке Add Variable. В диалоговом окне Add Member Variable установите имя в **m_PenWidth**,

категорию — в Value и тип — в **int**. На рис. 9.12 экран показан таким, каким он должен быть. Щелкните на OK.

3. Когда ClassWizard появится снова, диалоговое окно будет содержать пару элементов управления редактированием, в которые следует ввести минимальное и максимальное значения для **m_PenWidth**, которые составляют диапазон допустимых значений, проверяемый MFC. Установите минимальное значение равным 1 (самое тонкое перо), а максимальное значение равным 32 (максимально толстое перо); Windows, однако, способна отображать перья большей ширины, так что можно свободно экспериментировать с различными значениями. Если пользователь попытается ввести значение, выходящее за пределы заданного диапазона, либо дробное значение, Windows выведет на экран сообщение об ошибке. Диалоговое окно должно иметь вид, представленный на рис. 9.13. По завершении щелкните на OK.

Связывание с кодом

Как только счетчик будет подключен к своему партнеру, для генерирования перьев различной ширины останется выполнить всего лишь три операции:

- Инициализируйте счетчик и партнер приемлемыми значениями в момент запуска приложения. В противном случае программа в момент запуска будет приветствовать пользователя сообщением об ошибке выхода за пределы допустимого диапазона, что не вселит уверенность в ваших возможностях.

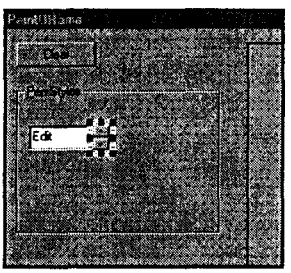


РИСУНОК 9.10. *Установка элементов управления, отвечающих за изменение ширины пера*

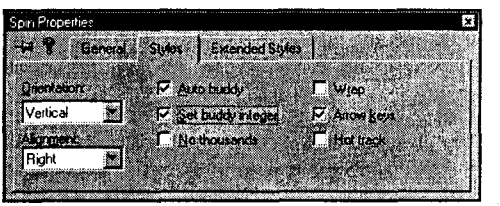


РИСУНОК 9.11. *Диалоговое окно Spin Properties*

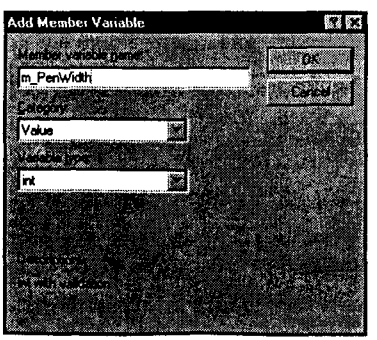


РИСУНОК 9.12. *Добавление элемента данных m_PenWidth*

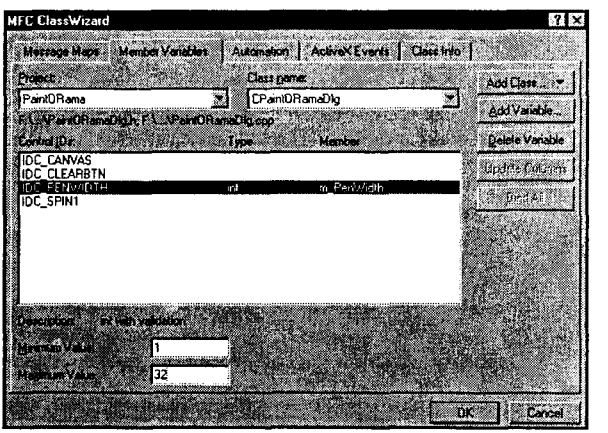


РИСУНОК 9.13. *Добавление проверки вводимых целых значений*

- Установите такой диапазон значений счетчика, который согласуется с диапазоном значений партнера. (Установка минимального и максимального значений в окне ClassWizard затрагивает только элемент управления редактированием, но не счетчик.)
- Каждый раз когда начинается очередной рисунок, убедитесь, что используется перо, с шириной, которую определил пользователь. Просто, получите значение ширины пера у партнера счетчика, а затем создайте новое перо в функции **OnMouseDown()** всякий раз, когда чертится линия.

Давайте решать эти проблемы последовательно.

Инициализация элемента данных m_PenWidth

Может показаться, что поскольку ClassWizard создает и проверяет переменную **m_PenWidth**, он должен также выполнять ее инициализацию. К сожалению, ClassWizard выполняет инициализацию не столь мастерски, как этого хотелось.

Когда ClassWizard создает переменную, ассоциированную с элементом управления, такую как **m_PenWidth**, задачу передачи значения из этой переменной в элемент управления и из элемента управления обратно в переменную выполняет MFC. Это производится автоматически в определенные моменты времени периода жизни диалогового окна. Можно сделать так, чтобы такая операция выполнялась вручную, вызывая функцию **UpdateData()** и передавая ей значение **TRUE** (для передачи информации из элемента управления в переменную) или **FALSE** (для передачи информации из переменной в элемент управления).

Когда диалоговое окно создается впервые, библиотека MFC обращается к функции **UpdateData(FALSE)** для передачи значения из переменной **m_PenWidth** в элемент управления **IDC_PENWIDTH**. Последнее означает, что переменная **m_PenWidth** типа **int** должна существовать до момента построения окна и должна хранить допустимое значение. Это может произойти только в конструкторе класса **CPaintORamaDlg**.

Если посмотреть на конструктор в листинге 9.6, можно обнаружить в его середине фрагмент, обрамленный комментариями, содержащими слово **AFX_DATA_INIT**. Между этими комментариями **m_PenWidth** инициализируется значением **0**. В этом месте ClassWizard автоматически инициализирует все созданные им управляющие переменные. К сожалению, он игнорирует тот факт, что допустимые значения переменной **m_PenWidth** находятся в пределах от 1 до 16.

Листинг 9.6. Конструктор класса CPaintORamaDlg.

```
CPaintORamaDlg::CPaintORamaDlg(CWnd* pParent /*=NULL*/)
      : CDialog(CPaintORamaDlg::IDD, pParent)
{
   //{{AFX_DATA_INIT(CPaintORamaDlg)
   m_PenWidth = 0;
   //}}AFX_DATA_INIT
   // Обратите внимание, что в Win32 LoadIcon не требует
   последующего
   // вызова DestroyIcon
   m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
   m_PenWidth = 1;
}
```

СОВЕТ

Не вмешивайтесь в священный код!

Несмотря на то что редактор не запрещает вносить изменения в "священный" программный код в специальных блоках, поддерживаемых ClassWizard, нет никаких гарантий, что внесенные вами изменения сохранятся до следующего раза, когда они потребуются. Как правило, если код появляется в редакторе в "притушенном" (сером) виде, его не следует изменять.

Может возникнуть желание вмешаться и "исправить" недосмотры ClassWizard, отредактировав код, содержащийся в разделе **AFX_DATA_INIT**. Не поддавайтесь такому искушению! Вместо этого просто присвойте **m_PenWidth** требуемое значение после блока автоматической инициализации (см. листинг 9.6).

Инициализация счетчика

Нет возможности инициализировать счетчик в конструкторе, как это имело место в случае **m_PenWidth**. В этот момент счетчик еще не существует. Его следует инициализировать в методе **OnInitDialog()**, после инициализации элементов данных **m_Canvas** и **m_IsDrawing**.

Указатель на счетчик можно получить, обратившись к функции **GetDlgItem(IDC_SPIN1)**. Вспомните, что **GetDlgItem()** возвращает указатель **CWnd**. Поскольку класс **CSpinButtonCtrl** счетчика происходит из **CWnd**, все работает хорошо. Тем не менее, если необходимо вызвать какую-либо функцию, специфическую для счетчика, возвращенный указатель следует привести к **CSpinButtonCtrl***, например, так:

```
CSpinButtonCtrl* pSpin = (CSpinButtonCtrl*)GetDlgItem(IDC_SPIN1);
```

Как только указатель на счетчик получен, им можно воспользоваться для вызова функций **SetRange()** и **SetPos()**. Функция **SetRange()** принимает два аргумента: максимальное и минимальное значения для счетчика. Если вызов **SetRange()** завершается неудачей, по умолчанию используется диапазон значений от 100 до 0. Само собой разумеется, пользователям не доставляет особой радости, когда они обнаруживают, что щелчок на стрелке вверх уменьшает числовые значения, поэтому обычно производится вызов функции **SetRange()**. Функция **SetPos()** просто присваивает счетчику начальное значение. После вызова функции **GetDlgItem()** добавьте две таких строки:

```
pSpin->SetRange(1, 32);
pSpin->SetPos(1);
```

Инициализация пера

Каждый раз когда пользователь начинает рисовать новую линию, в **OnMouseDown()** создается новое перо. Таким образом, необходимы средства, обеспечивающие существование пера к моменту начала рисования. Это реализуется за счет добавления в класс **CPaintORamaDlg** нового поля **CPen**.

Поскольку вы уже набили руку на добавлении элементов данных в класс, рассматривать необходимые для этого действия еще раз мы не будем. Присвойте элементу данных имя **m_Pen**. Выполнять инициализацию **m_Pen** не требуется.

Вы еще на забыли, как работает двухшаговое создание? Сначала создается объект **CPen** с использованием конструктора по умолчанию, а затем для завершения инициализации вызывается метод **CreatePen()**. Вызывать **CreatePen()** второй раз для существующего объекта **CPen** нельзя. Однако это именно то, что хотелось бы здесь сделать — каждый раз, когда вызывается функция **OnMouseDown()**, неплохо было бы вызвать функцию **m_Pen.CreatePen()** и передать ей новые (возможно, измененные) свойства пера. Нет проблем — просто "отцепите" предшествующее перо, связанное с **m_Pen**, при помощи вызова метода **DeleteObject()**. После этого появляется возможность вызвать **CreatePen()** еще раз.

Как бы сказал герой телевизионного сериала детектив Коломбо, "Имеется еще один вопрос". Поскольку для создания нового пера необходимо использовать переменную **m_PenWidth**, следует позаботиться о получении новейшей информации из счетчика и его партнера. Для этого перед построением нового пера выполняется вызов функции **UpdateData(TRUE)**.

Обновленная версия программного кода **OnLButtonDown()** приведена в листинге 9.7.

Листинг 9.7. Построение нового пера в OnLButtonDown().

```
void CPaintORamaDlg::OnLButtonDown(UINT nFlags, CPoint point)
{
   if (m_Canvas.PtInRect(point))      // Если точка находится на
   {
         m_IsDrawing = true;          // холсте, то мы рисуем
         m_LineStart = point;         // А это новая исходная точка
         SetCapture();                // Убедитесь в том, что получено
   WM_LBUTTONUP
      // Построить m_Pen
      UpdateDate(TRUE);               // Получить новые значения из
                                      // элементов управления
      m_Pen.DeleteObject();
      m_Pen.CreatePen(PS_SOLID, m_PenWidth, RGB(0, 0, 0));
   }
   // Иначе ничего не делать
}
```

Активизация пера

Путь, который мы проделали, был долог, но конец уже виден. Следует освоить решение еще одной задачи, требующей написания кода, прежде чем вы получите в свое распоряжение линии различной толщины. Новое перо необходимо активизировать. К счастью, делается это легко.

Весь программный код рисования в PaintORama в данный момент сосредоточен в методе **OnMouseMove()**. Для активизации нового пера включите следующие строки после строк, создающих новый контекст устройства:

```
// Здесь находится код создания DC
dc.SelectObject(&m_Pen);
```

Теперь во время рисования вместо пера по умолчанию можно использовать новое пером. Однако осталась нерешенной еще одна маленькая проблема. Откомпилируйте и запустите программу PaintORama (естественно, после добавления в нее строки, реализующей выбор нового пера). Выберите самое толстое перо, скажем, шириной 16 или 32 пиксела, и начертите линию поблизости к краю области

рисования. Результат показан на рис. 9.14; легко заметить, что стрелка выходит за пределы области холста.

Очевидно, что проверки условия выхода пера за пределы холста оказывается недостаточно, если толщина пера превышает один пиксел. Несмотря на то что начало линии находится в пределах холста, из-за большой ширины пера линия как бы "проливается" через стенку.

К счастью, эта проблема решается достаточно просто: попросите Windows задействовать механизм отсечения, чтобы вывод не проникал туда, где он не желателен. Добавьте следующую строку

```
dc.IntersectClipRect(m_Canvas);
```

в метод **OnMouseMove()** после строки, создающей контекст устройства. Теперь Windows будет отсекать все, что попадет за пределы прямоугольника, определенного переменной **m_Canvas**.

Смотрите нас на следующей неделе, когда...

Как и любой захватывающий сериал, версия 2 приложения PaintORama оставляет вас в состоянии нетерпеливого ожидания продолжения. Вот уже доступны перья переменной ширины, однако как насчет цветов? Как насчет стилизованных перьев? Кистей? Как насчет...?

К сожалению, придется подождать. Но не слишком долго — переключитесь на первый интригующий эпизод главы 10, где сможете найти все, что когда-либо хотели знать о перьях (и заодно ознакомиться с несколькими элегантными строками кода).

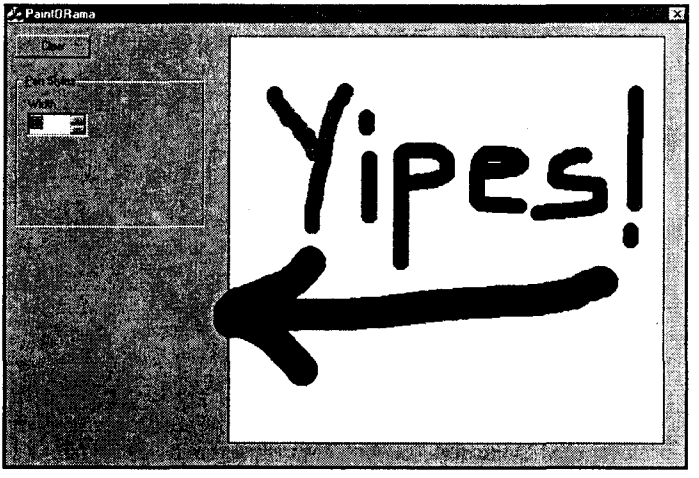


РИСУНОК 9.14.
Выход рисуемой линии за пределы холста.

Глава 10

Приложение PaintORama: новая версия программы

Некоторые люди считают, что следующая версия никогда не бывает лучше исходной. Зачастую они правы. Однако иногда продолжение оказывается ничуть не хуже исходной версии — и это лучший вариант.

Когда Джин Родденберри (Gene Roddenberry) создавал телевизионное шоу *Star Trek*, он не мог предвидеть, насколь богатую жилу он открыл — ее все еще разрабатывают спустя 30 лет. С большинством телевизионных шоу и фильмов такое не случается. Лишь немногие убеждены в высоком качестве "пляжных" кинофильмов шестидесятых либо фильмов семидесятых годов о байкерах.

Время, однако, показало, что шоу "Star Trek" — это не обычный основоположник течения в искусстве. Хотя заложенные в нем идеи и предпосылки многократно использовались и разрабатывались, оно отнюдь не истощалось и не деградировало, становясь пустой оболочкой, а, напротив, росло и расширялось.

Аналогично, программное обеспечение также должно развиваться и изменяться, если его разработчики хотят, чтобы оно выжило. В этой главе приложение PaintORama представляет собой скачок вперед, предлагая возможности выбора цветов, стилей, стилизованных линий, кистей и форм. Наряду с этим вы столкнетесь с рядом новых свойств MFC:

- *Класс **CColorDialog*** — Представитель обширной категории сложных, но тем не менее, оказывающих существенную помощь структур, известных как общие диалоговые окна Windows.
- *Классы **CComboBox** и **CListBox*** — Близкие родственники в генеалогическом древе элементов управления Windows.

Версия 3 приложения PaintORama: цвета и стили

Когда последний раз мы расстались с непритязательной, но честной программой PaintORama, в нее были подключены достопочтенный Счетчик и верный ему партнер — Элемент управления редактированием. Тем не менее, несмотря на присутствие этих двух друзей, не все благоприятно складывается для госпожи Раскраски; можно сказать, что она фактически одноцветная. "Все что мне нужно, — это немного цвета в моей жизни", — говорит Раскраска сама себе, и в один прекрасный день отправляется на его поиски, а за ней по пятам следуют Счетчик и Редактирование.

Колоритное дополнение

Для управления цветом в программе PaintORama мы должны реализовать *образец цвета, на котором можно щелкать* (или просто образец цвета). Это не встроенный элемент управления Windows — это просто небольшой квадрат, в котором воспроизводится цвет текущего пера, имитируя аппаратный образец раскраски. Однако если щелкнуть на таком образце, изменится цвет рисующего пера.

Вы увидите, как эта штука работает чуть-чуть позже. Сейчас же мы приступаем к созданию требуемой инфраструктуры, добавляя новые переменные и компонуя интерфейсы.

Построение основы

Для начала необходимо иметь по меньшей мере одну новую переменную для хранения цвета текущего пера, поскольку новое перо создается всякий раз, когда рисуется линия. Новая переменная — **m_PenColor** — должна быть приватной **COLORREF**. Для добавления переменной можно воспользоваться диалоговым ок-

ном Add Member Variable или просто вставить приведенную ниже строку в программу сразу после определения класса **CPaintORamaDlg**.

```
COLORREF m_PenColor;
```

Переменную **m_PenColor** потребуется также проинициализировать. Лучше всего это сделать в конструкторе класса, куда необходимо поместить следующую строку, задающую черный цвет в качестве исходного цвета пера:

```
m_PenColor = RGB(0, 0, 0);
```

Теперь приступим к компоновке интерфейса. Откройте окно Dialog Editor, перетащите элемент управления статическим текстом и элемент управления изображением из панели Control и поместите их справа от счетчика. (Точное местоположение пока роли не играет.) Измените заголовок элемента управления статическим текстом на "Color" (Цвет), а его идентификатором пусть остается **IDC_STATIC**. Измените идентификатор элемента управления изображением на **IDC_PENCOLOR**, его Type — на Rectangle и Color — на Black, как показано на рис 10.1.

Кроме того, отметьте флажки Sunken и Notify на странице Styles диалогового окна Picture Properties (см. рис. 10.2).

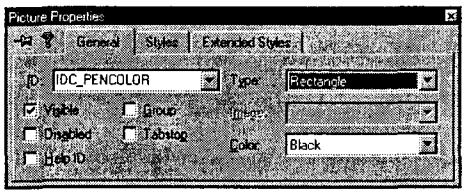


РИСУНОК 10.1. *Установки для элемента управления IDC_PENCOLOR в диалоговом окне Picture Properties.*

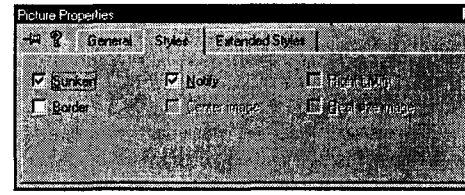


РИСУНОК 10.2. *Дополнительные установки для элемента управления IDC_PENCOLOR в диалоговом окне Picture Properties.*

И наконец, прежде чем приступать к написанию кода, расположите новые элементы управления так, как показано на рис. 10.3.

Цвета по заказу: CColorDialog

Сейчас вы, наконец, готовы к тому, чтобы написать код, который даст возможность пользователям выбирать цвета по своему усмотрению. Упомянутую возможность несложно реализовать множеством способов — через набор переключателей, соответствующих различным цветам, или, возможно, при помощи списков либо полей со списками. Каждый из способов вполне осуществим, что и будет продемонстрировано. Однако имеются лучшие способы выбора цветов — за счет использования экземпляра встроенного класса **CColorDialog**, одного из *общих диалоговых окон Windows*.

Что представляют собой общие диалоговые окна?

Одна из первоначальных целей Windows (и графических пользовательских интерфейсов вообще) состояла в стандартизации работы приложений. Например, когда различные приложения используют одну и ту же структуру меню, пользователи с большей легкостью осваивают программы и могут быстро переключиться с одной программы на другую.

Ранние программы Windows не достигали этой цели, когда дело касалось дизайна диалоговых окон. В каждой Windows-программе диалоговые окна, которые открывали файлы и печатали документы, разрабатывались с различным дизайном, при всем при том, что эти операции фактически аналогичны на всех машинах. В этой связи, начиная с Windows 3.0, Microsoft добавила в Windows шесть встроенных диалоговых окон, каждое из которых ориентировано на выполнение конкретной операции общего назначения. В табл. 10.1 перечислены эти шесть диалоговых окон вместе с именами их классов в MFC.

Таблица 10.1. Общие диалоговые окна Windows

Класс диалогового окна	*Описание*
CColorDialog	Интерактивный выбор цвета
CFileDialog	Интерактивный выбор файла для открытия или сохранения
CFindReplaceDialog	Интерактивный поиск и замена строк
CFontDialog	Интерактивный выбор шрифта
CPageSetupDialog	Интерактивный выбор размеров страницы
CPrintDialog	Интерактивный выбор принтера

Легко видеть, что каждое диалоговое окно позволяет выбирать конкретный ресурс, такой как шрифт, цвет или принтер.

Использование диалогового окна CColorDialog

В приложении PaintORama для выбора цвета будет использован экземпляр класса **CColorDialog**, который открывается, когда пользователь щелкает на прямоугольнике **IDC_PENCOLOR**. (В этой связи элемент управления **IDC_PENCOLOR** должен иметь стиль **SS_NOTIFY**, что достигается отметкой флажка Notify в диалоговом окне Picture Properties, как показано на рис 10.2. Если это не сделать, сообщение об ошибке не появится, однако и диалоговое окно Color не отобразится.)

Откройте окно Dialog Editor с главным диалоговым окном приложения PaintORama, а затем выполните следующие шаги:

1. Дважды щелкните на элементе управления **IDC_PENCOLOR** в окне Dialog Editor. Поскольку с этим элементом управления не связана ни одна функция, появляется диалоговое окно Add Member Function (см. рис. 10.4) с предложением добавить функцию **OnPencolor**. Оставьте все, что предлагается по умолчанию, и щелкните на OK.

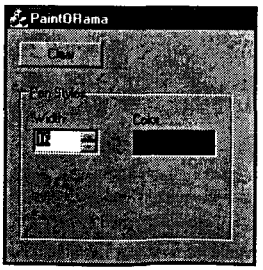


РИСУНОК 10.3. *Позиционирование элементов управления приложения PaintORama*

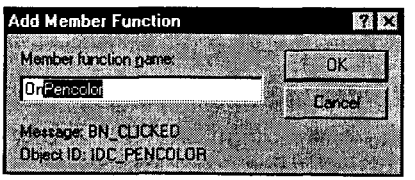


РИСУНОК 10.4. *Добавление метода OnPencolor()*

2. Добавьте к программному коду, сгенерированному ClassWizard, код, выделенный в листинге 10.1.

Листинг 10.1. Функция CPaintORamaDlg::OnPencolor().

```
void CPaintORamaDlg::OnPencolor()
{
    CColorDialog dlg(m_PenColor);
    if (dlg.DoModal() == IDOK)
    {
        m_PenColor = dlg.GetColor();
    }
}
```

3. Поместите строку, которая вызывает функцию **CreatePen()**, ближе к концу функции **OnLButtonDown()**. Измените аргумент цвета с **RGB(0, 0, 0)** на **m_PenColor**. Строка примет такой вид:

```
m_Pen.CreatePen(PS_SOLID, m_PenWidth, m_PenColor);
```

4. Откомпилируйте и запустите приложение PaintORama. Щелкните на образце цвета, в результате чего откроется общее диалоговое окно Color, как показано на рис. 10.5. (На этом рисунке окно показано полностью развернутым, что происходит после щелчка на кнопке Define Custom Colors (Определить пользовательский цвет).)

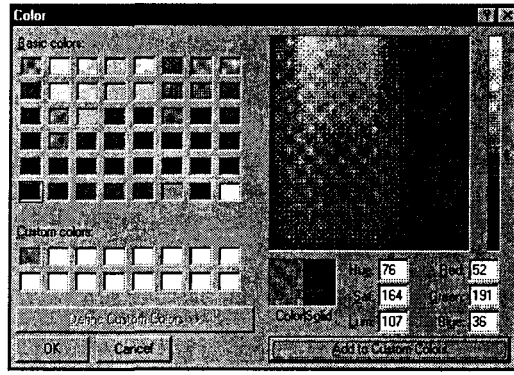


РИСУНОК 10.5.
Диалоговое окно Color

5. Выберите цвет в диалоговом окне Color, затем щелкните на OK. С помощью счетчика измените ширину пера и попробуйте нарисовать несколько линий. Теперь имеется возможность создавать перо любой ширины и цвета, что легко заметить на рис. 10.6. После рисования нескольких произвольных линий посмотрим, как работает класс **CColorDialog**. Затем мы займемся раскраской переключателя цвета, который сейчас остается черным и мрачным независимо от того, каким бы ярким перо не было.

Как работает класс CColorDialog

Класс **CColorDialog** является одним из простейших в использовании. Объект класса **CColorDialog** создается следующим образом:

```
CColorDialog dlg(RGB(255, 0, 0));
```

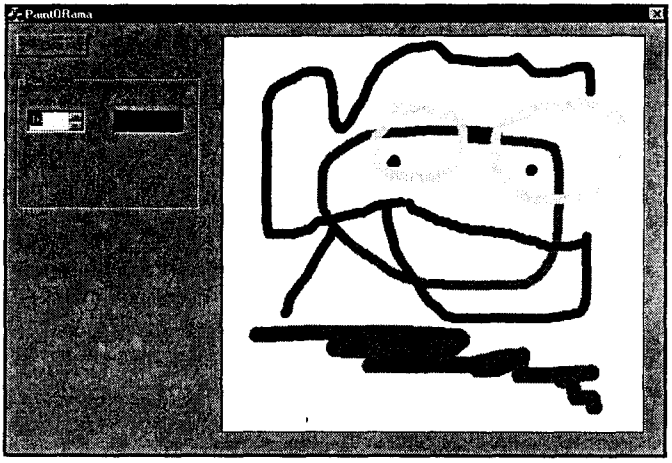


РИСУНОК 10.6.
Рисование цветным пером

Фактически для конструктора требуется задать три аргумента, однако для всех аргументов определены значения по умолчанию, поэтому любые аргументы можно опустить. В конструктор обычно передается первый аргумент, который задает начальный выбор цвета. Второй аргумент определяет набор флажков, позволяющих настроить **CColorDialog** по собственному усмотрению; он используется реже. Третий аргумент используется еще реже; он представляет собой указатель на родительское окно.

Созданный объект **CColorDialog** начинает свою жизнь в памяти компьютера, но на экране не появляется. Вызов функции **DoModal()** приводит к открытию диалогового окна и предоставлению возможности выбирать цвета. Если пользователь закрывает это диалоговое окно, щелкнув на OK, **DoModal()** возвращает значение **IDOK** — в этом случае значение переменной, хранящей цвет изменяется на выбранное пользователем. Если же пользователь закрывает диалоговое окно, нажав Esc, либо щелкнув на Cancel, либо прибегнув к помощи кнопки Close, **DoModal()** возвращает значение **IDCANCEL**.

Используя функцию **GetColor()**, можно получить цвет, выбранный пользователем. Общепринято это делать только в тех случаях, когда возвращаемым **DoModal()** значением является **IDOK**. Сейчас, поскольку отмена диалогового окна не изменяет выбранного цвета, код в функции **OnPenColor()** можно заменить на:

```
CColorDialog dlg(m_PenColor);
dlg.DoModal();
m_PenColor = dlg.GetColor();
```

Несмотря на то что после возврата из **DoModal()** диалоговое окно на экране больше не отображается, сам объект диалогового окна — в данном случае **dlg** — остается в памяти. До тех пор пока переменная **dlg** остается в области видимости, при помощи функции **GetColor()** можно получить выбранный цвет.

Раскрашивание индикатора

Прежде чем завершить тему цветных перьев и перейти к стилям, рассмотрим, что нужно для раскрашивания образца цвета **IDC_PENCOLOR** текущим цветом. Потребуется выполнить три задачи:

- Создать и проинициализировать элемент данных типа **CRect**, сохраняющий размеры образца цвета, как это делалось ранее для структуры холста.
- Создать временную кисть с цветом пера, которым вы рисуете.
- Воспользоваться временной кистью для закрашивания **CRect** всякий раз при изменении цвета пера.

Внесите необходимые изменения в приложение PaintORama:

1. Создайте новый элемент данных типа **CRect** с именем **m_PenColorSwatch**.
2. Проинициализируйте переменную, добавив в конец функции **OnInitDialog()** программный код из листинга 10.2. Внимательный читатель заметит, что этот код позаимствован из программы, применяемой для инициализации переменной **m_Canvas**; изменены только некоторые имена.

Листинг 10.2. Инициализация переменной m_PenColorSwatch (код добавляется в конец функции OnInitDialog()).

```
// 4. Вычислить корректные координаты образца цвета
// Получить указатель на CWnd
CWnd* pPenColor = GetDlgItem(IDC_PENCOLOR);
// Найти местоположение образца цвета на экране
pPenColor->GetWindowRect(&m_PenColorSwatch);
// Сохранить клиентские координаты
ScreenToClient(&m_PenColorSwatch);
m_PenColorSwatch.DeflateRect(2, 2, 1, 1);
```

3. Добавьте выделенные в листинге 10.3 строки в функцию **OnPencolor()**. Функция **FillRect()** берет указатель на объект **CRect** и указатель на объект **CBrush**, а затем заполняет прямоугольник с использованием кисти.

Листинг 10.3. Закрашивание образца цвета пера в функции OnPencolor().

```
void CPaintORamaDlg::OnPencolor()
{
      CColorDialog dlg(m_PenColor);
      if (dlg.DoModal() == IDOK)
      {
         m_PenColor = dlg.GetColor();
         Cbrush swatch;
         swatch.CreateSolidBrush(m_PenColor);
         CClientDC dc(this);
         dc.FillRect(&m_PenColorSwatch, &swatch);
      }
}
```

Пока об этом все. Теперь после выбора цвета пера образец цвета визуально отображает произведенный выбор. Откомпилируйте программу и запустите ее на выполнение.

Переключатели и стилизованные перья

Итак, все в порядке — вы получили в свое распоряжение толстые перья с любыми цветами. Теперь все, что необходимо — это некоторая совокупность элегантных перьев. Поскольку MFC предлагает только семь стилей пера, их легко отобразить при помощи переключателей. (Однако подобного рода подход неприме-

ним в случае цветов; согласитесь, что диалоговое окно с 256 переключателями — это явный перебор. А если пользователь располагает 24-разрядным видеоадаптером, то мысль использовать для каждого цвета отдельный переключатель не должна даже и возникать.)

Добавление интерфейса

Для помещения в PaintORama переключателей стилей потребуется выполнить следующие шаги:

1. Перетащите семь переключателей из панели Control в нижнюю часть группы Pen Styles (Стили пера). Выстройте их в два столбца: в первом — четыре переключателя, а остальные — во втором.
2. Двигаясь сверху вниз по первому столбцу, а затем перейдя в начало следующего, присвойте кнопкам следующие имена: Solid Pen, Dash Pen, Dot Pen, Dash-Dot, Dash-Dot-Dot, Null Pen, Inside Frame. Диалоговое окно должно приобрести вид, показанный на рис. 10.7.
3. Измените идентификатор *первого* переключателя на **IDC_SOLID_PEN**. Не забудьте также отметить флажок Group; если этого не сделать, автоматическая групповая передача состояния переключателей не произойдет. Диалоговое окно Radio Button Properties показано на рис. 10.8.

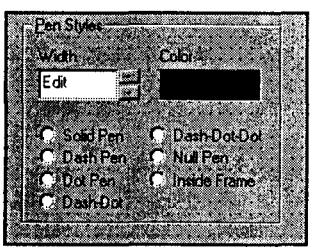


РИСУНОК 10.7. *Переключатели в группе Pen Styles*

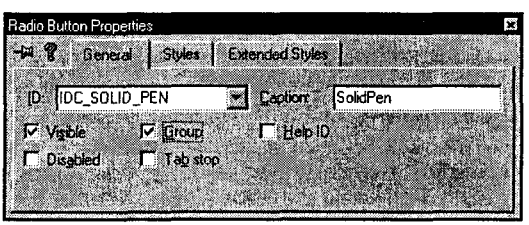


РИСУНОК 10.8. *Опции диалогового окна Radio Button Properties для переключателя IDC_SOLID_PEN*

4. После выбора переключателя **IDC_SOLID_PEN** откройте ClassWizard и перейдите в нем на страницу Member Variables. Найдите **IDC_SOLID_PEN** и щелкните на кнопке Add Variable. В диалоговом окне Add Member Variable введите "**m_PenStyle**" в качестве имени элемента данных, "Value" — в качестве Category и "int" — в качестве Type. В нижней части диалогового окна присутствует сообщение "radio button group transfer" (групповая передача состояния переключателей), как показано на рис. 10.9. Щелкните на OK. Еще раз щелкните на OK для закрытия ClassWizard.
5. Для того чтобы групповая передача состояния переключателей выполнялась правильно, очень важно корректно установить *порядок обхода элементов управления (tab order)*. Windows и MFC используют эту информацию для внутренних целей, чтобы определить, какие элементы управления с какими объединены. Для установки порядка обхода убедитесь в том, что выбрано диалоговое окно приложения PaintORama, а затем выберите Layout|Tab Order (Компоновка|Порядок обхода) из меню (или нажмите Ctrl+D). В окне появится набор небольших пронумерованных меток голубого цвета (см. рис. 10.10). Щелкайте на каждом элементе управления в порядке, показанном на рис. 10.10, тем самым устанавливая корректный порядок обхода.

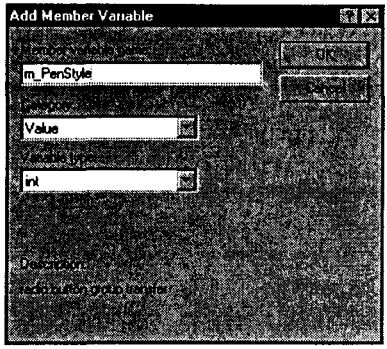


РИСУНОК 10.9. *Добавление элемента данных m_PenStyle в приложение PaintORama*

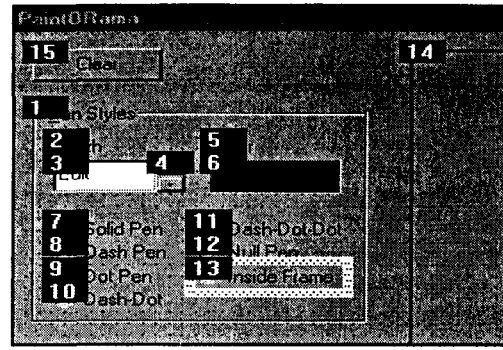


РИСУНОК 10.10. *Изменение порядка обхода элементов управления диалогового окна*

Запись кода

В главе 6 вы сталкивались с переключателями, однако, по-видимому, еще не уяснили, насколь просто работать с переключателями в Visual C++. После щелчка на любом из переключателей в PaintORama MFC берет на себя обязанность сохранить индекс переключателя в переменной **m_PenStyle**. Первому переключателю присвоен номер 0, второму — 1 и т.д. Поскольку стиль пера также представлен целыми значениями, начиная с 0 для **PS_SOLID**, никакие преобразования не потребуются — это значение можно использовать для представления стиля пера без изменений.

Как и в случае счетчика, передача данных с переключателей в переменную **m_PenStyle** автоматически выполняется при первом создании диалогового окна и далее может осуществляться при помощи вызова функции **UpdateData(TRUE)**. Поскольку это уже делалось для переменной **m_PenWidth**, обновление переменной **m_PenStyle** не составит большого труда.

Хорошо, некоторую работу все же придется выполнить, а именно: изменить две строки кода. В конструктор диалогового окна добавьте строку

```
m_PenStyle = PS_INSIDEFRAME;
```

Кроме того, измените строку в функции **OnLButtonDown()**, создающую перо, таким образом, чтобы она выглядела как

```
m_Pen.CreatePen(m_PenStyle, m_PenWidth, m_PenColor);
```

Завершив корректировки, откомпилируйте и запустите новую версию программы PaintORama, в которой теперь реализованы все функции, обеспечивающие выбор необходимого пера. Обратите внимание, что сейчас можно создавать стилизованное перо любого цвета, однако ширина пера должна быть выбрана равной 1. При попытке создания стилизованного пера с шириной, отличной от 1, получается старое доброе сплошное перо.

На этом этапе вы не заметите никакого различия между сплошным стилем пера и стилем **IDC_INSIDE_FRAME** — это различие становится очевидным только, когда рисуются формы и фигуры. Которое, как ни странно, является темой следующего раздела. Невероятное стечение обстоятельств!

Приложение PaintORama: линии и формы

Как вы уже видели в нескольких последних главах, рисовать линии и фигуры совсем не трудно — это можно сделать с помощью всего лишь нескольких функций. Поскольку приложение PaintORama уже способно выполнять рисунки от руки, то все, что осталось — это соответствующие элементы управления, позволяющие переключаться с режима рисования от руки на режим вычерчивания линий и прямоугольников.

Конечно, можно воспользоваться переключателями, но мы сейчас попытаемся применить что-нибудь новое, например, поле со списком (combo box). Как и раньше, сначала будет построен интерфейс, а затем необходимые коды, обеспечивающие корректное функционирование.

Маленькое симпатичное поле со списком

Позаботьтесь о том, чтобы основная диалоговая форма проекта PaintORama была открыта в Dialog Editor. Когда все будет готово, выполните следующие шаги, чтобы построить интерфейс для линий и форм:

1. Перетащите групповой блок из панели Control на форму. Поместите ее снизу группы Pen Styles и придайте ей ширину в 40 единиц и высоту 128 единиц, чтобы она соответствовала по ширине группе Pen Styles (возможно, придется отключить сетку).
2. Замените заголовок на "&Lines and Shapes". (Как и в случае группы Pen Styles, амперсанд заставляет Windows подчеркнуть в заголовке букву "L"). Для переключения на элементы управления, содержащиеся в группе, можно нажать комбинацию Alt+L.) Идентификатор группы, имеющий значение **IDC_STATIC**, можно оставить неизменным, но обязательно следует щелкнуть на флажке Group в диалоговом окне Group Box Properties. Это приводит к тому, что группа действует в качестве терминатора, или указателя конца, для помещенных в нее переключателей.
3. Выберите элемент управления полем со списком и перетащите его из панели Control в группу. На рис. 10.11 элемент управления полем со списком выделен.
4. Установите размеры и местоположение поля со списком в пределах группы. Этот процесс двухступенчатый: растяните элемент управления с таким расчетом, чтобы его ширина была равна ширине группы (оставив видимым обрамление), затем поместите курсор мыши на кнопке поля со списком и щелкните. Появится новый прямоугольник. Воспользуйтесь этим прямоугольником для задания размеров открытого поля со списком. Если этого не сделать, щелчок на поле со списком не приведет к его открытию. На рис. 10.12 показано, как устанавливаются размеры активной области поля со списком.
5. После выбора подходящих размеров поля со списком откройте диалоговое окно Combo Box Properties, щелкнув правой клавишей на поле со списком и выбрав элемент Properties из контекстного меню. В закладке General измените имя элемента управления на **IDC_SHAPES** и убедитесь в том, что флажки Tab Stop и Visible отмечены. Все другие флажки должны быть неотмечены.

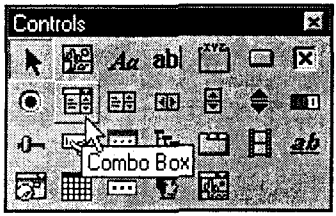


РИСУНОК 10.11. *Выбор элемента управления поля со списком в панели Control.*

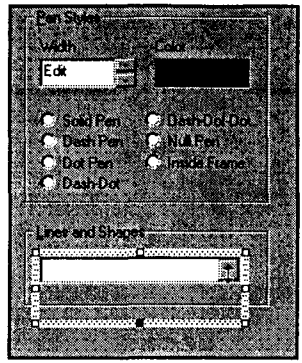


РИСУНОК 10.12. *Установка размеров активного прямоугольника поля со списком.*

6. Выбрав закладку Data (Данные), введите следующие строки: "Freehand" (Рисунок от руки), "Lines" (Линии), "Ovals" (Овалы) и "Rectangles" (Прямоугольники). В конце каждой строки нажимайте Ctrl+Enter. (Нажатие только Enter приводит к закрытию диалогового окна.) Диалоговое окно должно выглядеть подобно показанному на рис. 10.13.

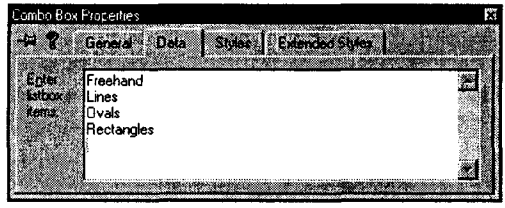


РИСУНОК 10.13. *Запись данных в поле со списком*

7. Выберите закладку Styles и установите Type на Drop List (Выпадающий список). (Другие стили поля со списком позволяют, помимо выбора из списка, вводить новые значения, но в нашем случае новые значения не допускаются.) Установите значение No в Owner Draw. Флажок Vertical Scroll должен быть отмечен, а все другие флажки — сброшены.
8. Для включения нового элемента данных, соответствующего полю со списком, воспользуйтесь ClassWizard. Присвойте элементу данных имя **m_ShapesCombo**, установите значение Control для Category и значение **CComboBox** для Type.
9. В метод **OnInitDialog()** добавьте следующие строки, выполняющие инициализацию:

```
// 5. Инициализация переменной m_ShapesCombo
m_ShapesCombo.SetCurSel(0);
```

Поскольку **m_ShapesCombo** — это управляющая переменная, ее нельзя инициализировать в конструкторе — она пока еще не существует. Переменную следует инициализировать в рамках метода **OnInitDialog()**. Метод **CComboBox SetCurSel()** устанавливает текущий выбор в поле со списком, при этом первому выбранному элементу присваивается 0, второму элементу — 1 и т.д. Если этого не сделать, поле со списком останется пустым, пока на нем не будет выполнен щелчок.

Линии и формы

Вычерчивание линий несколько отличается от ручных рисунков. Во время рисования от руки каждый раз, когда перемещается мышь, программа PaintORama

рисует небольшой сегмент линии, соединяя текущее положение мыши с позицией, зафиксированной в переменой **m_LineStart**. Затем программа обновляет значение **m_LineStart** таким образом, что она в процессе рисования "следует" за мышью по экрану.

В случае линий и форм требуется нарисовать форму в окончательном виде, только когда клавиша мыши отпущена. Следовательно, обновление значения переменной **m_LineStart** по мере продвижения мыши выполняться не будет; вместо этого придется ввести новую переменную **m_LineEnd**, а затем реализовать общую функцию **DrawShapes()**, которая будет рисовать фигуру между любыми двумя точками.

Выполните следующие шаги:

1. Добавьте новую приватную переменную типа **CPoint** с именем **m_LineEnd** в определение класса **CPaintORamaDlg**. Для этих целей можно воспользоваться диалоговым окном Add Member Variable либо добавить эту переменную вручную.
2. В функции **OnLButtonDown()** найдите строку, которая инициализирует переменную **m_LineStart**. Под ней вставьте следующую строку, выполняющую инициализацию переменной **m_LineEnd**:

```
m_LineEnd = point;      // Это новая конечная точка
```

3. Удалите вызов функции **SetCapture()**.
4. Измените код функции **OnLButtonUp()** таким образом, чтобы она устанавливала переменную **m_LineEnd**, а затем вызывала новую функцию **DrawShape()**. Удалите обращение к функции **ReleaseCapture()**. Модифицированная функция **OnLButtonUp()** показана в листинге 10.4.

Листинг 10.4. Модифицированный вариант функции OnLButtonUp().

```
void CPaintORamaDlg::OnLButtonUp(UINT nFlags, CPoint point)
{
      // Установить значение режима рисования в false
      m_IsDrawing = false;

      // Обновить конец линии
      m_LineEnd = point;

      // Нарисовать соответствующую линию или форму
      DrawShape();
}
```

5. Добавьте метод **DrawShape()** в класс **CPaintORamaDlg**, для чего щелкните правой кнопкой мыши на имени класса в окне ClassView, а затем выберите Add Member Function из контекстного меню. Возвращаемый методом тип должен быть **void**, а объявление функции должно иметь вид

```
DrawShape(bool stretch = false);
```

6. Добавьте код, показанный в листинге 10.5, к коду, сгенерированному ClassWizard. В следующем разделе мы рассмотрим, как работает полученный программный код.

Листинг 10.5. Метод DrawShape().

```
void CPaintORamaDlg::DrawShape(bool stretch)
{
      CClientDC dc(this);
      dc.IntersectClipRect(m_Canvas);
```

```
    // Определить текущий режим рисования
    int drawMode = m_ShapesCombo.GetCurSel();
    // Подготовить контекст устройства
    dc.SelectObject(&m_Pen);
    if (stretch && drawMode != 0)
    {
        dc.SetROP2(R2_NOT);
    }

    // Нарисовать соответствующую форму
    switch(drawMode)
    {
    case 0 // Рисунок от руки
           // Переместиться к началу линии
           dc.MoveTo(m_LineStart);
           // Рисовать до текущей позиции
           dc.LineTo(m_LineEnd);
           // Обновить текущее положение мыши
           m_LineStart = m_LineEnd;
           break;
    case 1 // Линии
           dc.MoveTo(m_LineStart);
           dc.LineTo(m_LineEnd);
           break;
    case 2: // Овалы
           dc.Ellipse(CRect(m_LineStart, m_LineEnd));
           break;
    case 3: // Прямоугольники
           dc.Rectangle(CRect(m_LineStart, m_LineEnd));
           break;
    }
}
```

7. Измените функцию **OnMouseMove()** в соответствии с листингом 10.6.

Листинг 10.6. Модифицированная версия функции OnMouseMove().

```
void CPaintORamaDlg::OnMouseMove(UINT nFlags, CPoint point)
{
    // Рисовать, если m_IsDrawing истинно, мышь находится на
    // холсте и нажата левая кнопка мыши
    if (m_IsDrawing && (nFlags & MK_LBUTTON) &&
m_Canvas.PtInRect(point))
    {
        DrawShape(true); // Удалить первую линию
        m_LineEnd = point;
        m_LineShape(true); // Обновить рисунок
    }
}
```

8. Откомпилируйте и запустите программу. При переключении на режим рисования прямоугольников или эллипсов программа PaintORama обеспечивает визуальную обратную связь при помощи "растягиваемой" формы, которая не принимает окончательный вид до тех пор пока не будет отпущена кнопка мыши. В следующем разделе будет показано, как работает *метод резиновой*

нити (rubber-banding). На рис. 10.14 показана программа PaintORama с новыми возможностями.

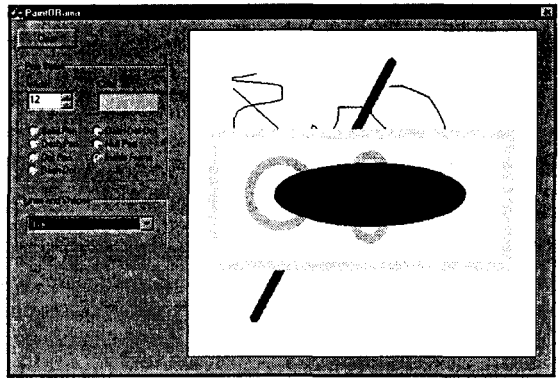


РИСУНОК 10.14. *Использование метода резиновой нити для рисования эллипсов.*

Секреты метода резиновой нити

Функция **DrawShape()** несколько длиннее функций, рассмотренных ранее, однако после знакомства с общей структурой функции, ее работа становится совершенно понятной. Функция **DrawShape()** работает во взаимодействии с функциями **OnMouseMove()** и **OnLButtonUp()** для реализации эффекта резиновой нити.

Рассмотрим работу функцию **DrawShape()** шаг за шагом:

- Две первых строки просто устанавливают контекст устройства. Эти строки совпадают с теми, которые ранее использовались в **OnMouseMove()**.
- На следующем шаге (третья строка кода) определяется текущее состояние поля со списком с использованием функции **GetCurSel()**. **GetCurSel()** возвращает номер выбранного элемента в поле со списком, начиная с 0. Это значение потребуется для принятия решения о том, какой тип формы рисовать, поэтому его следует сохранить в локальной переменной **drawMode**.
- На третьем шаге устанавливается контекст устройства. Здесь же выбирается текущее перо, как это делалось ранее. Затем, если вы не рисуете от руки, а работаете в "растягиваемом" режиме, при помощи функции **SetROP2()** следует установить режим рисования в **R2_NOT**, для которого характерно обращение каждого пиксела на экране. (Чуть позже станет понятно, почему это так важно.) Обратите внимание, что в случае режима рисования от руки вызов функции **SetROP2()** не выполняется.
- На четвертом, заключительном этапе выполняется оператор **switch**, примененный к значению **drawMode** (значение, полученное из поля со списком). В каждом случае рисуется конкретная фигура. В первом случае (0) используется программный код, реализующий рисование от руки, который берется из исходной функции **OnMouseMove()**. Во втором случае (1) рисование линии выполняется с помощью функций **MoveTo()** и **LineTo()**. Третий и четвертый случаи еще проще; вычерчивание овалов и прямоугольников выполняется с помощью функций **Ellipse()** и **Rectangle()**.

Для понимания метода резиновой нити при рассмотрении функции **OnMouseMove()** держите в уме код **DrawShape()**. В тексте **OnMouseMove()** первый оператор (при условии, что мышь попала в область холста) вызывает **DrawShape(true)**. Когда поле со списком настраивается на рисование линии или формы, передача значения **true** означает переход в режим рисования **R2_NOT** (обращение). В режиме обращения повторное рисование линии или формы приводит к ее стиранию — подобный прием делает возможной реализацию метода резиновой нити.

Обратите внимание на то обстоятельство, что когда функция **OnMouseMove()** вызывается впервые, переменные **m_LineStart** и **m_LineEnd** содержат одно и то же значение, так что ничего не рисуется. Тем не менее, после вызова **DrawShape(true)** функция **OnMouseMove()** обновляет значение **m_LineEnd**, устанавливая его в новую позицию пера, после чего еще раз вызывает **DrawShape(true)**. На этот раз переменные **m_LineEnd** и **m_LineStart** имеют разные значения, так что на экране отрисовывается "обращенная" линия или форма.

При следующем обращении к функции **OnMouseMove()** она снова вызывает **DrawShape (true)**. Поскольку с момента последнего обращения к функции **DrawShape()** значение переменной **m_LineEnd** не изменялось, в результате текущего вызова рисуется новый прямоугольник или овал непосредственно поверх предыдущего. В качестве режима рисования установлен **R2_NOT**, поэтому цвет каждого пиксела восстанавливается таким, каким он был до рисования первого прямоугольника, тем самым затирая фигуру. Далее переменная **m_LineEnd** еще раз обновляется, и весь процесс повторяется.

После отпускания кнопки мыши, в методе **OnLButtonUp()** функция **DrawShape()** вызывается в последний раз без аргументов. Поскольку по умолчанию принимается аргумент **false**, используются обычное перо и режим рисования. Форма или линия рисуются в окончательном виде и с правильно выбранными цветом и толщиной; на этот раз она не затирается.

Приложение PaintORama: возможность выбора кисти

Поскольку опыт включения в программу элементов управления стилями пера уже имеется, совершенно несложно добавить и управление стилем кисти. Стиль кисти выбирается из списка, а цвета — при помощи образца цвета (как и при выборе пера).

Памятуя о наличии соответствующих навыков, дальнейшие наши инструкции будут весьма краткими.

Формирование списка

Перетащите из панели Control и поместите ниже группы Lines And Shapes следующие элементы управления:

- *Группу.* Установите ее заголовок в "&Brush Styles". Оставьте значение идентификатора по умолчанию, т.е. **IDC_STATIC**.
- *Список.* Момент выбора списка показан на рис. 10.15. Его свойства будут устанавливаться позже.
- *Два изображения.* Измените их Type на Rectangle, а Color — на Black. Присвойте первому изображению идентификатор **IDC_BRUSHCOLOR**, а второму — идентификатор **IDC_BRUSHPREVIEW**. Назначьте **IDC_BRUSHCOLOR** стиль Notify.
- *Три статических текста.* Оставьте их имена такими, какими они приняты по умолчанию, т.е. **IDC_STATIC**. Измените их заголовки на "Styles", "Color" и "Preview" (предварительный просмотр). Установите размеры и расположение элементов управления в соответствие с рис. 10.16. Элемент управления в верхнем правом углу — это **IDC_BRUSHCOLOR**, а изображение в нижнем правом углу — **IDC_BRUSHPREVIEW**.

Установка свойств списка

Теперь выберем свойства элемента управления списком. В закладке General измените идентификатор списка на **IDC_BRUSHSTYLE**. Позаботьтесь о том, чтобы флажки Tab Stop и Visible были отмечены, а все другие оставались сброшенными.

В закладке Styles выберите Single из выпадающего списка Selection и No — из списка Owner Draw. Активизируйте флажки Border, Notify, Vertical Scroll и No Integral Height (Не учитывать общую высоту), а все другие сбросьте. Результат установок показан на рис. 10.17.

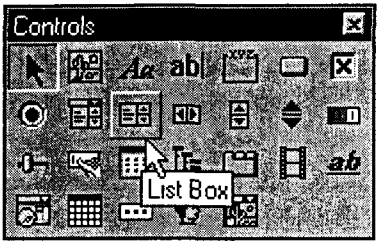


РИСУНОК 10.15. *Элемент управления списком в панели Control.*

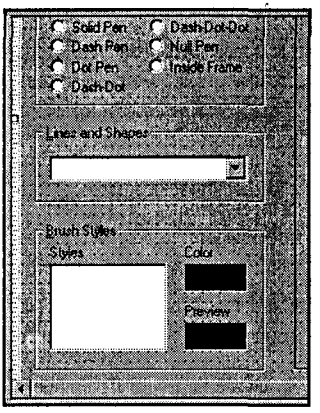


РИСУНОК 10.16. *Расположение элементов управления стилем кисти.*

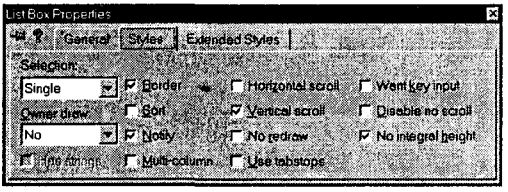


РИСУНОК 10.17. *Свойства флажка списка IDC_BRUSHSTYLE.*

Инициализация элемента управления списком

Несмотря на то что Visual C++ может автоматически заполнять содержимое поля со списком, подобное не может выполняться для простого списка — это потребуется сделать с помощью функции **CListBox::AddString()**. Для инициализации элемента управления списком необходимо выполнить два следующих шага:

1. При помощи ClassWizard добавьте новую переменную для списка **IDC_BRUSHSTYLE**. Присвойте этой переменной имя **m_BrushStyleList** и установите для нее Category в Control, а Type — в **CListBox**.
2. Включите программный код из листинга 10.7 в функцию **OnInitDialog()**. Этот код добавляет в элемент управления списком описание всех возможных стилей кисти. Последняя строка создает девятый элемент (Стандартная белая кисть в позиции 8, начиная с 0), назначаемый по умолчанию.

Листинг 10.7. Инициализация элемента управления списком IDC_BRUSHSTYLE.

```
// 6. Инициализация списка m_BrushStyleList
m_BrushStyleList.AddString("(none)");
m_BrushStyleList.AddString("Solid");
m_BrushStyleList.AddString("LL-UR Diagonal");
m_BrushStyleList.AddString("UL-LR Diagonal");
m_BrushStyleList.AddString("Grid");
m_BrushStyleList.AddString("Grid Diagonal");
m_BrushStyleList.AddString("Horizontal");
m_BrushStyleList.AddString("Vertical");
m_BrushStyleList.AddString("White");
m_BrushStyleList.AddString("Light Gray");
m_BrushStyleList.AddString("Medium Gray");
```

```
m_BrushStyleList.AddString("Dark Gray");
m_BrushStyleList.AddString("Black");
m_BrushStyleList.SetCurSel(8);
```

Добавление переменных для работы с кистью

Для работы с кистями потребуется тот же набор переменных, который создавался ранее для перьев. Переменные вместе с их типами и описаниями перечислены в табл. 10.2. Их можно добавить вручную или использовать для этих целей диалоговое окно Add Member Variable.

Таблица 10.2. Переменные для работы с кистью

Имя переменной	*Тип*	*Описание*
m_Brush	CBrush	Заполняет требуемые фигуры
m_BrushColor	COLORREF	Создает **m_Brush**
m_BrushStyle	int	Проводит различие среди штриховых кистей
m_BrushColorSwatch	CRect	Сохраняет координаты образца цвета кисти
m_BrushPreviewSwatch	CRect	Сохраняет координаты образца предварительного просмотра кисти

Инициализация переменных для работы с кистью

Четыре переменных для работы с кистью — **m_BrushColorSwatch**, **m_BrushPreviewSwatch**, **m_BrushColor** и **m_Brush** — можно проинициализировать в функции **OnInitDialog()**. В листинге 10.8 показан код, который потребуется включить в программу. Оставшаяся переменная, **m_BrushStyle**, будет инициализироваться во время внесения изменений в список.

Листинг 10.8. Инициализация переменных для работы с кистью.

```
// 7. Вычислить точное местоположение индикатора цвета кисти
CWnd* pBrushColor = GetDlgItem(IDC_BRUSHCOLOR);
pBrushColor->GetWindowRect(&m_BrushColorSwatch);
ScreenToClient(&m_BrushColorSwatch);
m_BrushColorSwatch.DeflateRect(2, 2, 1, 1);
// 8. Вычислить точное местоположение образца предварительного
// просмотра кисти
CWnd* pPreviewColor = GetDlgItem(IDC_BRUSHPREVIEW);
pPreviewColor->GetWindowRect(&m_BrushPreviewSwatch);
ScreenToClient(&m_BrushPreviewSwatch);
m_BrushPreviewSwatch.DeflateRect(2, 2, 1, 1);

// 9. Проинициализировать переменную m_BrushColor
m_BrushColor = RGB(255, 255, 255);

// 10. Использовать шаблон белой кисти в качестве значения по
// умолчанию
m_Brush.CreateStockObject(WHITE_BRUSH);
```

Код в листинге 10.8 вычисляет размеры двух образцов цветов. Это тот же самый код, который использовался для образца цвета пера, с измененными именами переменных. Значение переменной **m_BrushColor**, **(RGB(255, 255, 255))**, установлено в белый цвет, однако **m_Brush** инициализируется с использованием стандар-

тной белой кисти, но не путем создания сплошного пера с использованием переменной **m_BrushColor**.

Добавление в программу возможностей выбора кисти

Код, обеспечивающий использование кистей, оказывается проще кода, необходимого для работы с пером. Все что нужно сделать — это включить единственную строку

```
dc.SelectObject(&m_Brush);
```

в функцию **DrawShape()** сразу же после строки, которая выбирает **m_Pen** в контексте устройства.

Хотя рисование кистью проще, чем рисование пером, все же создание кисти требует чуть больше усилий. Тем не менее, соответствующий программный код не особенно сложный — он просто длинный. В двух словах, базовая идея такова:

1. Удалить старую кисть. Для этого служит строка:

```
m_Brush.DeleteObject();
```

2. Выяснить, какой стиль кисти выбран на текущий момент в списке **m_BrushStyleList**, воспользовавшись для этой цели функцией **GetCurSel()**. Сохранить эту переменную в локальной переменной типа **int** с именем **style**.
3. Записать большой оператор **switch**, в котором будут создаваться различные кисти для каждого возможного значения переменной **style**.
4. Вызвать функцию **PaintBrushPreview()** для отображения нового стиля кисти в переменной **m_BrushPreviewSwatch**.

Обработка сообщения LBN_SELCHANGE

Этот процесс выглядит достаточно простым, каковым он, по сути дела, и является. Вопрос, однако, заключается в том, *куда поместить* код? Когда осуществляется другой выбор из списка, последний генерирует сообщение **LBN_SELCHANGE**. Если дважды щелкнуть на вашем списке в окне Dialog Editor, ClassWizard предложит добавить в программу новый метод с именем **OnSelchangeBrushstyle()**, обеспечивающий обработку упомянутого сообщения.

Завершенный код функции показан в листинге 10.9.

Листинг 10.9. Метод OnSelchangeBrushstyle().

```
void CPaintORamaDlg::OnSelchangeBrushstyle()
{
      m_Brush.DeleteObject();
      int style = m_BrushStyleList.GetCurSel();
      switch(style)
      {
      case 0:
            m_Brush.CreateStockObject(NULL_BRUSH);
            break;
      case 1:
            m_Brush.CreateSolidBrush(m_BrushColor);
            break;
      case 2:
```

```
            m_Brush.CreateHatchBrush(HS_BDIAGONAL, m_BrushColor);
            break;
      case 3:
            m_Brush.CreateHatchBrush(HS_FDIAGONAL, m_BrushColor);
            break;
      case 4:
            m_Brush.CreateHatchBrush(HS_CROSS, m_BrushColor);
            break;
      case 5:
            m_Brush.CreateHatchBrush(HS_DIAGCROSS, m_BrushColor);
            break;
      case 6:
            m_Brush.CreateHatchBrush(HS_HORIZONTAL, m_BrushColor);
            break;
      case 7:
            m_Brush.CreateHatchBrush(HS_VERTICAL, m_BrushColor);
            break;
      case 9:
            m_Brush.CreateStockObject(LTGRAY_BRUSH);
            break;
      case 10:
            m_Brush.CreateStockObject(GRAY_BRUSH);
            break;
      case 11:
            m_Brush.CreateStockObject(DKGRAY_BRUSH);
            break;
      case 12:
            m_Brush.CreateStockObject(BLACK_BRUSH);
            break;
      default:
            m_Brush.CreateStockObject(WHITE_BRUSH);
            break;
      }
      PaintBrushPreview();
}
```

Добавление поддержки предварительного просмотра кисти и выбора цвета

Для изменения цвета кисти нужно делать то же, что имело место для случая пера. Дважды щелкните на элементе управления **IDC_BRUSHCOLOR** (не забудьте, что флажок Notify должен быть отмечен) и примите в качестве имени функции **OnBrushcolor()**. Включите в полученную функцию код из листинга 10.10.

Листинг 10.10. Изменение цвета кисти в функции OnBrushcolor().

```
void CPaintORamaDlg::OnBrushcolor()
{
      CColorDialog dlg(m_BrushColor);
      if (dlg.DoModal() == IDOK)
      {
            m_BrushColor = dlg.GetColor();
            CClientDC dc(this);
```

```
            CBrush b(m_BrushColor);
            dc.FillRect(&m_BrushColorSwatch, &b);
        }
        OnSelchangeBrushstyle();
}
```

Обратите внимание на то обстоятельство, что этот программный код практически идентичен коду, используемому в методе **OnPencolor()** в начале главы. Единственное более-менее существенное различие состоит в том, что каждый раз, когда меняется цвет кисти, необходимо создавать новую кисть — таким образом, в конце кода вызывается функция **OnSelchangeBrushstyle()**. Новая кисть создается всякий раз, когда рисуется линия, так что никаких затруднений возникать не должно.

Последняя функция, которая нам понадобится, прежде чем можно будет перейти к тестированию приложения PaintORama — это **PaintBrushPreview()**. Эта функция не есть обработчик сообщений Windows, подобно многим функциям, разработанным в главе. Добавить ее в программу можно через команду Add Member Function в контекстном меню класса. (Если вы включаете эту функцию вручную, не забудьте объявить ее в определении класса (файл заголовков) и поместить ее код в файл реализации (CPP).)

Функция **PaintBrushPreview()** состоит всего лишь из пяти строк:

```
void CPaintORamaDlg::PaintBrushPreview()
{
      CClientDC dc(this);
      dc.FillRect(&m_BrushPreviewSwatch, &m_Brush);
}
```

Откомпилируйте и запустите программу на выполнение. Исправьте возможные ошибки. Рисунок 10.18 иллюстрирует работу программы PaintORama.

Вскоре ожидается: только в театрах

Следует признать, что это была весьма длинная глава с изобилием подробностей. К сожалению, в программе PaintORama все еще остаются некоторые ограничения, устранение которых откладывается на последующие главы. Поскольку вы, упорно изучая материал, сумели-таки достичь этого места, предлагаем краткий обзор следующей главы. Мы не обещаем раскрыть все тайны вселенной, тем не менее, познакомим с решением одной надоедливой проблемы, с которой приходится сталкиваться даже опытным программистам, работающим в среде Windows.

Вот в чем заключается эта проблема: в случае рисования вне функции **OnPaint()**, каким образом можно восстановить полученный шедевр на экране, если "паразитное" окно непреднамеренно разрушит его? Для интерактивной программы, коей является PaintORama, перенос кода рисования в **OnPaint()**, по всей вероятности, чреват проблемами.

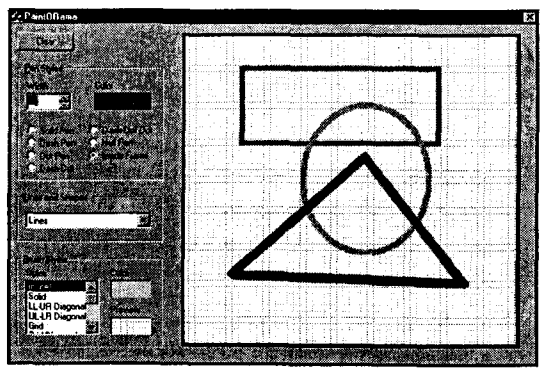


РИСУНОК 10.18. *Выполнение программы PaintORama.*

Решение проблемы достаточно просто и предусматривает использование архитектуры "документ-представление". В упомянутой архитектуре обработка данных и их представление разделены.

Например, рассмотрим приложение PaintORama. Каждая линия и форма вместе с ее характеристиками может быть порцией данных — именно так работают программы векторной графики, подобные Adobe Illustrator и CorelDRAW! С другой стороны, документ PaintORama может оказаться просто структурой данных, которая обеспечивает возможность восстановления растрового изображения в любое время — это несколько напоминает работу некоторых программ рисования, таких как Windows Paint или Adobe Photoshop.

В следующей главе мы продолжим дальнейшее совершенствование приложения PaintORama, ознакомив вас с быстрым и эффективным решением проблемы восстановления рисунка: класс **CMetaFileDC**. На рассмотрении класса **CMetaFileDC** исследование приложений на основе диалоговых окон завершается — больше мы не будем обращаться к этой теме, пора двигаться дальше. Класс **CDialog** и редактор Dialog Editor еще раз встретятся при изучении элементов управления ActiveX и класса **FormView**.

Попрощаемся с Dialog Editor до лучших времен. Завтра наступит новый день, и вы захотите немного отдохнуть, прежде чем приступить к исследованию архитектуры "документ-представление" и внедрению ее в приложении PaintORama.

Глава 11

Построение документов и представлений

Не так давно у самых первых компьютеров не было программ в том виде, в каком мы их знаем сегодня. Каждый раз программисты должны были выполнять монтаж аппаратных средств таким образом, чтобы этот мастодонт смог решать новые задачи.

Изобретение сохраняемой в памяти программы, в конечном итоге позволившее создать программируемый компьютер, стало первым шагом на пути в нирвану программного обеспечения, которой мы наслаждаемся сегодня. Ладно, *нирвана* — это слишком сильно сказано. Однако согласитесь, что с тех пор программирование ушло далеко вперед.

Например, большинство современных программ значительно лучше *структурированы*. Ранние программы представляли собой огромные массивы монолитного, прямолинейного кода без процедур и локальных переменных. Программисты убедились в том, что с такими программами очень трудно работать. Например, необходимость поменять переменную в строке 25435 требует от программиста тщательной проверки остальной части программы в поисках возможных побочных эффектов, которые могут возникнуть хотя бы из-за наличия в программе безусловного перехода из отдаленной области программы в зону, подвергающуюся модификации.

После того как были вскрыты проблемы, связанные с ничем не ограничиваемыми безусловными переходами (операторы **goto**) и использованием глобальных переменных, были выработаны неформальные правила создания удобных в эксплуатации программ. Некоторые из этих правил сохранились в новых языках программирования, например, принцип области видимости, реализованный в языке Pascal; другие так и остались эмпирическими правилами, нащупанными вслепую на основе удачного опыта разработки программ. Наиболее интересными являются правила, касающиеся структуры программы; они помогают программистам принимать решения, сколько процедур включать в программы, насколько большими могут быть эти процедуры и как эти процедуры должны взаимодействовать друг с другом.

У современного объектно-ориентированного программирования имеется свой собственный набор правил. Когда программисты приступают к созданию классов и объектов, старые правила, появившиеся во времена процедурных кодов, по всей вероятности, использоваться не могут. Таким образом, начался поиск новых путей компоновки фрагментов, составляющих программы. Программисты, работающие на языке Smalltalk (один из первых объектно-ориентированных языков программирования), разработали широко используемую архитектуру MVC (*Model-View-Controller* — Модель-Представление-Контроллер).

Так же как и прежние поколения программистов научились подразделять свои программы на самодостаточные процедуры, разработчики архитектуры MVC провозгласили, что современные программисты должны разбивать программы на три специальных вида объектов:

- *Объекты-модели* (Model objects) — Предназначены для хранения компьютерного представления счетов, заказчиков и каких-либо других реально существующих субъектов окружающего мира, с которыми имеет дело программа.
- *Объекты-представления* (View objects) — Предназначены для отображения информации.
- *Объекты-контроллеры* (Controller objects) — Реагируют на ввод пользователя.

Библиотека MFC поддерживает архитектуру MVC за счет реализации архитектуры типа *"документ-представление"* (Document-View architecture), обычно известной под именем *DocView*. Архитектура DocView обрабатывает документы: ее

модельная часть сохраняет документы, а представленческая часть отображает их. Если вас интересует, что произошло с контроллерной частью MVC, то нет никаких оснований для беспокойства — контроллер образуют компоненты Windows и сама Windows. Нет необходимости создавать класс контроллера для выборки входных и очередизованных сообщений с целью доставки их в документ или представление — Windows выполняет эти функции автоматически.

Вместо того чтобы приступить к работе с приложением архитектуры DocView, вновь обратимся к программе PaintORama и посмотрим, как можно использовать предшественника DocView из Windows — класс **CMetaFileDC** — для записи состояния каждого шедевра, создаваемого в приложении PaintORama.

Приложение PaintORama: еще раз о сообщении WM_PAINT

Документная часть DocView-программы сохраняет пользовательские данные. Например, в программе обработки текстов документальная часть запоминает все символы текста и любые знаки форматирования, которые он содержит. В программах электронных таблиц документ хранит формулы и данные, введенные пользователем. В программе PaintORama документ хранит цвет, форму, стиль и толщину пера, а также кисть, используемую для вычерчивания каждой линии, прямоугольника и эллипса.

В программе PaintORama документная часть отсутствует. Следовательно, если возникает необходимость воспроизвести рисунок на холсте после переключения на другое приложение, вся предыдущая работа пользователя будет потеряна. Попробуйте сами. Нарисуйте несколько линий, используя для этого текущий вариант программы PaintORama. Перетащите окно вниз, чтобы был виден только его заголовок, затем верните его в прежнее положение. Вот вам и сюрприз — рисунок исчез. Для устранения такого недостатка предоставим пользователю возможность запомнить рисунок.

Можно начать с создания набора классов для моделирования каждой рисованной формы, после чего предусмотреть коллекцию, в которой будет храниться каждый класс. Однако Windows предлагает намного более быстрый и простой подход: использование класса **CMetaFileDC**, предназначенного для создания и хранения метафайла. *Метафайл* — это просто структура данных, содержащая команды интерфейса GDI, такие как **LineTo()** или **Rectangle()**. Воспользовавшись классом **CMetaFileDC**, можно автоматически записывать линии и формы такими, какими их нарисовал пользователь. Если образ требуется нарисовать заново, программе остается лишь воспроизвести этот метафайл в методе **OnPaint()**. Итак, возьмем программу PaintORama и наделим ее этим свойством, тем самым подарив миру новую версию приложения PaintORama.

Обзор программы PaintORama

Для включения в ваше приложение метафайла требуется выполнить всего лишь три шага. (Разумеется, собственно реализация кода, как всегда, потянет за собой множество деталей.) Вот краткое описание этих шагов:

- В программу необходимо добавить новый указатель на **CMetaFileDC** в виде элемента данных класса. Эта переменная указывает на объект **CMetaFileDC**,

созданный в куче. Потребуется также записать код, распределяющий и освобождающий память под объект **CMetaFileDC**.

- Необходимо записать в метафайл каждую операцию GDI.
- В рамках функции **OnPaint()** воспользоваться метафайлом для воспроизведения линий и форм, составляющих рисунок. После отображения рисунка следует выполнить повторную инициализацию метафайла, так чтобы он продолжал записывать каждую деталь рисунка.

Проанализируем каждый из этих шагов. Изменения можно вносить непосредственно в приложение PaintORama. Или, что еще лучше, можно скопировать весь проект в новую папку, чтобы случайно не исказить более ранней версии. На сопровождающем CD-ROM новая версия программы PaintORama находится в отдельном каталоге.

Создание объекта CMetaFileDC

Начнем с того, что откроем приложение PaintORama и выберем панель ClassView. Раскройте класс **CPaintORamaDlg**, после чего выполните следующие шаги:

1. Откройте диалоговое окно Add Member Variables, щелкнув правой кнопкой мыши на классе **CPaintORamaDlg** и выбрав из контекстного меню пункт Add Member Variable. В этом диалоговом окне добавьте приватный указатель на **CMetaFileDC** (тип данных **CMetaFileDC***). Присвойте указателю имя **m_pMF**, как показано на рис. 11.1.
2. Воспользуйтесь окном ClassView, чтобы найти и открыть функцию **OnInitDialog()**. В конце функции (до возврата значения **TRUE**) добавьте код, выполняющий инициализацию переменной **m_pMF** и создающий в куче новый объект **CMetaFileDC**.

```
// 11. Добавить метафайл
m_pMF = new CMetaFileDC;
m_pMF->Create();
```

3. Щелкните правой кнопкой мыши на классе **CPaintORamaDlg** в окне ClassView еще раз и выберите из контекстного меню Add Windows Message Handler. В появившемся диалоговом окне выберите сообщение **WM_DESTROY**, как показано на рис. 11.2. Щелкните на Add And Edit и в окне редактирования добавьте код, выделенный в листинге 11.1.

Листинг 11.1. Метод CPaintORamaDlg::OnDestroy().

```
void CPaintORamaDlg::OnDestroy()
{
      CDialog::OnDestroy();
      // ЧТО СДЕЛАТЬ: Поместить здесь код обработчика сообщений
      m_pMF->Close();
      delete m_pMF;
}
```

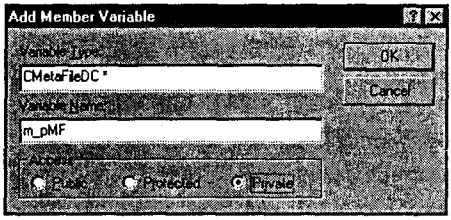


РИСУНОК 11.1. *Добавление указателя m_pMF*

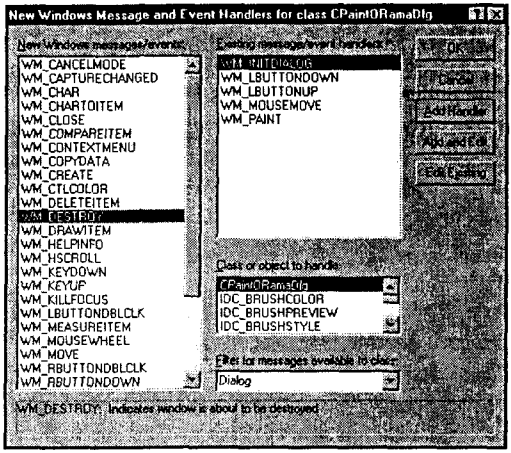


РИСУНОК 11.2. *Создание обработчика сообщений WM_DESTROY*

Нетрудно убедиться в том, что код инфраструктуры, необходимый для объекта **CMetaFileDC**, достаточно прост. Для инициализации метафайла используются **new** и **Create()**, а для освобождения используемой им памяти перед завершением программы — **Close()** и **delete**. На этом завершается шаг 1; на следующем шаге в метафайл записываются операции GDI.

Запись с помощью объекта CMetaFileDC

Базовый класс контекстов устройств **CDC** с каждым порожденным от **CDC** объектом (например, **CPaintDC** или **CClientDC**) ассоциирует два контекста устройств. Один контекст устройства, дескриптор которого хранится в **m_hDC**, известен как *контекст устройства вывода* (output device context). Этот контекст используют методы DC, генерирующие вывод — такие как **LineTo()** или **Rectangle()**. Второй DC, дескриптор которого хранится в **m_hAttribDC**, известен как *контекст атрибутов устройства* (attribute device context). Как следует из самого имени, его можно использовать для изменения атрибутов DC вывода, например, для изменения значения толщины пера.

Работая со стандартным DC, вы не должны следить за DC вывода и DC атрибутов, поскольку оба они характеризуют одно и то же устройство. У класса **CMetaFileDC** обычно нет DC атрибутов (его значением является **NULL**). В случае использования функций, которые изменяют DC атрибутов — такие как **SetBkMode()** или **SelectObject()** — класс **CMetaFileDC** эти вызовы игнорирует.

Такое встроенное поведение очевидным образом не подходит для приложения PaintORama. Например, хотелось бы, чтобы первоначально нарисованные толстые красные линии выглядели таковыми и после повторного воспроизведения изображения. К счастью, несмотря на то что класс **CMetaFileDC** не записывает установленные значения атрибутов автоматически, тем не менее он обеспечивает их легкую запись. Рассмотрим, как производится запись выходных операций и изменений атрибутов.

Запись контекста устройства вывода

Программа PaintORama генерирует весь вывод GDI в одной функции **DrawShape()**. На первый взгляд подключение функции записи в метафайл может показаться вполне простым делом, состоящим в сопоставлении с каждым обращением к DC экрана эквивалентного обращения к DC метафайла, например:

```
dc.MoveTo(m_LineStart);
dc.LineTo(m_LineEnd);
m_pMF->MoveTo(m_LineStart);
m_pMF->LineTo(m_LineEnd);
```

Правильность показанного кода не подлежит сомнению. Однако здесь не учитывается код метода резиновой нити, включаемый с целью обеспечения визуальной обратной связи. Необходимо, чтобы в метафайле фиксировались только окончательный вариант линии или фигуры, но отнюдь не промежуточные шаги. К счастью, можно без труда сказать, является форма фиксированной или всего лишь резиновой нитью. Если аргумент **stretch** функции принимает значение **true**, форма представляет собой резиновую нить и в метафайл записываться не должна. Соответственно, если аргумент **stretch** является **false** либо установлен режим рисования от руки (который не предусматривает режима резиновой нити), каждое обращение к DC вывода должно отображаться на вызов DC метафайла.

Необходимый программный код, содержащийся в операторе **switch DrawShape()**, представлен в листинге 11.2. Добавленные строки в листинге выделены.

Листинг 11.2. Использование класса CMetaFileDC в операторе switch DrawShape().

```
// Нарисовать соответствующую форму
      switch(drawMode)
      {
case  0: // Рисунок от руки
      dc.MoveTo(m_LineStart);    // Переместиться в начало линии
      dc.LineTo(m_LineEnd);      // Рисовать до текущей позиции
      m_pMF->MoveTo(m_LineStart);
      m_pMF->LineTo(m_LineEnd);
      m_LineStart = m_LineEnd; // Обновить текущую позицию мыши
      break;
case  1: // Линии
      dc.MoveTo(m_LineStart);
      dc.LineTo(m_LineEnd);
      if (! stretch)
      {
          m_pMF->MoveTo(m_LineStart);
          m_pMF->LineTo(m_LineEnd);
      }
      break;
case  2: // Овалы
      dc.Ellipse(CRect(m_LineStart, m_LineEnd));
      if (! stretch)
      {
          m_pMF->Ellipse(CRect(m_LineStart, m_LineEnd));
      }
      break;
case  3: // Прямоугольники
      dc.Rectangle(CRect(m_LineStart, m_LineEnd));
      if (! stretch)
```

```
        {
            m_pMF->Rectangle(CRect(m_LineStart, m_LineEnd));
        }
        break;
}
```

Рисование в DC возникает в нескольких различных местах. Например, чтобы раскрасить образец цвета пера или образец предварительного просмотра кисти, программа использует команды вывода GDI. Однако записывать их в метафайл не следует, поскольку они не оказывает воздействия на собственно документ (рисунок на холсте). Лишь в одном случае влияние на холст все же оказывается: функция **OnClearbtn()** стирает содержимое холста и раскрашивает его белым цветом. Обработка кнопки Clear будет реализована после учета команд изменения атрибутов DC.

Команды изменения атрибутов контекста устройства

Подобно дублированию команд вывода GDI, т.е. отображению их в **CMetaFileDC** и в DC экрана, придется продублировать команды, изменяющие значения атрибутов GDI. Однако нельзя просто записать вызов атрибутов GDI для объекта **CMetaFileDC**, поскольку по умолчанию поле **CMetaFileDC m_hAttribDC** установлено в **NULL**. Первоначально потребуется создать соответствующий DC.

Для создания DC атрибутов для класса **CMetaFile** выполните следующие действия:

1. Создайте неметафайловый DC. Этот DC будет ассоциироваться с объектом **CMetaFileDC**, который будет рассматривать его в качестве посредника при обращении к атрибутам.
2. Свяжите этот неметафайловый DC с метафайлом путем вызова функции **CMetaFile::SetAttribDC()**, передавая ей в качестве параметра неметафайловый DC.
3. Как и в случае команд вывода GDI, отобразите в метафайле каждую команду, вносящую изменения в окружение DC. (Если хотите узнать, почему это так важно, закомментируйте строки, которые отображают команды манипулирования атрибутами GDI в метафайл. Вскоре обнаружится, что рисунок сохранять можно, однако все линии затем воспроизводятся в виде черных линий толщиной в 1 пиксел.)

Для записи команд изменения DC атрибутов измените код функции **DrawShape()**, добавив строки, выделенные в листинге 11.3. Обратите внимание, что в листинге функция показана не полностью; не удаляйте оставшийся код **DrawShape()**.

Листинг 11.3. Изменение DC атрибутов объекта CMetaFileDC в функции DrawShape().

```
void CPaintORamaDlg::DrawShape(bool stretch)
{
   CClientDC dc(this);
   m_pMF->SetAttrib(dc);
   dc.IntersectClipRect(m_Canvas);
   m_pMF->IntersectClipRect(m_Canvas);

   // Определить выбранный режим рисования
   int drawMode = m_ShapesCombo.GetCurSel();
```

```
    // Подготовить DC
    dc.SelectObject(&m_Pen);
    dc.SelectObject(&m_Brush);
    m_pMF->dc.SelectObject(&m_Pen);
    m_pMF->dc.SelectObject(&m_Brush);

    // Остальная часть функции опущена
}
```

Воспроизведение метафайла

Теперь, когда программный код, обеспечивающий запись команд GDI, добавлен, можно записать код, обеспечивающий воспроизведение этих команд. Новый программный код содержится в методе **OnPaint()**. В программе хранителя экрана, разработанной в главе 8, вы удаляли программный код, сгенерированный AppWizard в **OnPaint()** — здесь можно проделать то же самое. На место удаленного кода необходимо поместить код, который воспроизводит метафайл и подготавливает его для приема дальнейших входных данных.

Потребуется выполнить такие действия:

1. Удалите весь код, содержащийся в методе **OnPaint()**. Создайте DC для холста; этот DC будет воспроизводить метафайл. Поскольку код находится в функции **OnPaint()**, она использует **CPaintDC**, а не **CClientDC**. Добавить необходимо следующий код:

```
void CPaintORamaDlg::OnPaint()
{
    CPaintDC dc(this);
    // Здесь последуют остальные строки
}
```

2. Функция **PlayMetaFile()**, которая отображает команды GDI, накопленные в метафайле, в качестве аргумента принимает **HMETAFILE** (дескриптор метафайла), но не **CMetaFileDC**. Создайте дескриптор **HMETAFILE** с именем **hmf**, воспользовавшись для этой цели функцией **Close()** из **CMetaFileDC**:

```
HMETAFILE hmf = m_pMF->Close();
```

3. Вызов функции **Close()** немедленно прекращает запись в исходный метафайл. Вызовите функцию **PlayMetaFile()** контекста устройства для повторного выполнения записанных ранее команд GDI. В вызов **PlayMetaFile()** передается новая переменная:

```
dc.PlayMetaFile(hmf);
```

В этому моменту записанные команды GDI уже отображались на экране, а пользователь готов возобновить рисование, так что программа должна снова регистрировать в метафайле пользовательские операции. К сожалению, в программе нет средств, позволяющих открыть закрытый метафайл. Следовательно, потребуется создать новый объект **CMetaFileDC** (в куче), скопировать содержимое старого метафайла в новый, а затем удалить старый **HMETAFILE**, *а также* старый объект **CMetaFileDC**, который породил его. И наконец, новый объект **CMetaFileDC** необходимо активизировать.

Для решения всех упомянутых задач внесите в метод **OnPaint()** следующие изменения:

4. Создайте новый объект **CMetaFileDC** для записи дополнительных команд. Используйте **new** и **Create()**, как это делалось для исходного объекта **CMetaFileDC**. В **OnPaint()** добавьте такой код:

```
CMetaFileDC* temp = new CMetaFileDC;
temp->Create();
```

5. Для обновления нового объекта **CMetaFileDC** воспользуйтесь функцией **PlayMetaFile()**, определяя в качестве получателя **CMetaFileDC**, например, так:

```
temp->PlayMetaFile(hmf);
```

После выполнения функции **PlayMetaFile()** новый **CMetaFileDC** будет содержать все операции рисования, записанные в исходном метафайле.

6. Удалите метафайл **HMETAFILE**, воспользовавшись функцией Windows API **DeleteMetaFile()**, освободите старый объект **CMetaFileDC**, расположенный в куче, а затем присвойте новый **CMetaFileDC** указателю **m_pMF**. Ниже показаны требуемые строки кода:

```
DeleteMetaFile(hmf);
delete m_pMF;
m_pMF = temp;
```

После завершения всех операций откомпилируйте и запустите приложение. Как видно на рис. 11.3, каждый раз, когда приложение минимизируется или через холст перетаскивается диалоговое окно Color, программа PaintORama воспроизводит изображение, как только холст раскрывается.

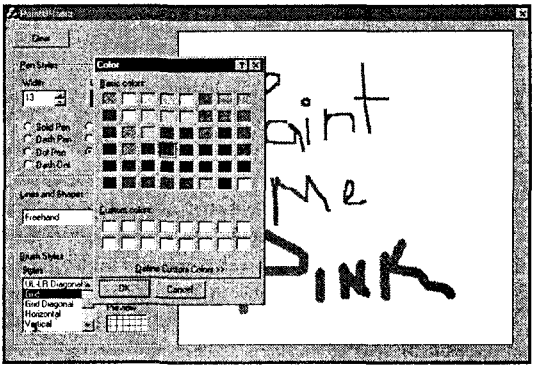


РИСУНОК 11.3. *Приложение PaintORama с активной перерисовкой*

Различные мелочи

Ранее вы добились существенного улучшения качества программы PaintORama. Однако она еще не обеспечивает корректного рисования фона, когда пользователь щелкает на кнопке Clear. Образцы цвета пера и кисти также не работают должным образом. Давайте вначале решим проблему, связанную с кнопкой Clear.

Обработка нажатия на кнопку Clear

Когда пользователь очищает холст, программа PaintORama должна начать новый документ. Новый документ не должен получать старые команды GDI — он должен начинаться заново, с новым DC.

Решение этой задачи не связано с особыми трудностями. Просто добавьте в функцию **OnClearbtn()** новый (выделенный) код их листинга 11.4.

Листинг 11.4. Пересмотренный вариант функции OnClearbtn().

```
void CPaintORamaDlg::OnClearbtn()
{
```

```
    CClientDC dc(this);
    HMETAFILE hmf = m_pMF->Close();  // Закрыть старый метафайл
    ::DeleteMetaFile(hmf);           // Освободить ресурсы GDI
    delete m_pMF;                    // Освободить ресурсы C++
    m_pMF = new CMetaFileDC;         // Разместить новый объект
    m_pMF->Create();                 // Создать ресурсы GDI
    m_pMF->SetAttribDC(dc);          // Подключиться к текущему dc

    // Рисовать на экране
    dc.SelectStockObject(NULL_PEN);
    dc.Rectangle(m_Canvas.left, m_Canvas.top,
                 m_Canvas.right+1, m_Canvas.bottom+1);

    // Рисовать в метафайле
    m_pMF->SelectStockObject(NULL_PEN);
    m_pMF->Rectangle(m_Canvas.left, m_Canvas.top,
                     m_Canvas.right+1,
                     m_Canvas.bottom+1);
}
```

Обратите внимание, что код очистки холста практически полностью совпадает с кодом функции **OnPaint()**, создающим новый объект **CMetaFileDC**. Однако в данном случае **CMetaFileDC** ассоциируется с текущим DC холста, а затем холст очищается.

ПРИМЕЧАНИЕ

Вызов функции *Rectangle()* несколько отличается от предложенного ранее варианта. Вспомните, что в функции *DrawShape()* отсекающий прямоугольник привязывается к координатам, которые хранятся в элементе данных *m_Canvas*. Отсекающий прямоугольник позволяет рисовать в любом места пространства внутри этих координат. Однако очистка холста осуществляется при помощи функции *Rectangle()*. Подобно другим функциям вывода GDI, использующим ограничивающий прямоугольник, функция *Rectangle()* является включающей/исключающей: она не рисует последний столбец пикселов справа и последний ряд пикселов снизу. Представленный здесь программный код компенсирует ошибку "сдвига на один ряд".

Начинаем заново

Новый код в функции **OnClearbtn()** исправно работает при каждом щелчке на кнопке Clear. Можно ли теперь использовать этот код для очистки холста во время запуска программы? В конце концов, серый фон используется только из-за того, что так установлено по умолчанию в классе **CDialog**.

Такого рода использование функции **OnClearbtn()** приводит к возникновению еще одной проблемы: необходимо определить, где ее вызывать. Вызывать ее в конструкторе нельзя, поскольку переменные **m_Canvas** и **m_pMF** еще не существуют. Можно попытаться поместить ее вызов в конец **OnInitDialog()**, но, к сожалению, это тоже не срабатывает.

Можно попробовать вызвать **OnClearbtn()** в начале функции **OnPaint()**, чтобы окно создавалось заново в момент, когда в нем начинается рисование. Данный способ работает, но при этом в момент поступления сообщения **WM_PAINT** стирается предыдущий рисунок.

Для решения рассматриваемой проблемы в библиотеке MFC имеется функция **OnNewDocument()** архитектуры DocView. К сожалению, время приложений DocView еще не наступило, так что придется проявить определенную изобретательность.

Вы можете избрать прямолинейное решение, обеспечив однократный вызов **OnClearbtn()** из метода **OnPaint()**. Это позволит сделать помещенная в нужное место локальная статическая переменная:

```
static bool firstTime = true;
if (firstTime)
{
     OnClearbtn();
     firstTime = false;
}
```

Поскольку переменная **firstTime** статическая, она инициализируется только один раз, при загрузке программы. Поместив показанный выше код в метод **OnPaint()** перед кодом воспроизведения метафайла, вы обеспечите белый фон для всех рисунков.

Так решается проблема серого фона. Можно было бы также продемонстрировать подход к решению проблемы некорректного рисования образцов цветов, но это практически не даст вам ничего нового. Вместо этого перейдем к использованию архитектуры DocView, что позволит разрабатывать более сложные программы.

SDIOne: переход к архитектуре DocView

Наступило время прощания с приложением PaintORama. Многие базовые идеи и подходы, заложенные в этой программе, не лишены изящества, однако необходимо двигаться в новом направлении. Итак, откроем карту и посмотрим, что ждет впереди.

В оставшейся части данной главы мы займемся построением первого из серии приложений архитектуры DocView. Как и первые версии программы PaintORama, первое приложение DocView будет простой, одноцветной программой рисования. Однако, на этот раз, помимо классов главного окна и приложения, программа будет включать класс документов и класс представлений. Новые классы существенно расширяют возможности программы.

В следующей главе классы документов и представлений исследуются более глубоко. А в последующих главах вы научитесь пользоваться Редактором меню (Menu Editor) среды Visual C++ для воспроизведения существующих возможностей программы PaintORama и добавления новых.

Построение первого приложения SDI

Теперь, когда мы знаем, куда двигаться, не будем отвлекаться ни на что другое. Закройте ваш текущий проект, выберите из главного меню File|New (Файл | Новый) и создайте новый проект MFC AppWizard (exe) с именем SDIOne.

Необходимо выполнить следующие действия:

1. В диалоговом окне AppWizard Step 1 выберите переключатель Single Document и убедитесь в том, что установлен флажок Document/View Architecture Support (Поддержка архитектуры "документ/представление"). Окно должно соответствовать рис. 11.4.

2. Не изменяйте значений; установленных по умолчанию в диалоговом окне AppWizard Step 2, поддержка баз данных пока не нужна.
3. Выберите None из опций поддержки составного документа в диалоговом окне AppWizard Step 3. Кроме того, сбросьте флажок ActiveX Controls (см. рис. 11.5).

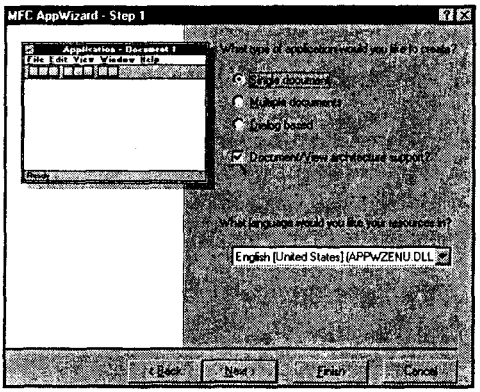


РИСУНОК 11.4. *Приложение SDIOne: окно MFC AppWizard Step 1*

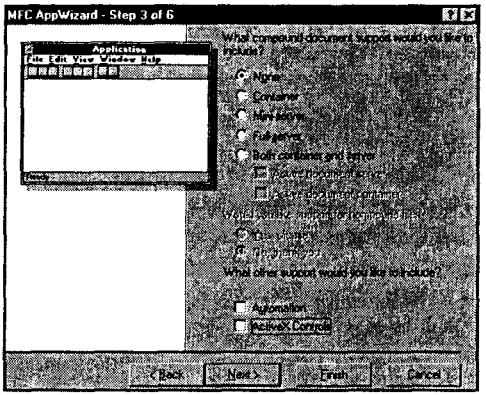


РИСУНОК 11.5. *Приложение SDIOne: окно MFC AppWizard Step 3*

4. В диалоговом окне AppWizard Step 4 оставьте все значения по умолчанию. Должны быть установлены флажки Docking Toolbar, Initial Status Bar, Printing And Print Preview и 3D Controls, в то время как другие флажки должны быть сброшены. Выберите Normal toolbars (обычные панели инструментов) вместо стиля ReBar. Прежде чем покинуть этот экран, щелкните на Advanced (Дополнительные параметры).
5. В диалоговом окне Advanced Options (Дополнительные параметры) выберите закладку Document Template Strings (Строки шаблона документа). Все установленные в ней значения допустимы, но вы должны указать расширения файла. Введите "sdi1", как показано на рис. 11.6. Щелкните на Close. Вернувшись в диалоговое окно AppWizard Step 4, щелкните на Next.
6. Оставьте без изменений значения по умолчанию в диалоговом окне AppWizard Step 5. Щелкните на Next для перехода в диалоговое окно Step 6, которое позволит пересмотреть полученный класс. Здесь также оставьте без изменений значения по умолчанию. Щелкните на Finish, чтобы отобразить на экране диалоговое окно New Project Information, показанное на рис. 11.7. За исключением каталога приложения, который будет соответствовать вашей конкретной системе, вы должны получить на экране точно такую же информацию.

Обработка документа в SDIOne

Интерактивные приложения, создаваемые до сих пор, содержали только два класса — само приложение и окно приложения. С другой стороны, в приложении SDIOne имеется пять классов: класс приложения, класс главного окна, класс представления, класс документа и класс диалогового окна About.

В следующей главе вы узнаете, какая ответственность закреплена за каждым из перечисленных классов. Сейчас можно отметить только то, что класс представлений **CSDIOneView** будет использоваться для рисования, а класс документов **CSDIOneDoc** — для хранения данных рисунка.

Поскольку класс представления требует взаимодействия с классом документа, но отнюдь не наоборот, изучение начинается именно с класса документа. В SDIOne будет реализована только та часть программы PaintORama, которая обеспечивала рисунки от руки. Следовательно, в документе будут храниться только сегменты линий — их можно хранить в виде пар объектов **CPoint**.

Объекты CPoint и класс CArray

В поставляемой библиотеке MFC определено несколько классов коллекций, упорядоченных в две обширных категории. Ранние версии MFC содержали классы коллекций, основанные на хранении указателей на объекты и простых типов. В версии 3 MFC введено новое множество классов коллекций, в основе которых лежат шаблоны C++.

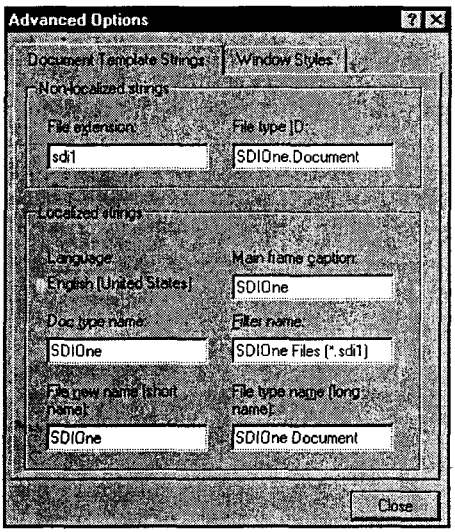


РИСУНОК 11.6. *Приложение SDIOne: окно MFC AppWizard Step 4*

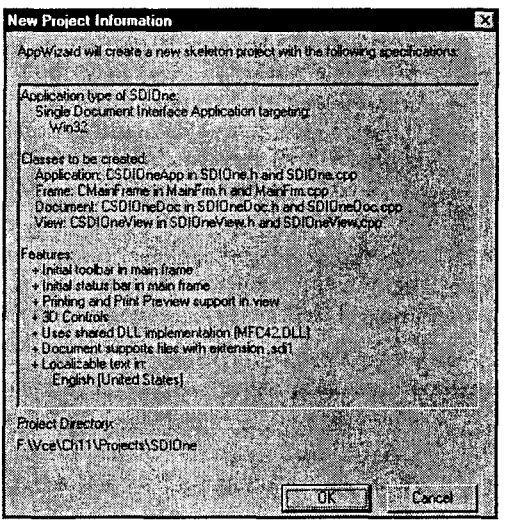


РИСУНОК 11.7. *Приложение SDIOne: окно информации о новом проекте*

Классы коллекций, в основе которых лежат шаблоны, обладают рядом преимуществ по сравнению с более ранними классами, не основанными на шаблонах. Например, классы на базе шаблонов позволяют создавать коллекции, безопасные к приведению типов. Более ранние классы коллекций хранили только обобщенные указатели, которые требовали выполнения потенциально опасных операций приведения типов. Например, выбранный из некоторой коллекции объект должен был быть явно приведен к правильному типу, либо это осуществляла вспомогательная функция, специально написанная для этих целей. В случае использования классов на базе шаблонов, проверку типов и их преобразование осуществляет C++, тем самым гарантируя, что коллекция содержит только правильные типы объектов.

В приложении SDIOne будет использоваться класс **CArray** коллекции на базе шаблонов. Объекты **CArray** ведут себя подобно обычным массивам языка C, но за одним исключением: объекты **CArray** автоматически изменяют размеры, в то время как встроенные массивы C++ этого не делают.

Реализация методов класса документа

Для запоминания данных в виде, допускающем их обработку классом представления, в классе документа должны быть реализованы функции, обеспечивающие выполнение следующих операций:

- *Сохранение* ***CPoint***. Класс представления не знает или не следит за тем, хранит ли класс документа объект **CPoint** в **CArray** или нет. Класс представления просто передает объект **CPoint** документу, а документ запоминает его для дальнейшего обращения и использования. Назовите эту функцию **AddPoint()**.
- *Определение количества объектов* ***CPoint****, содержащихся в документе*. Когда класс представления будет вызван для повторного воспроизведения рисунка, он должен воспроизвести каждую линию по отдельности. Однако представление не должно рыться в документе в поисках этих линий. Вместо этого вы должны применить принцип инкапсуляции, в соответствие с которым массив, хранящийся в классе документа, превращается в приватный элемент данных. Поскольку представление не имеет доступа к этому приватному элементу данных, требуется специальная функция общего доступа, которая подсчитывает количество объектов **CPoint**. Назовите эту функцию **NumPoints()**.
- *Получение объекта* ***CPoint*** *по значению его индекса*. После того как представление определит, сколько имеется точек, оно будет выбирать каждую из них. Назовите эту функцию **GetPoint()**.

Кроме того, в программу потребуется добавить код создания и инициализации переменной коллекции **CArray**. Для этого выполните следующие действия:

1. Откройте и расширьте класс **CSDIOneDoc** в окне ClassView, как показано на рис. 11.8.
2. Добавить вручную новую приватную переменную в класс **CSDIOneDoc** (это делается вручную, поскольку Add Member Variable шаблонов не поддерживает). Присвойте этой переменной имя **m_data** и назначьте ей тип **CArray**. Поскольку **CArray** — это шаблон, ему следует передать аргументы (в угловых скобках) для инициализации. Класс **CArray** требует двух аргументов: тип значений, которые будут храниться в массиве, и тип аргументов, используемых для доступа к объектам, хранящимся в массиве. Как правило, второй аргумент представляет собой ссылку на тип значения, сохраняемого в массиве. В данном случае оба аргумента бу-

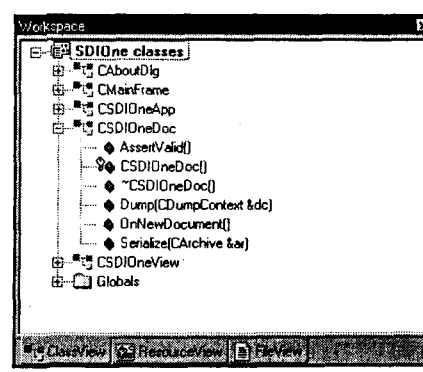


РИСУНОК 11.8. *Просмотр класса CSDIOneDoc в окне ClassView*

дут объектами **CPoint**. Вот как выглядит объявление, помещаемое в CSDIOneDoc.h:

```
private:
      CArray<CPoint, CPoint> m_data;        // Здесь
                             // сохраняется документ
```

3. В начало CSDIOneDoc.h добавьте приведенный ниже оператор **#include**. Он требуется всякий раз при включении класса шаблонов:

```
#include <afxtempl.h>
```

4. В принципе, инициализировать **CArray** подобно массиву переменных **m_data** не обязательно. По умолчанию при добавлении переменной в заполненный **CArray** размер массива увеличивается на один элемент. Поскольку элементы сохраняются в непрерывной области памяти, весь массив должен быть перераспределен. Во избежание этого, можно воспользоваться методом **SetSize()**, чтобы задать начальный размер, а также *фактор возрастания*, который определяет, сколько элементов добавляется при расширении массива. Используя окно ClassView, найдите функцию **OnNewDocument()**, которая вызывается каждый раз при создании нового документа. Поместите следующую строку непосредственно перед оператором возврата:

```
m_data.SetSize(0, 128);
```

Оператор инициализирует переменную **m_data** как пустой массив, который при каждом расширении возрастает на 128 элементов.

5. Воспользуйтесь диалоговым окном Add Member Function для создания объявления и остова функций **AddPoint()**, **GetPoint()** и **NumPoints()**. Типами значений, возвращаемых функциями являются, соответственно, void, **CPoint** и **int**. **AddPoint()** принимает в качестве аргумента одиночный **CPoint**, в то время как функция **GetPoint()** — индекс типа **int**. Функции **NumPoints()** аргументы не нужны. После добавления в программу остовов этих функций завершите их строками, выделенными в листинге 11.5.

Листинг 11.5. Методы AddPoint(), GetPoint() и NumPoints().

```
void CSDIOneDoc::AddPoint(CPoint point)
{
    m_data.Add(point);
}
CPoint CSDIOneDoc::GetPoint(int item)
{
    return m_data[item];
}
int CSDIOneDoc::NumPoints()
{
    return m_data.GetSize();
}
```

6. В функциях, определенных в **CSDIOneDoc**, внимание на себя обращает функция **Serialize()**. **Serialize()** вызывается тогда, когда каркас приложения DocView предполагает, что документ будет сохраняться. Чтобы иметь возможность сохранить или выбрать документ, потребуется записать программный код, взаимодействующий с классом **CArchive** при чтении/записи данных. К счастью, класс **CArray** уже знает, как это сделать. Весь код функции

CSDIOneDoc::Serialize() можно заменить единственной строкой, отмеченной в листинге 11.6.

Листинг 11.6. Метод CSDIOneDoc Serialize().

```
void CSDIOneDoc::Serialize(CArchive& ar)
{
    m_date.Serialize(ar);
}
```

Поддержка представления в программе SDIOne

Программный код рисования, использующий класс представления, оказывается намного проще программы PaintORama. Например, нет необходимости проверять, находитесь ли вы внутри области рисования, что обязательно для программы PaintORama — все окно класса представления являет собой область рисунка, поэтому все заботы, связанные с отсечением или определением положения мыши, отпадают.

Для класса **CSDIOneView** потребуется реализовать четыре функций. Три из них, **OnLButtonDown()**, **OnMouseMove()** и **OnLButtonUp()**, работают точно так же, как и аналогичные функции в PaintORama. Четвертая, **OnDraw()**, во многом подобна функции **OnPaint()**, с которой вы сталкивались на протяжении нескольких последних глав.

Обработчики нажатия кнопок мыши

Чтобы запустить в действие класс представления, необходимо выполнить следующие действия:

1. Выберите класс **CSDIOneView** в окне ClassView, как показано на рис. 11.9. Расширьте этот класс, чтобы в нем были видны все перечисленные выше функции.

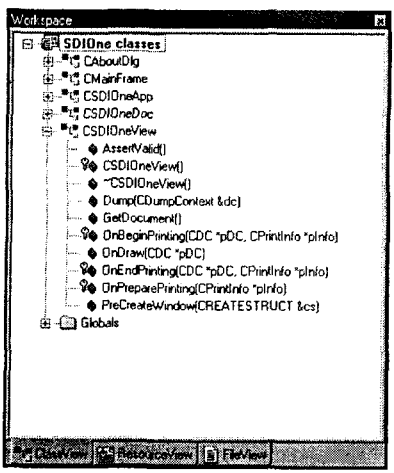


РИСУНОК 11.9. *Класс CSDIOneView в окне ClassView*

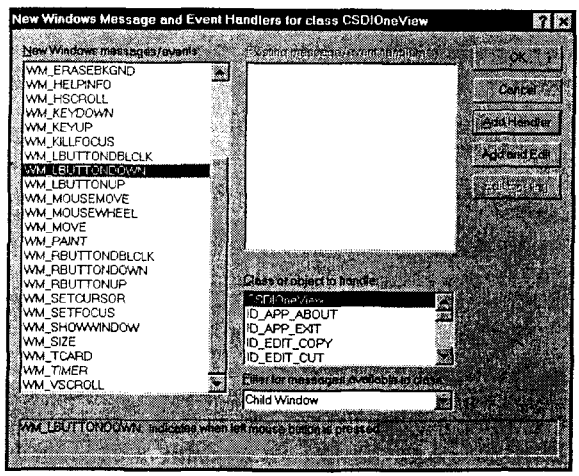


РИСУНОК 11.10. *Добавление обработчика сообщений WM_LBUTTONUP*

2. Добавьте в класс приватный элемент данных типа **CPoint** с именем **m_LineStart**. (Для этой цели либо воспользуйтесь диалоговым окном Add

Member Variable, либо сделайте соответствующие изменения в файле CSDIOneView.h вручную.) Добавленная переменная служит тем же целям, что и аналогичная переменная в программе PaintORama.

3. Используя контекстное меню, добавьте обработчик сообщений **WM_LBUTTONDOWN**, как показано на рис. 11.10. Функция **OnLButtonDown()** решает две задачи: устанавливает начальную позицию сегмента линии и захватывает мышь, чтобы можно было быть уверенным в получении сообщения **WM_LBUTTONUP**. Замените код функции **OnLButtonDown()** кодом, выделенным в листинге 11.7.

Листинг 11.7. Функция OnLButtonDown().

```
void CSDIOneView::OnLButtonDown(UINT nFlags, CPoint point)
{
      m_LineStart = point;
      SetCapture();
}
```

4. Добавьте аналогичный обработчик для сообщения **WM_LBUTTONUP**. Код обработчика показан в листинге 11.8; он оказывается даже проще кода **OnLButtonDown()** — выполняется лишь освобождение мыши от захвата.

Листинг 11.8. Функция OnLButtonUp().

```
void CSDIOneView::OnLButtonUp(UINT nFlags, CPoint point)
{
      ReleaseCapture();
}
```

5. Третий обработчик сообщений Windows, как и ранее, выполняет фактическое рисование. Никакого кода, проверяющего факт нахождения в области холста, не требуется, однако необходимо проверить, нажата ли левая кнопка мыши, — именно это отличает процесс перетаскивания мыши. Программный код, отображающий сегмент линии (получение контекста устройства и вызов функций **MoveTo()** и **LineTo()**), берется из PaintORama без изменений. Тем не менее, прежде чем сегмент линии будет вычерчен, две конечных точки пересылаются в объект документа, который выбирается при помощи вызова метода **GetDocument()**. Окончательный вид функции показан в листинге 11.9.

Листинг 11.9. Функция OnMouseMove().

```
void CSDIOneView::OnMouseMove(UINT nFlags, CPoint point)
{
      if (nFlags & MK_LBUTTON)
      {
            // Сохранить в классе документа
            CSDIOneDoc* pDoc = GetDocument();
            ASSERT_VALID(pDoc);
            pDoc->AddPoint(m_LineStart);
            pDoc->AddPoint(point);

            CClientDC dc(this);
            dc.MoveTo(m_LineStart);
            dc.LineTo(point);
            m_LineStart = point;
      }
}
```

Работа с функцией OnDraw()

В приложениях DocView класс представления не производит обработку сообщений **WM_PAINT**, как это имело место в предыдущих приложениях. Вместо этого каркас DocView осуществляет обработку этих сообщений самостоятельно. Разумеется, реализовать код рисования все равно придется. В приложении DocView этот код помещается в виртуальную функцию **OnDraw()**, а не в **OnPaint()**.

Вы, несомненно, заметите некоторую странность поведения функции **OnDraw()**: получать собственный контекст устройства не потребуется. Это объясняется тем, что каркас DocView использует код, помещенный в **OnDraw()**, в нескольких различных DC. Фактически используемый контекст передается в виде указателя на базовый класс DC, **CDC**. Один и тот же программный код используется для рисования на экране, вывода на принтере и для режима предварительного просмотра печати. Это фактически равносильно бесплатному получению двух дополнительных контекстов!

В результате код функции **OnPaint()** оказывается проще, чем аналогичный код из приложения PaintORama. Каркас передает вам указатель на DC и он же выбирает указатель на класс документа в строках:

```
CSDIOneDoc* pDoc = GetDocument();
ASSERT_VALID(pDoc);
```

Для завершения **OnDraw()** потребуется обратиться к документу с запросом о количестве сохраненных в нем точек, затем получить и отрисовать каждую из них, используя уже знакомые функции DC. Поместите в функцию **OnDraw()** строки, отмеченные в листинге 11.10.

Листинг 11.10. Функция OnDraw() класса CSDIOneView.

```
void CSDIOneView::OnDraw(CDC* pDC)
{
      CSDIOneDoc* pDoc = GetDocument();
      ASSERT_VALID(pDoc);

 // ЧТО СДЕЛАТЬ: добавить здесь код рисования по собственным данным
      int nPoints = pDoc->NumPoints();
      CPoint start, stop;
      for (int item = 0; item < nPoint; item += 2)
      {
            start = pDoc->GetPoint(item);
            stop = pDoc->GetPoint(item + 1);
            pDC->MoveTo(start);
            pDC->LineTo(stop);
      }
}
```

Ближайшая перспектива

Откомпилируйте и запустите приложение SDIOne. Несмотря на то что оно обладает меньшими возможностями, чем программа PaintORama, оно построено на более прочном фундаменте. По мере добавления новых возможностей в PaintORama, она становится менее устойчивой и более сложной, и все же, не

решает всех тех задач, которые способна решать программа SDIOne. Ниже приводятся некоторые тесты, которые потребуется выполнить в SDIOne:

- Создать новый рисунок, воспользовавшись опцией File|New или пиктограммой New File панели инструментов. В программе PaintORama для обработки новых документов вы помещали в функцию **OnClearbtn()** дополнительный код.
- Нарисовать несколько фигурок, а затем изменить размеры окна. Обратите внимание, что код функции **OnDraw()** сохраняет пропорции изображений, даже в случае минимизации или максимизации размеров окна, в чем несложно убедиться, взглянув на рис. 11.11.
- Напечатайте документ или воспользуйтесь режимом предварительного просмотра печати. И в данном случае эти функции работают без необходимости добавления каких-либо кодов с вашей стороны (хотя вас может не удовлетворить то, как масштабируется ваше изображение). Режим предварительного просмотра печати показан на рис. 11.12.
- Воспользуйтесь опциями меню File | Save или File | Save As и вы увидите, что программный код, добавленный в метод **Serialize()** документа, может читать и записывать в файлы. Поскольку было использовано нестандартное расширение файла (.sdi1), то для запуска приложения SDIOne, можно также дважды щелкнуть на сохраненном файле в Windows Explorer (Проводнике).

Очевидно, что программа унаследовала от архитектуры DocView гораздо больше свойств, чем кажется на первый взгляд. В следующей главе мы сорвем покров тайны с некоторых ее аспектов и уделим большее внимание на анализ внутренних свойств программы SDIOne и классов, которые она использует.

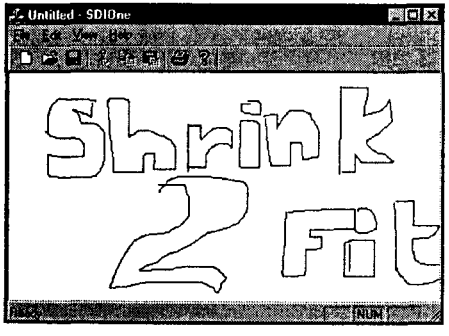


РИСУНОК 11.11. *Изменение размеров окна приложения SDIOne*

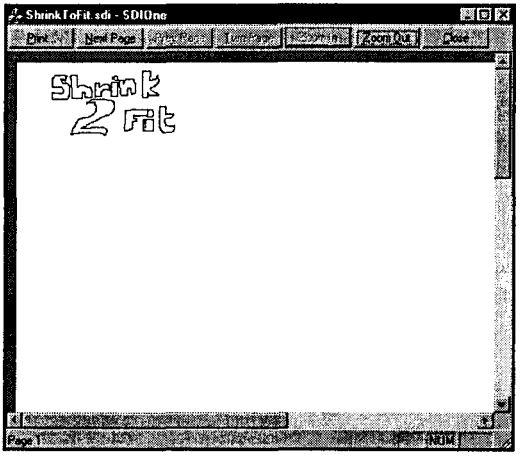


РИСУНОК 11.12. *Предварительный просмотр печати в приложении DIOne*

Глава 12

Особенности архитектуры DocView

Вспомните очень маленькую компанию — знаменитый бизнес "Mom-and-Pop". Сначала Mom и Pop участвуют в любом деле, от мытья окон до стряпания книг. Однако по мере расширения бизнеса, они нанимают специалистов — юристов, бухгалтеров, вахтеров — для достижения большей эффективности. Примерно то же можно сказать и о DocView.

Для программ, не использующие архитектуру DocView, характерны два класса "Mom" и "Pop" общего назначения, непосредственно порожденные от **CWnd** и **CWinApp**. В более сложной среде DocView в программах имеются по меньшей мере четыре специальных класса. Каждый из этих классов тесно взаимодействует с другими, но на каждый из них возлагаются четкие и конкретные задачи.

В этой главе вы узнаете много нового об основных четырех игроках архитектуры DocView: о классе приложений, классе рамочных окон, классе документов и классе представлений. На протяжении большей части главы основное внимание будет уделяться общему рассмотрению. Ближе к концу главы мы рассмотрим крупным планом класс приложений, воспользовавшись для этой цели программой SDIOne из предыдущей главы. Изучение класса **CSDIOneApp** и его функции **InitInstance()** послужит хорошим введением в новые свойства, вносимые архитектурой DocView.

Однако прежде чем мы начнем, ответим сначала на один вопрос, устранив тем самым очередное препятствие, мешающее продвигаться вперед: что это, в конце концов за штука, архитектура DocView? По крайней мере она кажется более сложным делом, чем интерактивные приложения, построением которых мы занимались. Какой от нее будет толк?

Кто, что и почему?

В программах, не использующих архитектуру DocView, нет класса документов и класса представлений, типичных для DocView-программ. Что вносят эти новые классы в ваши программы?

В среде DocView каждое приложение работает с конкретными типами данных: например, текстовый процессор работает с символьными данными, электронные таблицы — с числовыми данными, графические программы — с графическими данными. В приложениях DocView *класс документов* работает с одним видом данных, манипулирует ими и сохраняет или выбирает их из внешних файлов, таких как дисковые файлы.

Можно записывать программы, не имеющие пользовательского интерфейса — программы, которые осуществляют выборку данных, их обработку, после чего отсылают их по назначению. Однако архитектура DocView не разрабатывалась для приложений такого типа. Она скорее создавалась для итеративных программ, которые дают пользователям возможность просматривать и манипулировать документами. В программе DocView, класс представлений воспроизводит документ для пользователя.

Типы программ DocView

С использованием классов архитектуры DocView можно разрабатывать несколько видов программ:

- *Программы, которые работают с одним или несколькими различными типами документов.* Хороший пример такой программы — это Microsoft Works. На практике преобладают программы с одним типом данных.
- *Программы с однодокументным интерфейсом (SDI — Single Document Interface).* Такие программы (Windows Notepad, например) могут открывать в рабочем сеансе только один документ, даже если они способны работать с несколькими типами документов.

- *Программы с многодокументным интерфейсом (MDI — Multiple Document Interface).* Программы этого типа могут открывать несколько документов одновременно. У таких программ имеется одно главное окно и несколько второстепенных дочерних окон, в которых появляются документы. Microsoft Word, Microsoft Excel и даже наша собственная программа NotePod являются приложениями MDI. Очень важно понять различие между одновременным открыванием документов нескольких *типов* и просто открыванием нескольких документов.
- *Программы с одним или многими представлениями.* В приложении MDI каждый документ автоматически имеет по меньшей мере одно представление. В приложении SDI можно воспользоваться *окнами с разбиением* (splitter windows), в каждом из которых воспроизводится несколько представлений одного и того же документа.
- *Программы с различными типами представлений.* Такие программы могут использоваться только одно представление в каждый момент времени. Программа, поддерживающая множество типов представлений, обеспечивает возможность просмотра документа несколькими способами, например, в виде таблицы или в виде карты. Пользователи, которые работали с современными пакетам электронных таблиц, знакомы с такими типами программ.

Достоинства архитектуры DocView

Теперь, когда вы понимаете, *кто* и *что*, сосредоточимся на *почему*. Существуют две причины, почему следует затратить время на то, чтобы научиться хорошо пользоваться архитектурой DocView.

Во-первых, это принесет несомненную пользу. Приложения DocView помогут применять принцип *модульной структуры* программ. Программы, разбитые на модули с конкретно заданными границами, обязанностями, коммуникационными путями, более понятны; они обладают большей надежностью и более просты в настройке по сравнению с программами без модульной структуры. Архитектура на основе диалоговых окон хорошо зарекомендовала себя в небольших программах, но для объемных программ потребуется помощь архитектуры DocView, чтобы наладить взаимодействие всех элементов программы.

Рекламные агенты давно обнаружили, что лозунг "Это принесет несомненную помощь" редко стимулирует продажу продуктов. К счастью, имеются гораздо более убедительные причины для применения архитектуры DocView: у вас появляется шанс стать неотразимым для противоположного пола, приобрести новый изысканный спортивный автомобиль с откидным верхом и провести остаток жизни, развлекаясь на пляжах Майями.

Однако погодите. Речь идет вовсе не об архитектуре DocView; это об антикариесном зубном порошке от компании DocWhite. Если вы используете архитектуру DocView, то просто обнаруживаете, что вам становится проще разрабатывать приложения. Разумеется, более простая разработка приложений означает больше свободного времени. Если вы хотите провести это время на пляжах Майями, то это ваше личное дело.

Выражаясь серьезно, только применяя DocView, можно полностью воспользоваться всеми преимуществами системы MFC, которая обеспечивает внесение

многих усовершенствований в каркасы на базе DocView. Вы хотите в своих программах использовать панели инструментов и строки состояния? Можно либо потратить непонятно какое время на приготовление собственной смеси, либо воспользоваться простой структурой, предоставляемой библиотекой MFC. Догадываетесь, какой? Версия MFC предполагает работу в рамках архитектуры DocView. Требуется упростить процедуру печати? Опять-таки, MFC строит свои каркасы для реализации печати на основе архитектуры DocView. Выбирая для себя каркас MFC DocView, вы получаете доступ к тысячам строк готового программного кода, реализующих большую часть наиболее часто используемых в приложениях программных возможностей. При этом дополнительных усилий для их использования тратить не придется.

Теперь рекламная фраза звучит так: "DocView имеет превосходный вкус и полезна для вашего здоровья!" Кто сказал, что реклама — это умирающее искусство?

Архитектура DocView: кто с кем разговаривает?

Начнем нашу экскурсию в страну DocView с изучения четырех основных действующих лиц. Когда в главе 11 вы обращались к AppWizard с запросом создать приложение SDIOne, он подготовил четыре класса: класс приложений, класс главного или рамочного окна, класс документов и класс представлений. Эти четыре класса показаны в нижней панели рис. 12.1. Верхняя панель содержит классы каркаса MFC, от которых порождаются классы вашего приложения.

Коротко рассмотрим все четыре класса, созданные AppWizard.

ПРИМЕЧАНИЕ

На рис. 12.1 не показаны все классы, которые используются для создания того или иного приложения DocView. В этой главе не обсуждаются некоторые промежуточные классы, такие как *CWinThread*, находящийся между классами *CcmdTarget* и *CWndApp*; она опущена, чтобы не загромождать рисунок. Другие классы, подобные *CSingleDocTemplate*, рассматриваются по мере их упоминания.

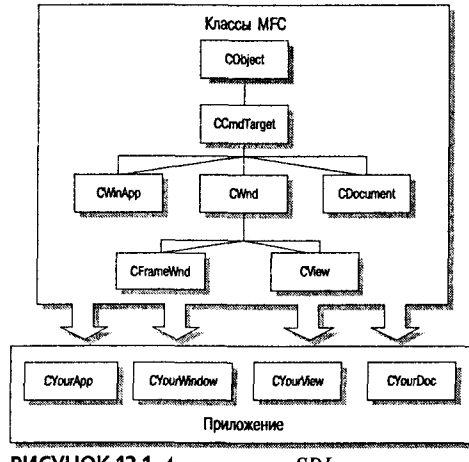


РИСУНОК 12.1. *Архитектура SDI-приложения*

Класс приложения

В каждой DocView-программе присутствует класс приложения, порожденный от класса **CWinApp**. Аналогично классам приложений, с которыми вы сталкивались в программах с архитектурой, отличной от DocView, он обеспечивает решение задач загрузки и создания программы, принимает сообщения от Windows и распределяет сообщения соответствующим адресатам.

На рис. 12.1 этот класс называется **CYourApp**, однако данное имя выбрано лишь для удобства изложения. В каждом приложении имена классов выбираются по собственному усмотрению. Например, в приложении SDIOne класс приложения получил имя **CSDIOneApp**, а его файлами являются SDIOne.h и SDIOne.cpp. Класс приложения и его функция **InitInstance()** подробно рассматривается далее в главе.

Какую информацию несет имя?

Когда AppWizard создает новый проект, он пользуется несколькими простыми правилами для присвоения имен создаваемым классам и файлам, содержащим реализацию этих классов. В большинстве случаев имена, предлагаемые AppWizard, можно изменить. Например, возможно, требуется, чтобы имена соответствовали соглашениям, действующим в рамках вашей компании.

Когда AppWizard создает экземпляр одного из четырех классов архитектуры DocView, он записывает файл заголовков (.h) и файл реализации (.cpp). Объявление класса находится в файле заголовков, а объявление каждой функции — в файле реализации. (Некоторые программисты называют файл заголовков *интерфейсным* файлом, поскольку он содержит всю информацию, необходимую для использования класса.)

AppWizard присваивает классу главного окна имя CMainFrame, независимо от названия проекта. Файл заголовков класса окна получает имя MainFrm.h, а его файл реализации — имя Mainfrm.cpp. Для присвоения имен остальным классам AppWizard использует несколько простых правил. Имя вашего проекта например, SDIOne, применяется в качестве *базового* при именовании файлов и классов. В начало каждого имени класса добавляется прописная буква "C", что означает класс. Имя класса дополняется суффиксом, описывающим его тип: Doc, View или App. Имена файлов, содержащие классы документов и приложений, образуются путем удаления первой буквы "C" из названия класса, а класс приложений сохраняется в файле, которому присвоено имя проекта.

В большинстве нет особой необходимости соблюдать соглашения об именах, применяемые AppWizard. Можно изменить имя любого класса или файла, за исключением тех, которые принадлежат классу приложений. Главный класс приложения должен храниться в файлах заголовков и реализации, имеющих то же базовое имя, что и проект.

Класс главного окна

До сих пор в качестве класса главного окна программы использовался **CDialog** — подкласс **CWnd**. DocView-программы порождают свои классы главного (рамочного) окна из класса **CFrameWnd**, также являющегося подклассом класса **CWnd**. Подобно тому как класс **CDialog** поддерживает разработку визуального интерфейса через Dialog Editor, так и класс **CFrameWnd** обладает собственными особенностями.

По умолчанию каждая DocView-программа получает свой класс **CMainFrame**, который является подклассом **CFrameWnd**. Класс **CMainFrame** — перекрываемое окно высокого уровня с изменяемыми размерами — содержит один или большее количество экземпляров класса представлений приложения. Класс **CMainFrame** может содержать элементы управления, такие как меню, панели инструментов, строки состояний. В дополнение к этому, он ведет себя как диспетчер сообщений,

генерируемых меню и панелями управлений, автоматически направляя их соответствующему получателю.

Более подробно класс **CFrameWnd** изучается в главах 13 и 14, когда рассматривается работа с меню. Сейчас кратко рассмотрим класс **CMainFrame** программы SDIOne.

Поскольку класс **CMainFrame** наследует многие функциональные возможности от **CFrameWnd**, он относительно прост. **CMainFrame** определяет конструктор по умолчанию, виртуальный деструктор, виртуальную функцию **PreCreateWindow()** и обработчик сообщений Windows **WM_CREATE** — функцию **OnCreate()**. Кроме того, в этом классе имеются два защищенных элемента данных — **m_wndStatusBar** и **m_wndToolBar**, которые ссылаются на строку состояния и на панель инструментов главного окна.

Конструктор класса CMainFrame

Как это ни удивительно, но конструктор класса **CMainFrame** является защищенным **(protected)**. Поэтому создавать объекты **CMainFrame** при помощи **new** или в виде локальных переменных в стеке нельзя. Это выглядит несколько своеобразно. Это же покажется еще более странным, когда вы осознаете, что конструкторы классов документов и представлений также защищенные. Какой смысл иметь упомянутые классы, если для них нельзя создать объектов?

Как и для большинства вопросов, имеющих отношение к MFC, ответ далеко не очевиден. При глубоком исследовании файла заголовков, определяющего класс **CMainFrame**, обнаруживается зашифрованная строка, которая следует непосредственно за объявлением конструктора:

```
DECLARE_DYNCREATE(CMainFrame)
```

Аналогичная строка появляется в файле реализации MainFrm.cpp:

```
IMPLEMENT_DYNCREATE(CMainFrame, CFrameWnd)
```

Из того факта, что имена представлены прописными буквами, следует, что это макрокоманды MFC. Данные макрокоманды добавляют в определения классов набор скрытых функций, обеспечивающих возможность динамического ("on-the-fly") создания объектов. Мы рассмотрим эти вопросы несколько глубже во время изучения функции **InitInstance()** приложения и назначения класса шаблонов документов.

При окончательном анализе выясняется, что конструкторы классов документов, представлений или рамочных окон защищены просто потому, что соответствующие объекты никогда не потребуется создавать явно — это дело каркаса приложения. Сокрытие создания объекта за макрокомандой **DYNCREATE** уменьшает количество проблем, которые должны решаться.

Методы класса CMainFrame

Класс **CMainFrame** необычен для среды MFC, поскольку имеется возможность написать полнофункциональную MFC-программу, которая не производит модификации класса **CMainFrame**. Но это не имеет места для классов документов и представлений — эти классы не способны делать что-либо интересное до тех пор, пока в них не будет добавлено определенное поведение.

Программисты иногда вносят изменения в класс **CMainFrame**, часто для помещения в него метода, который реагирует на выбор в меню или в панели уп-

равления. Аналогично, если функции, производящие обработку команд, требуют наличия элементов данных (что они обычно и делают), следует воспользоваться конструктором класса **CMainFrame**, чтобы инициализировать их. Распределенная под элемент данных память должна освобождаться в деструкторе класса **CMainFrame**.

Программисты редко вносят изменения в методы **PreCreateWindow()** и **OnCreate()**, сгенерированные AppWizard. MFC вызывает **PreCreateWindow()** перед созданием рамочного окна, предоставляя возможность изменить его стиль за счет модификации **CREATESTRUCT**, передаваемой в качестве аргумента.

Метод **OnCreate()** вызывает метод создания базового класса **CFrameWnd::OnCreate()**, после чего устанавливает панель инструментов и строку состояния. Можно модифицировать **OnCreate()** таким образом, чтобы характеристики панели инструментов или строки состояния изменились. Например, можно изменить количество панелей, отображаемых в строке состояния. Последние три строки в **OnCreate()**

```
m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
EnableDocking(CBRS_ALIGN_ANY);
DockControlBar(&m_wndToolBar);
```

делают оконные панели инструментов *стыкуемыми (dockable)*. Пользователь получает возможность захватить стыкуемую панель управления и зафиксировать ее сбоку рамочного окна или превратить ее в свободно плавающее окно, как показано на рис. 12.2. Если требуется нестыкуемая панель инструментов, просто удалите три строки, представленные выше.

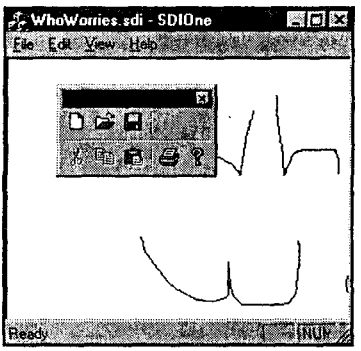


РИСУНОК 12.2. *Использование стыкуемой панели инструментов в приложении SDIOne*

Класс документов

Как упоминалось ранее, программа может поддерживать один тип документов либо много различных типов документов. Тем не менее, AppWizard создает один класс документов для любого разрабатываемого приложения — будь то SDI- или MDI-приложение. Вам решать, нужно ли добавлять дополнительные типы документов.

Согласно рис. 12.1, класс документов **CYourDoc** порожден от класса MFC **CDocument**. В приложении SDIOne MFC присваивает аналогичному классу имя **CSDIOneDoc** и запоминает его в файлах CSDIOneDoc.h и CSDIOneDoc.cpp.

Когда вы получаете свой класс документов прямо с фабрики MFC, он не знает, что делать дальше. В конце концов, AppWizard не известно, *что* разрабатывается — текстовый процессор, приложение электронных таблиц или программа рисования. Очевидно, что документы для таких приложений существенно отличаются друг от друга. В этой связи класс документов, необходимо изменить, добавив в него переменные для хранения данных приложения и методы, обеспечивающие изменение и манипулирование данными.

Несмотря на то что AppWizard не может генерировать программный код, реализующий всю функциональность конкретного приложения, класс документов, который он предоставляет, содержит каркас, существенно упрощающий добавление необходимого кода. Класс **CSDIOneDoc**, построенный AppWizard, включает в себя следующие методы:

- *Защищенный конструктор.* Будучи членом трио DocView, класс **CSDIOneDoc** допускает создание объекта только через посредство динамического метода **CreateObject()**, доступного благодаря паре макрокоманд **DYNACREATE()**. Каркас приложения создает необходимые объекты практически во всех случаях.
- *Виртуальный деструктор.* Несмотря на то что тело виртуального деструктора пусто, он имеет важное значение, ибо обеспечивает вызов деструктора базового класса **CFrameWnd** каждый раз, когда удаляется указатель на объект **CSDIOneDoc**. Без виртуального деструктора ваша программа может стать причиной утечки памяти.
- *Две виртуальных функции, **OnNewDocument()** и **Serialize()**.* Эти функции практически всегда должны перекрываться. MFC вызывает функцию **OnNewDocument()** при каждой загрузке нового документа. Функцию **OnNewDocument()** можно использовать для инициализации элементов данных класса документов, приводя их в соответствующее состояние. Метод **Serialize()** выполняет различные вспомогательные операции, необходимые для записи и считывания документов из постоянных файлов. Следует просто принять меры, чтобы элементы данных документа были *упорядочиваемыми*, а затем отослать их в предназначенный для этой цели объект **CArchive** — обо всем остальном позаботится каркас. Более подробно о том, что означает упорядочиваемость, а также о том, как обращаться с файлами, вы узнаете из глав 15 и 16, когда мы займемся проблемой дальнейшего расширения класса **CSDIOneDoc**.
- *Две виртуальных отладочных функции, **AssertValid()** и **Dump()**.* Эти функции используются для вывода диагностической информации. Они должны присутствовать только в случае, когда создается отладочная версия программы. Каркас обращается к ним при возникновении каких-либо проблем. **AssertValid()** можно перекрыть, обеспечив проверку состояния переменных внутри документов. Метод **Dump()** применяется для отображения в отладочном окне значений полей документа.

Методы, унаследованные от CDocument

В дополнение к этим локально определенным функциям, которые можно перекрыть, класс **CSDIOneDoc** наследует несколько важных методов из класса **CDocument**. Эти унаследованные методы применяются при работе с документами и представлениями в последующих главах. Данный раздел можно рассматривать в качестве введения.

В своей работе вы часто будете пользоваться несколькими важными методами класса **CDocument**. Эти методы разделяются на три категории: доступ к представлениям, информация о документе и состояние модификации документа:

- *Доступ к представлениям.* Класс **CDocument** предоставляет пользователю три метода, которые существенно упрощают работу с представлением (или представлениями), присоединенным к документу. Вспомните, что одному документу можно сопоставить несколько представлений, но каждое представление должно быть присоединено только к одному документу. Получить представление, присоединенное к документу, можно за счет обращения к функции **GetFirstView()**. В случае нескольких представлений функция **GetNextView()** должна вызываться до тех пор, пока она не возвратит значение **NULL**. Часто возникает ситуация, когда документ изменяет некоторые данные и требуется, чтобы его представления выполнили соответству-

ющие изменения. Упомянутую задачу реализует метод, весьма удачно названный **UpdateAllViews()**.

- *Информация о документе.* Получить данные о том, к какому файлу присоединен документ, позволяют два метода: **GetPathName()** и **GetTitle()**. **GetPathName()** возвращает полностью определенный путь документа, а **GetTitle()** — заголовок документа, в основу которого обычно положено имя его файла. Если документ ни разу не сохранялся, эти функции возвращают **NULL**.
- *Состояние модификации документа.* Класс **CDocument** содержит две функции, которые обеспечивают возможность отслеживать то, подвергался ли документ изменениям с момента последнего его сохранения. Если данные менялись, функция **IsModified()** возвращает ненулевое значение. Однако для корректного функционирования этого механизма каждая функция, вносящая изменения в документ, должна взаимодействовать с функцией **SetModifiedFlag()**, вызывая ее в процессе обновления данных.

Класс CSDIOneDoc

Класс **CSDIOneDoc** представляет собой минимальную реализацию класса документов. Трудно сделать так, чтобы его размеры были еще меньше, и в то же время он продолжал работать. Как и любой другой класс документов, **CSDIOne** содержит две части: переменные, в которых запоминается информация о документе, и методы, манипулирующие переменными и обеспечивающие к ним доступ.

Важный принцип объектно-ориентированного проектирования — инкапсуляция — утверждает, что переменные должны быть **private** (приватные) или **protected** (защищенные), но никогда **public** (общедоступные). К сожалению, этот принцип в MFC обычно игнорируется; например, MFC определяет все данные диалога ClassWizard как **public**. Даже обучающая программа MFC Scribble объявляет свои данные как **public**. Однако вы не должны следовать плохим примерам MFC — в классе CSDIOneDoc как **private** должна быть объявлена только единственная переменная **m_data**.

Переменная **m_data** является объектом **CArray**. Класс **CArray** — это класс шаблона, следовательно, определение класса записывается во время компиляции на основе аргументов, значения которых задаются во время инициализации шаблона. Объявление переменной **m_data**

```
private:
      CArray<CPoint,CPoint> m_data;
```

создает приватный, с изменяемыми размерами массив объектов **CPoint**.

Поскольку переменная **m_data** является приватной, в связи с чем доступ со стороны других классов невозможен, класс **CSDIOneDoc** реализует три операции, которые другие классы могут использовать для внесения изменений в документ. Добавление в документ нового объекта **CPoint** осуществляется при помощи вызова его функции **AddPoint()**. Определение количества **CPoint**, содержащихся в документе выполняется через обращение к его функции **NumPoints()**. Кроме того, можно выбрать конкретный объект **CPoint**, вызвав функцию **GetPoint()** и передав ей индекс. На первый взгляд может показаться, что объявление переменной **m_data** как **private** и использование функций *доступа* и *модификации* не намного лучше, чем просто объявление этой переменной как **public**. Однако при таком подходе имеет место двойная выгода:

- Появляется возможность изменять внутреннее представление **m_data**, внеся изменения только в реализацию функций **AddPoint()** и **GetPoint()**. Поскольку интерфейс **CSDIOne** остается прежним, нет необходимости корректировать другие классы, которые зависят от него.
- Пропадает головная боль, связанная с неограниченным доступом к переменным **public**. Поскольку обеспечиваются только функции, работающие с **m_data**, устранение возможных ошибок существенно упрощается. Если разрешить доступ **public**, придется просмотреть все приложение в поисках ссылок (прямых и косвенных) на переменную **m_data**.

Класс представлений

Последним солистом квартета DocView является класс представлений. Как нетрудно заметить на рис. 12.1, класс представлений по умолчанию — это подкласс класса **CView**, который, в свою очередь, представляет собой подкласс класса **CWnd**. На рис. 12.1 класс представлений выступает под именем **CYourView**. В программе SDIOne система MFC наделила аналогичный класс именем **CSDIOneView** и сохранила его в файлах SDIOneView.h и SDIOneView.cpp.

Разрабатывая приложение SDI, в основу класса приложений можно положить класс, отличный от **CView**. Например, в программе NotePod, которая создавалась в главе 1, в качестве базового класса представлений использовался **CEditView**. Как вы помните, класс представлений выбирался в AppWizard. В SDI-приложении диалоговое окно AppWizard Step 6 позволяет выделить каждый отдельный класс. При выделении класса представлений приложения поле вывода базового класса заменяется списком, из которого можно выбрать другой вид представления (рис. 12.3).

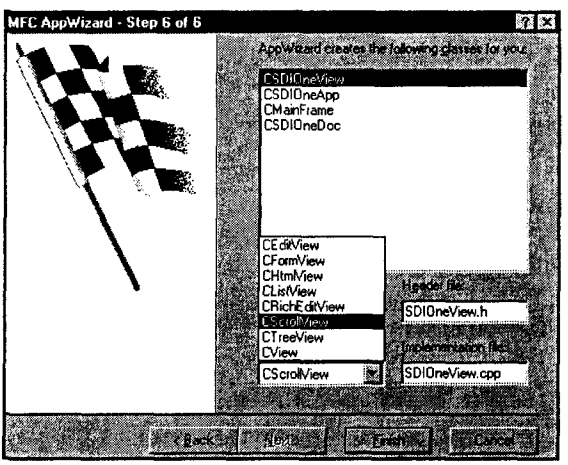


РИСУНОК 12.3.
Выбор другого базового класса для класса представлений в AppWizard

Класс CView

Как и класс документов, класс представлений, создаваемый AppWizard, необходимо настроить. **CView** представляет собой простой класс окон без обрамления. Во время запуска SDI-приложения окно представлений настраивает свои размеры таким образом, чтобы оно помещалось внутри клиентской области главного окна и покрывало ее.

Поскольку окно класса представлений не имеет обрамления, далеко не очевидно, что клиентская область рамочного окна оказалась перекрытой таким образом. Тем не менее, это становится очевидным в процессе взаимодействия с окном. Сообщения мыши пересылаются в класс представлений, а не в рамочное окно, так что рисование на поверхности клиентской области рамочного окна не приводит к каким-то ощутимым результатам — вывод полностью заслоняется окном представления.

Как и класс **CDocument**, **CView** имеет защищенный конструктор, виртуальный деструктор и методы диагностики. Кроме того, **CView** содержит метод **OnDraw()**, роль которого подобна функции **OnPaint()** в простом приложении, не использующем архитектуру DocView. Метод **OnDraw()** является тем местом, где выполняется визуализация документа. В главе 11, в классе **CSDIOneView** этот метод был перекрыт; именно так ведут себя классы представлений в большинстве программ DocView.

В дополнение к **OnDraw()** класс **CSDIOneView** содержит три метода, имеющих отношение к печати. Функция **OnDraw()** визуализирует печатный образ так же, как это делается для экранных образов. Три функции печати — **OnPreparePrinting()**, **OnBeginPrinting()** и **OnEndPrinting()** — позволяют выполнять операции настройки, необходимые при переходе от большого экрана к печатной странице, к которым относятся масштабирование и разбивка на страницы.

И наконец, класс **CSDIOneView** содержит метод **GetDocument()**, возвращающий указатель на объект **CDocument**, должным образом приведенный к указателю **CSDIOneDoc**. При сборке окончательной версии программы MFC подставляет эту функцию, тем самым повышая эффективность, но в то же время ограничивая проверку ошибок во время выполнения.

Права и ответственность

Как можно было убедиться, добавление новых классов в смесь не вносит дополнительных сложностей. К сожалению, нередко случается, что задача, которая *должна* работать — как, например, рисование в клиентской области окна — на самом деле *не работает*. Безупречное программирование предполагает постоянный учет отношений между каждым классом DocView и возлагаемыми на него задачами.

Рис. 12.4 показаны четыре класса, включенные в программу DocView, вместе с различными коммуникационными путями. (Здесь не отображена полная картина, в чем вы сможете убедиться, когда в двух следующих главах речь пойдет о меню, панелях инструментов и маршрутизации команд. В то же время рисунок может послужить хорошим начальным приближением.)

Как и в случае не-DocView-приложений, класс приложения DocView выбирает сообщения из очереди сообщений Windows и направляет их в объекты окон. Это происходит в методе **Run()** каркаса приложения, который инкапсулирует стандартный цикл ожидания сообщений Windows. Когда приложение выполняет диспетчеризацию сообщений, большая часть сообщений поступает в главное окно, которое направляет их в другие объекты для обработки. Однако сообщения мыши и клавиатуры направляются в окно представлений.

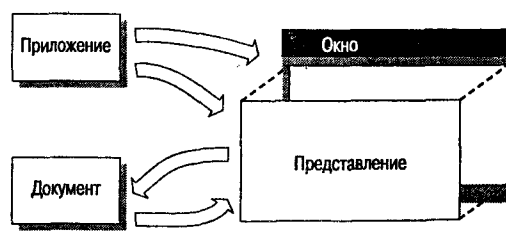


РИСУНОК 12.4. *Архитектура DocView SDI*

Наряду со стандартным путем через цикл ожидания сообщений (путь, которым может воспользоваться любой объект для связи с другим объектом), классы документов и представлений устанавливают прямую двустороннюю "горячую линию". Когда между документом и представлением установлена связь, каждый из них содержит указатель на другой. Таким образом, документ может обратиться к своему представлению с запросом перерисовать себя за счет вызова функции **OnUpdate()** представления. Аналогично, класс представлений напрямую связывается с объектом документа, чтобы получить данные, необходимые ему для визуализации.

Теперь, когда вы имеете общее представление о том, как выглядит приложение DocView, рассмотрим более подробно один из классов, присутствующих в приложении SDIOne. Поскольку эти классы относительно велики даже для такого простого приложения, как SDIOne, их листинги полностью приводиться не будут. Будет показан только код, имеющий непосредственное отношение к рассматриваемому вопросу.

Начнем с класса приложений **CSDIOneApp**.

Класс CSDIOneApp: вы называете *это* InitInstance()?

MFC присвоила классу приложения SDIOne имя **CSDIOneApp** и поместила его объявление в файл SDIOne.h, а реализацию — в SDIOne.cpp. **CSDIOneApp** содержит три перекрытых метода: конструктор **CSDIOneApp()**, виртуальную функцию **InitInstance()** и функцию **OnAppAbout()**, которая отображает диалоговое окно About приложения.

Файл реализации SDIOne.cpp также определяет глобальный объект **CSDIOneApp**. Когда объект приложения создается в глобальной области видимости, его конструктор вызывается до вызова функции **WinMain()** приложения. В силу такого раннего создания, SDIOne не может инициализировать свой класс приложения в конструкторе **CSDIOneApp**. Вместо этого он инициализируется в функции **InitInstance()**, что и показано на листинге 12.1.

Листинг 12.1. Метод CSDIOneApp::InitInstance().

```
////////////////////////////////////////////////////////////
// Инициализация CSDIOneApp
BOOL CSDIOneApp::InitInstance()
{
      // 1. Раздел начальной установки
      Enable3dControls();              // Выполнить этот вызов, если
                                       // применяется версия MFC с
                                       // совместно используемой DLL
      SetRegistryKey(_T("Local AppWizard-Generated Applications"));
      LoadStdProfileSettings();        // Загрузить из INI-файла
                                       // стандартные установки
      // 2. Объединение фрагментов в единое целое
      CSingleDocTemplate* pDocTemplate;
      pDocTemplate = new CSingleDocTemplate(
            IDR_MAINFRAME,
            RUNTIME_CLASS(CSDIOneDoc),
            RUNTIME_CLASS(CMainFrame),// главное рамочное окно SDI
```

```
        RUNTIME_CLASS(CSDIOneView));
    AddDocTemplate(pDocTemplate);
    // 3. Соединение с Windows Explorer
    EnableShellOpen();
    RegisterShellFileTypes(TRUE);
    // 4. Получение и обработка параметров командной строки
    CCommandLineInfo cmdInfo;
    ParseCommandLine(cmdInfo);
    if (!ProcessShellCommand(cmdInfo))
        return FALSE;
    // 5. Работа с окном
    m_pMainWnd->ShowWindow(SW_SHOW);
    m_pMainWnd->UpdateWindow();
    m_pMainWnd->DragAcceptFiles();
    return TRUE;
}
```

Если вспомнить метод **InitInstance()** минимальной программы MFC из главы 1, может поразить то, что показано здесь — неизменными сохранились только некоторые фрагменты. Однако различия только кажущиеся. Минимальный метод **InitInstance()** требует всего три строки, чтобы построить и отобразить новое окно. Метод **CSDIOneApp::InitInstance()** выполняет несколько дополнительных операций.

В листинге 12.1 метод **InitInstance()** разбит на пять разделов, каждый из которых выполняет конкретную задачу. Рассмотрим их по очереди.

Установка приложения

Раздел установки метода **InitInstance()** содержит три строки:

```
Enable3dControls();             // Выполнить этот вызов, если
                                // применяется версия MFC с совместно
                                // используемой DLL
SetRegistryKey(_T("Local AppWizard-Generated Applications"));
LoadStdProfileSettings();       // Загрузить из INI-файла стандартные
                                // установки
```

AppWizard добавляет эти строки в ответ на выбор, осуществленный в процессе генерации исходной программы. Строка **Enable3dControls()** помещается по причине отметки флажка 3D Controls в диалоговом окне MFC AppWizard Step 4 (см. рис. 12.5).

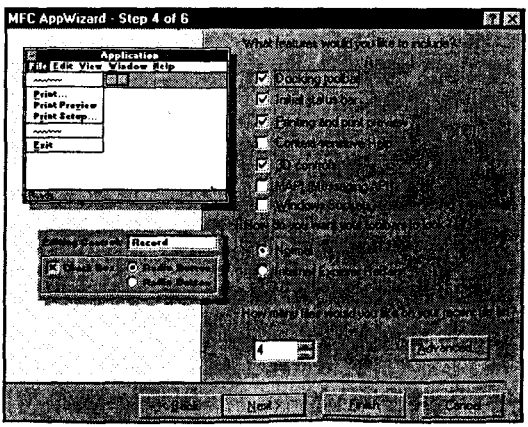


РИСУНОК 12.5.
Диалоговое окно MFC AppWizard Step 4.

Фактически две строки кода записаны на основе вашего выбора. Если в программе предусматривается динамическое связывание, вызывается функция, показанная выше; если же статистическое связывание — вызывается функция **Enable3dControlsStatic()**.

Enable3dControls() создает эффект трехмерности (как в Microsoft Excel) для элементов управления, используемых в диалоговых окнах и панелях инструментов. Вызов **Enable3dControls()** приводит к загрузке библиотеки динамической компоновки Ctrl3D32.dll, которая автоматически переводит диалоговый класс (а также и другие классы) на уровень подкласса. Как ни странно, даже если вызов функции **Enable3dControls()** завершится неудачей, диалоговые окна все равно обладают эффектом трехмерности при работе в Windows 95, Windows 98, или NT 4.0; плоский двумерный стиль будет получен только при работе в Windows NT 3.5.

Enable3dControls() не оказывает воздействия на элементы управления, созданные программным путем и отображаемые за пределами диалоговых окон. Такие элементы управления по умолчанию принимают плоский вид, если только при их создании не воспользоваться функцией **CreateEx()** и атрибутом **WS_EX_CLIENTEDGE**.

Регистрация при помощи мистера _T()

Две следующих строки раздела начальной установки активизируют *список MRU*. MRU — суть сокращение от *M*ost *R*ecently *U*sed (Использованные недавно); в этом списке содержатся имена файлов, открытые последними в приложении. По умолчанию список MRU содержит четыре файла. Если вы не хотите работать со списком MRU, установите число в нижней части диалогового окна AppWizard Step 4 равным 0 (данное значение лежит в пределах от 0 до 16). MFC использует это число при вызове функции **LoadStdProfileSettings()**. В случае установки числа равным 16 AppWizard сгенерирует строку

```
LoadStdProfileSettings(16);
```

При необходимости эту строку можно изменить вручную. Напоминаем, что значение по умолчанию равно 4. Благодаря вызову **LoadStdProfileSettings()**, Windows будет сохранять имена последних открытых файлов вместе с их путями. Эта информация появляется в меню File, как показано на рис. 12.6.

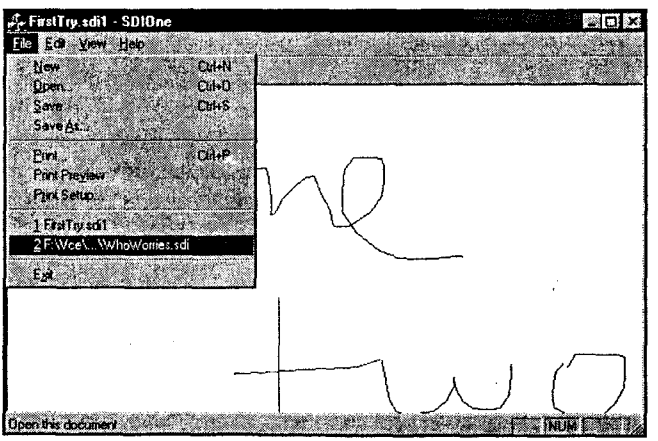


РИСУНОК 12.6.

Отображение списка файлов MRU в приложении SDIOne.

Поскольку Windows помнит, какой файл открывался последним даже после завершения и повторного запуска приложения, очевидно, что информация об этом где-то должна храниться. Этим где-то является ни что иное как Windows Registry (Системный реестр).

Системный реестр представляет собой двоичный файл, в котором хранится великое разнообразие конфигурационной информации в Windows NT, Windows 95 и Windows 98.

Информация, сохраняемая в системном реестре, организована в виде иерархической базы данных, во многом напоминающей привычную файловую систему. Однако вместо папок используются *параметры*, а вместо файлов — *значения*. Вызов функции

```
SetRegistryKey(_T("Local AppWizard-Generated Applications"));
```

заставляет Windows создать параметр (пусть *папку*, если вам так нравится) с громоздким именем HKEY_CURRENT_USER\Software\Local AppWizard-Generated Applications\SDIOne. Впечатляет, не так ли? Этот параметр содержит два подпараметра: Recent File List (Список последних открытых файлов) и Settings (Установки).

При вызове в функцию **SetRegistryKey()** в качестве параметра передается строка, определяющая параметр *семейства*, но не параметр конкретного приложения. Семейство могут составлять несколько приложений, при этом каждое имеет свой параметр приложения и свои подпараметры. Если изменить текущий аргумент с **_T("Local AppWizard-Generated Applications")** на **_T("Steve 'n' Bill's Scintillating Software")**, список MRU будет сохраняться именно в новом параметре (см. рис. 12.7).

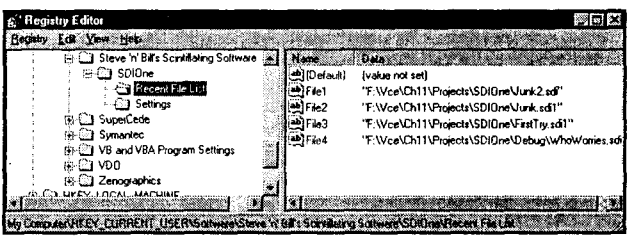


РИСУНОК 12.7.
Просмотр в реестре параметров программы SDIOne с помощью редактора реестра Regedit.

Если не вызывать **SetRegistryKey()**, то обращение к **LoadStdProfileSettings()** заставит Windows создать INI-файл с именем SDIOne.ini в подкаталоге Windows и записать в него список MRU. Если работа с системным реестром Windows не нравится, просто удалите вызов **SetRegistryKey()**.

И наконец, вас, возможно, заинтересует, что такое **_T()**. Программы, написанные для продажи на международном рынке, уже не используют однобайтный набор символов ASCII в строках и символах, поскольку обеспечиваемых им 256 символов для многих языков мира оказывается недостаточно. Windows поддерживает два международных набора символов: MBCS (Multi-Byte Character Set — Многобайтный набор) и Unicode, универсальное 16-разрядное кодирование. К сожалению, Unicode реализован только в Windows NT, но не в Windows 95 или Windows 98.

MFC поставляется со специальным файлом заголовков TCHAR.h, который определяет набор макрокоманд для прозрачного преобразования строк и символов между различными наборами символов Windows. Для выполнения преобразования необходимо поместить все символические константы в макрокоманду **_TCHAR()**, например, так: `_TCHAR('A')`.

Все строковые константы следует поместить в макрокоманду **_TEXT()** или **_T()**. После определения в программе константы **_UNICODE** символы и константы будут корректно представляться как символы Unicode. В противном случае они останутся обычными однобайтовыми символами.

К сожалению, написание программного обеспечения для международного рынка — дело значительно более трудное и сложное, чем помещение в код нескольких макрокоманд. С другой стороны, они не ничего не портят, и если их использовать, то ваша программа совершит еще один шаг на пути к всемирному успеху.

Что представляют собой эти расширения?

Если вы внимательно ознакомитесь с рис. 12.7, то заметите, что некоторые файлы имеют расширение .sdi1, в то время как другие — расширение .sdi. Поскольку в диалоговом окне Advanced Options в AppWizard .sdi1 было определено в качестве расширения файлов приложения, по умолчанию в диалоговом окне File Open отображаются только файлы с расширением .sdi1. Однако при сохранении вновь созданного рисунка без явного указания расширения Windows добавит расширение самостоятельно, но только из трех символов. По всей видимости, цель этого заключается в обеспечении совместимости с более ранними операционными системами.

Соединение составных частей

Теперь, когда все промежуточные препятствия устранены, настало время создания составных частей приложения: рамки, представления и документа. Памятуйте о том, что приложение DocView крепко связывает каждую составную часть с остальными. Чтобы классы представлений и документов поддерживали двусторонний обмен данными, каждый из них должен корректно указывать на другой. Дабы не заставлять вас делать эту тонкую, точную и утомительную инициализацию, MFC применяет вспомогательный класс, именуемый *шаблоном документа.*

MFC предлагает два класса шаблонов документа: **CSingleDocTemplate**, предназначенный для соединения представления, рамочного окна и документа в специальную конструкцию SDI, и **CMultiDocTemplate**, выполняющий то же самое для MDI-приложений.

Использование шаблона документа представляет собой трехшаговый процесс:

- Создать указатель на шаблон документа.
- Создать новый шаблон документа за счет использования оператора **new** и ссылок на экземпляры классов документа, представления и рамочного окна.
- Добавить шаблон документа в объект приложения, чтобы последний смог управлять всеми объектами.

Рассмотрим, как это работает на практике.

Создание указателя на шаблон документа не вызывает особых трудностей. Для приложения SDIOne соответствующая строка имеет вид:

```
CSingleDocTemplate* pDocTemplate;
```

Последний шаг — добавление шаблона документа в приложение — простая операция, которая сводится к вызову функции **AddDocTemplate()**:

```
AddDocTemplate(pDocTemplate);
```

Второй шаг выглядит достаточно запутанным. Ниже показан оператор, создающий новый шаблон документов для приложения SDIOne:

```
pDocTemplate = new CSingleDocTemplate(
      IDR_MAINFRAME,
      RUNTIME_CLASS(CSDIOneDoc),
      RUNTIME_CLASS(CMainFrame),     // главное рамочное окно SDI
      RUNTIME_CLASS(CSDIOneView));
```

Для конструктора класса **CSingleDocTemplate** требуется четыре аргумента: целочисленное представление идентификатора ресурса и три указателя **CRuntimeClass**.

Идентификатор ресурса — это многоцелевой идентификатор, используемый каркасом для распознания пиктограммы приложения, главного меню, таблицы акселераторов и строки документа, созданной на основе информации, введенной в диалоговом окне Advanced Options в AppWizard. Во время выполнения этот идентификатор используется для определения местоположения и установки каждого из перечисленных элементов. В конечном итоге эта часть вполне прояснилась. Тем не менее, использование указателей **CRuntimeClass** и макрокоманды **RUNTIME_CLASS()** несколько отличается.

При передаче в макрокоманду **RUNTIME_CLASS()** имени класса, например, **CSDIOneDoc**, она возвращает указатель, который можно использовать для создания экземпляра этого класса при помощи процесса *динамического создания объектов*. Классы, предназначенные для динамического построения объектов, содержат статический указатель на структуру **CRuntimeClass**, описывающую класс, и две функции: **GetRuntimeClass()** и **CreateObject()**. MFC будет помещать такую структуру в ваши классы, когда в файле заголовков и в файле реализации присутствуют макрокоманды **DECLARE_DYNAMIC()** и **IMPLEMENT_DYNAMIC()**. При условии соответствия класса всем требованиям, экземпляр можно создать, вызвав функцию **CreateObject()** вместо оператора **new**.

Самое время задать себе недоуменный вопрос: "Какое это имеет отношение ко мне?" К счастью, ответом будет: "Никакого". Вы редко, если вообще когда-либо, будете создавать динамические объекты. Как правило, вам даже не придется вносить изменения в код, сгенерированный AppWizard, чтобы образовать документ, представление и рамочное окно. Однако полезно знать, что MFC всегда находится за кулисами, особенно в тех случаях, когда программа ведет себя некорректно или терпит крах.

Игры с оболочками

Третий раздел метода **InitInstance()** класса **CSDIOneApp** обеспечивает связь вашей программы с оболочкой Windows. (Программа Windows Explorer называется *оболочкой* ввиду того, что она обеспечивает пользователю первичный интерфейс с операционной системой.) Возможно, вы уже заметили, что когда дважды щелкнуть на DOC-файле в Windows Explorer, автоматически открывается Microsoft Word и загружается документ, на котором выполнялся двойной щелчок. Добавляя в программу строки

```
EnableShellOpen();
RegisterShellFileTypes(TRUE);
```

это делается возможным и для вашего приложения. Именно благодаря наличию в программе этих строк, можно дважды щелкнуть на файле с расширением .sdi1 и тем самым открыть программу SDIOne. Более того, если перетащить файл с расширением .sdi1 на пиктограмму принтера, этот файл будет распечатан, как если бы была выбрана опция Print из меню File в SDIOne.

Командная строка

Четвертый раздел метода **InitInstance()** обрабатывает командную строку программы, выделяя аргументы, переданные в программу при запуске. Наверное, небезынтересно узнать, что именно делает здесь этот программный код. В конце концов, при столь длительном существовании механизма перетаскивания, только немногие из нас используют аргументы командной строки.

Все это, конечно же, правильно. Однако даже если вы вообще не передаете в программы аргументы командной строки, Windows может сделать это без вашего ведома. Например, раздел панели управления, относящийся к хранителю экрана обеспечивает передачу ключа в приложение, сообщающего, в каком режиме оно должно выполняться — как хранитель экрана (**s**, **/s** или **-s**) либо в режиме конфигурирования (**c**, **/c**, или **-c**). Нечто подобное делают программы DocView.

В DocView-программах MFC разбивает командную строку на части за счет создания объекта **CCommandLineInfo** и передачи его в функцию **ParseCommandLine()**. Полностью разобранная командная строка передается в **ProcessShellCommand()**.

Функция **ProcessShellCommand()** исследует объект **CCommandLineInfo**, чтобы определить, является какой-либо из аргументов именем файла. Если это так, то **ProcessShellCommand()** предпринимает попытку загрузить документ. Если ни одно имя файла не указано, **ProcessShellCommand()** пытается запустить приложение с пустым документом. Если функция **ProcessShellCommand()** не сумеет решить ни одной из перечисленных задач, она возвращает значение **FALSE** и выполнение вашей программы завершается.

Работа с окнами

Последний раздел метода **InitInstance()** должен показаться знакомым. Две строки

```
m_pMainWnd->ShowWindow(SW_SHOW);
m_pMainWnd->UpdateWindow();
```

идентичны строкам, использованным для отображения главного окна минимального MFC-приложения в главе 2. Последняя строка,

```
m_pMainWnd->DragAcceptFiles();
```

активизирует поддержку механизма перетаскивания. Разумеется, это отнюдь не все для реализации перетаскивания, однако все, что следует знать — так это то, что каркас обеспечивает вызов корректных методов, освобождая вас по крайней мере от этой работы.

Postscript и предварительный просмотр: карта сообщений для CSDIOneApp

В главе 4 вы имели дело с таблицей реакции, или *картой сообщений,* для приложения FourUp. Вы, конечно, помните, что карта сообщений — это сложный набор макрокоманд, предназначенных для маршрутизации сообщений между соответствующими методами.

Одной из наиболее распространенных макрокоманд карты сообщений является **ON_COMMAND()**, реагирующая на выбор элементов меню или панели инст-

рументов, которые генерируют сообщения **WM_COMMAND()**. Поскольку каждая команда меню генерирует одно и то же сообщение, MFC необходим способ сопоставления конкретной команды меню с функцией, обеспечивающей обработку этой команды.

Обычно уведомление MFC о том, что данная функция обрабатывает ту или иную команду меню, осуществляется с использованием макрокоманды **ON_COMMAND()**, которая получает аргументы, определяющие идентификатор ресурса меню и имя функции обработки. Как видно в листинге 12.2, каркас DocView предусмотрительно связал обработчики **ON_COMMAND()** с командами File|New, File|Open и File|Print Setup.

Листинг 12.2. Карта сообщений класса CSDIOneApp.

```
////////////////////////////////////////////////////////////////////////////
    /////
// CSDIOneApp
BEGIN_MESSAGE_MAP(CSDIOneApp, CWinApp)
      //{{AFX_MSG_MAP(CSDIOneApp)
      ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
      //}}AFX_MSG_MAP
      // Стандартные команды для документов, основанных на файлах
      ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
      ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
      // Стандартная команда настройки печати
      ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()
```

В дополнение к командам меню, автоматически подключаемым AppWizard, сам каркас содержит несколько "магических" идентификаторов, инициирующих то или иное действие. Например, если присвоить некоторому элементу меню идентификатор **ID_APP_EXIT ID**, то выбор этого элемента меню заставляет MFC вызывать функцию **OnAppExit()**, принадлежащую **CWinApp**. В число подобных "автоматических" идентификаторов входят **ID_FILE_SAVE** и **ID_FILE_SAVE_AS**.

Что еще есть в меню?

Изучая карту сообщений для **CSDIOneApp**, можно заметить, что карта содержит команды настройки печати, но не содержит команды собственно печати или предварительного просмотра печати. Почему?

Причина заключается в механизме маршрутизации команд. Имея такой механизм, не потребуется обрабатывать все команды меню в классе приложения, рамочного окна или представления — появляется возможность обработать их там, где это проще или целесообразней всего сделать. Поскольку весь код рисования находится в классе **CSDIOneView**, печать удобно выполнять именно в этом классе. Разумеется, что при исследовании карты сообщений для **CSDIOneView** несложно обнаружить обработчики команд предварительного просмотра печати и собственно печати.

Неплохо, если все эти разговоры о меню породили желание дальнейшего накопления знаний. Почему? Да потому что следующей темой будет ... опять-таки меню!

Глава 13

Мечта Пикассо: программа MiniSketch

Программы, управляемые меню, представляют собой воплощение романа Джозефа Хеллера (Joseph Heller) "Ловушка-22" ("Catch-22") — визуальный эквивалент хаоса речевой почты.

Если вы работали с ранними компьютеризированными системами бухгалтерского учета или системами резервирования авиабилетов, то понимаете, что в системах *на основе меню* управление осуществляет программа, но никак не вы. И очень часто программа поворачивает совсем не туда, куда требуется.

Windows-программы не управляются при помощи меню. Разумеется, меню составляют значительную часть графического пользовательского интерфейса Windows (GUI). В те далекие дни, когда компьютеры были железными, а программисты — "настоящими мужчинами", GUI иногда называли интерфейсом WIMP, ибо он опирался на Windows, Icons (пиктограммы), Mice (мыши) и Pull-down menus (выпадающие меню). Разумеется, в наши дни даже самые крайние недоброжелатели предпочитают мышь перед ручным вводом операций.

Однако Windows-программы в большей степени *управляются пользователем*, нежели меню. В программах, управляемых пользователем, последний определяет требуемый порядок выполнения действий. Программы, управляемые меню, были популярны в течение столь продолжительного времени по причине их более простой реализации по сравнению с программами, управляемыми событиями.

К счастью, как только сделан переход от иерархического, процедурного программирования к объектно-ориентированному программированию, управляемому событиями, создание меню, которые вовлекают пользователя в процессе управления, превращается в "конфетку". А имея в распоряжении инструментальные средства, обеспечиваемые Visual C++, сделать это намного легче.

Из данной главы станет известно, как создавать различные типы меню и как обрабатывать генерируемые ими сообщения. В следующей главе мы продолжим эту тему и расширим ваши знания о меню, рассмотрев принципы построения панелей инструментов Windows. Но перед этим потребуется решить некоторые из старых проблем.

Какую информацию несет в себе имя?

Теперь, когда программа SDIOne отслужила свое, перейдем к программе SDITwo, в которую будут добавлены меню и панели инструментов. Прежде всего, необходимо сделать одну вещь — *выполнить всю процедуру перед тем, как приступить к делу.*

Создайте новый каталог с именем SDITwo и скопируйте в него файлы из каталога SDIOne. Теперь замените в именах файлов строку SDIOne на SDITwo, оставляя расширения файлов нетронутыми. Ого..., файлов многовато, не так ли? Откройте каждый файл в редакторе и замените все вхождения строки "SDIOne" на "SDITwo". Откройте файл ресурсов и поменяйте все его ссылки. Ух..., *ужасно* много пришлось сделать изменений... А некоторые из этих файлов оказались двоичными, не так ли? Ну что ж, если у вас имеется шестнадцатиричный редактор, вы, по всей вероятности, им уже воспользовались, а если нет — вспомните о программе, называемой debug...

Вот такая имеется обстановка. Дело отнюдь не в отсутствии возможности изменения имени проекта после того, как AppWizard завершил свою работу. Все дело в отсутствии именно *практического* способа.

Итак, на выбор предлагается:

- Каждый раз, когда в программу добавляется новое средство, начинать новый проект под другим именем и повторно проделать все шаги из предыдущего проекта. Можете считать это "набиванием руки", если вам так нравится.

- По мере роста и совершенствования программы, не изменять ее имя, при этом вы будете постоянно получать напоминание о необходимости планирования наперед. Хм. Что-то подобное 6-символьному полю для представления даты или, скажем, ограничению памяти пределом в 640 К.
- Решить проблему сейчас и выбрать имя, которое окажется достаточным пусть не в течение ближайшего тысячелетия, но по крайней мере на протяжении нескольких последующих глав. В этом случае придется повторно ввести код программы SDIOne, однако только один раз.

Как вы, возможно, уже догадались, мы собираемся выбрать дверь под номером три и повторно ввести код программы SDIOne. Разумеется, *вы* можете избрать намного более безопасный путь: просто скопировать требуемые файлы из сопровождающего CD-ROM. Однако если уж вы хотите все сделать самостоятельно, сейчас мы предложим шаги для построения программы MegaloMatic (Мания величия). Нет ..., подождите, название что-то не соответствует назначению программы. А как насчет StupendousPainter (Непревзойденный художник)? Не подойдет, слишком претенциозно. А может быть, ScintillatingSketcher (Блестящий специалист по эскизам)? Вряд ли, трудно произносить и к тому же долго набирать на клавиатуре.

Нам требуется простое и содержательное имя; пусть им будет MiniSketch (Мини-эскиз).

Ниже приводятся инструкции по построению программы MiniSketch. За более пространными объяснениями вернитесь в главу 11 — за исключением нескольких опций, по сравнению с программой SDIOne изменились только имена.

1. Создайте новый проект MFC AppWizard (exe) и назовите его MiniSketch.
2. В диалоговом окне Step 1 выберите Single Document и отметьте флажок Document/View support. В диалоговом окне Step 2 выберите None и щелкните на Next. В диалоговом окне Step 3 сбросьте флажок ActiveX Controls и щелкните на Next. В диалоговом окне Step 4 примите значения по умолчанию, после чего щелкните на Advanced, чтобы открыть диалоговое окно Advanced Options.
3. Диалоговое окно Advanced Options для программы MiniSketch показано на рис. 13.1. Будет использоваться трехсимвольное расширение файлов .msk. Приведите каждое поле в соответствие этому рисунку. По завершении щелкните на Close, а затем на Next.
4. Примите значения по умолчанию, предлагаемые в диалоговом окне Step 5. В диалоговом окне Step 6 потребуется изменить имена классов для уменьшения их длины. Давайте воспользуемся аббревиатурой "MS" для обозначения MiniSketch. Таким образом, класс получит имя **CMSView**. Измените соответствующим образом имена классов в окне редактирования Class Name. На рис. 13.2 показано изменение имени класса представления. В качестве имени класса рамочного окна оставьте **CMainFrame**. Не меняйте имена ни одного из файлов — измените *только* имена трех классов, которые должны выглядеть как **CMSView**, **CMSApp** и **CMSDoc**. После внесения всех изменений закройте AppWizard.
5. Добавьте в класс **CMSView** приватный элемент данных **CPoint** с именем **m_LineStart**. Определение класса **CMSDoc** дополните массивом **CPoint** с именем **m_data**, введя с клавиатуры следующие две строки:

```
private:
      CArray<CPoint, CPoint> m_data;
```

6. Выберите закладку FileView в окне Workspace и найдите файл StdAfx.h в закладке Header Files. Откройте этот файл и после всех операторов include добавьте строку:

```
#include <afxtempl.h>
```

   Эта строка гарантирует то, что классы шаблонов коллекций, которые будут использоваться в нескольких последующих главах, станут доступными для каждого класса, а не только для класса **CMSDoc**.

7. Щелкните правой кнопкой мыши на классе **CMSView** для доступа в контекстное меню. Добавьте в класс **CMSView** обработчики сообщений для **WM_LBUTTONDOWN**, **WM_LBUTTONUP** и **WM_MOUSEMOVE**, как показано в листинге 13.1.

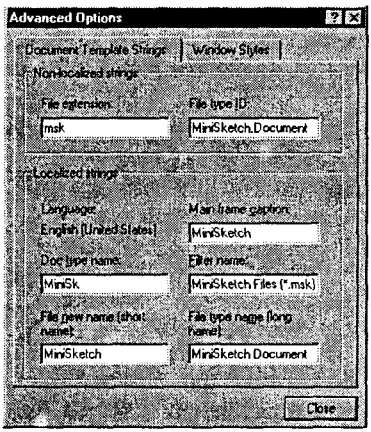


РИСУНОК 13.1. *Диалоговое окно Advanced Options для программы MiniSketch.*

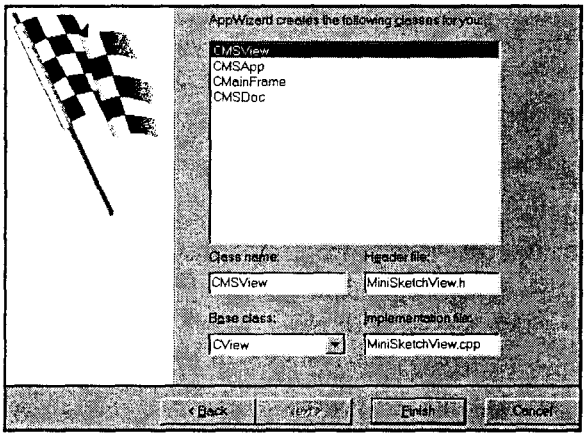


РИСУНОК 13.2. *Изменение имен классов программы MiniSketch.*

Листинг 13.1. Обработчики сообщений для класса CMSView.

```
void CMSView::OnLButtonDown(UINT nFlags, CPoint point)
{
      m_LineStart = point;
      SetCapture();
}

void CMSView::OnLButtonUp(UINT nFlags, CPoint point)
{
      ReleaseCapture();
}

void CMSView::OnMouseMove(UINT nFlags, CPoint point)
{
      if (nFlags & MK_LBUTTON)
      {
            CMSDoc* pDoc = GetDocument();
            ASSERT_VALID(pDoc);
            pDoc->AddPoint(m_LineStart);
```

```
            pDoc->AddPoint(point);
            CClientDC dc(this);
            dc.MoveTo(m_LineStart);
            dc.LineTo(point);
            m_LineStart = point;
        }
}
```

8. Перекройте виртуальную функцию **OnDraw()** в классе **CMSView**, поместив в нее код, выделенный в листинге 13.2.

Листинг 13.2. Виртуальная функция CMSView OnDraw().

```
void CMSView::OnDraw(CDC* pDC)
{
    CMSDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    int nPoints = pDoc->NumPoints();
    for (int index = 0; index < nPoints; index += 2)
    {
        pDC->MoveTo(pDoc->GetPoint(index));
        pDC->LineTo(pDoc->GetPoint(index+1));
    }
}
```

9. Добавьте в класс **CMSDoc** методы, представленные в листинге 13.3. Это можно сделать, вручную поместив код в конец файла MiniSketchDoc.cpp, либо воспользовавшись диалоговым окном Add Member Function. При ручном добавлении кода не забудьте поместить объявление каждой функции в определение класса **CMSDoc**, находящееся в файле MiniSketchDoc.h.

Листинг 13.3. Методы класса CMSDoc

```
int CMSDoc::NumPoints()
{
    return m_data.GetSize();
}

CPoint CMSDoc::GetPoint(int index)
{
    return m_data[index];
}

void CMSDoc::AddPoint(CPoint point)
{
    m_data.Add(point);
    SetModifiedFlag();
}
```

10. Перекройте функцию **CMSDoc::Serialize()**, заменив ее тело строкой

```
m_data.Serialize(ar);
```

Кроме того, добавьте в функцию **CMSDoc::OnNewDocument()** следующую строку, непосредственно перед оператором возврата:

```
m_data.SetSize(0, 128);
```

Откомпилируйте и запустите завершенную программу; она должна работать точно так же, как и SDIOne. Если возникнут проблемы, вспомните, что файлы можно загрузить из CD-ROM. *Сейчас* можно приступать к изменению программы.

MiniSketch получает меню

При переходе от PaintORama к SDIOne и MiniSketch оставим пока в покое такие усовершенствования, как цветные перья и кисти, а также возможность рисования форм и фигур. Причина частично связана с пользовательским интерфейсом. Поскольку PaintORama представляет собой диалоговое приложение, она использует элементы управления, подобные переключателям, полям со списком и просто спискам. В SDI-приложении нет *места* для таких свойств, поскольку "холст" занимает большую часть экрана.

Прибегая во время разработки программы MiniSketch к помощи интерфейса PaintORama, так сказать, поплывем по течению и научимся работать в стиле SDI — используя меню и панели инструментов. Хотя в общем и целом вы будете повторять процесс разработки программы PaintORama, вы скоро убедитесь в том, что на тропе SDI складывается совершенно другая обстановка.

Добавление элемента меню: очистка рисунка

Во время разработки PaintORama первым добавленным элементом управления была кнопка, обеспечивающая очистку рисунка. Нечто подобное сейчас проделается и для программы MiniSketch — добавится элемент меню Clear Drawing (Очистка рисунка), работающий точно так же, как и кнопка Clear в программе PaintORama.

"Минуточку", — скажете вы. "Приложение уже имеет кнопку Clear. Ведь при выборе команды File|New или выполнении щелчка на пиктограмме New File программа как раз и очищает рисунок."

В большинстве случаев это так, но не всегда. Выбор File|New приводит к созданию нового документа, но это вовсе не идентично стиранию содержимого существующего документа. Например, при попытке сохранения нового документа открывается диалоговое окно Save As, поскольку новый файл не имеет имени. В случае очистки содержимого документа и последующего выбора File|Save программа MiniSketch сохранит этот документ под существующим именем.

Щелкнув на элементе File|New меню MiniSketch, можно обнаружить еще одно различие: появляется предупреждающее диалоговое окно, показанное на рис. 13.3, предлагающее сохранить изменения. В отличие от SDIOne, программа MiniSketch знает о внесенных изменениях в рисунке (этот факт фиксирует метод **SetModifiedFlags()**, который вызывается классом **CMSDoc** при любой корректировке документа) и не позволит отказаться от изменений без соответствующего подтверждения.

Использования редактора меню

Windows сохраняет меню в виде ресурсов в файле RC, подобно шаблонам диалоговых окон. Шаблон меню описывает структуру всей системы меню и характеристики каждого ее элемента. При желании шаблон меню можно редактировать в ресурсном сценарии, однако что касается MiniSketch, лучше воспользоваться специализированным редактором меню (Menu Editor).

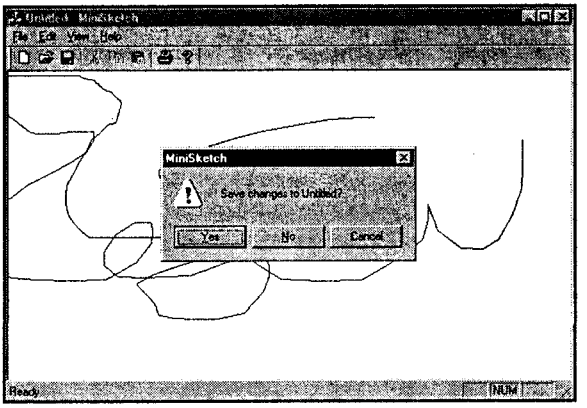


РИСУНОК 13.3.
Диалоговое окно подтверждения отказа от изменений, выполненных в документе.

Для запуска Menu Editor откройте проект MiniSketch и выберите закладку ResourceView в окне Workspace. Отыщите папку Menu и разверните ее. Вы обнаружите один ресурс меню с именем **IDR_MAINFRAME**. Дважды щелкните на **IDR_MAINFRAME** для открытия Menu Editor (см. рис. 13.4).

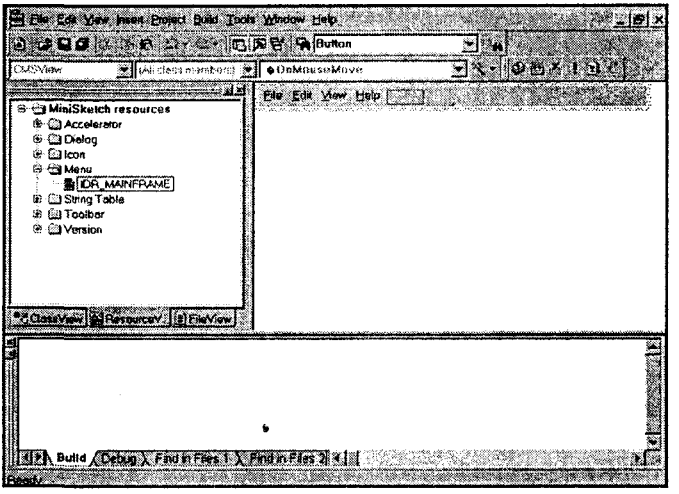


РИСУНОК 13.4.
Запуск Menu Editor.

ПРИМЕЧАНИЕ

MFC использует идентификатор ресурса IDR_MAINFRAME для главной пиктограммы приложения, панели инструментов и таблицы акселераторов. Один и тот же идентификатор можно использовать для нескольких ресурсов при условии, что они имеют различные типы. Когда класс шаблонов документов увязывает рамочное окно, документ и представления документа, он использует этот прием для поиска всех необходимых ресурсов.

Редактор меню визуально представляет структуру вашего главного меню. В верхней части отображаются заголовки выпадающих меню, которые появляются в строке меню во время выполнения программы. Справа от заголовков меню верхнего уровня находится очертание прямоугольника, показывающего место разме-

щения нового выпадающего меню. Вскоре вы этим займетесь; в настоящий же момент мы добавим в меню Edit команду Clear Drawing.

Для этого потребуется выполнить следующие действия:

1. Щелкните на заголовке Edit в Menu Editor, после чего на экране появится меню Edit. В нижней части меню Edit присутствует контурный прямоугольник; щелкните на нем, и пунктирная линия контура изменит свой цвет с черного на белый, при этом внутри контура будет присутствовать два ряда черных точек. Выберите элемент меню, щелкните на нем правой кнопкой мыши для открытия контекстного меню и выберите в контекстном меню команду Properties. Загружается диалоговое окно Menu Item Properties (Свойства элемента меню), как показано на рис. 13.5. (В случае выбора нового элемента меню, его заголовок можно просто ввести с клавиатуры. При этом диалоговое окно Menu Item Properties открывается автоматически.)

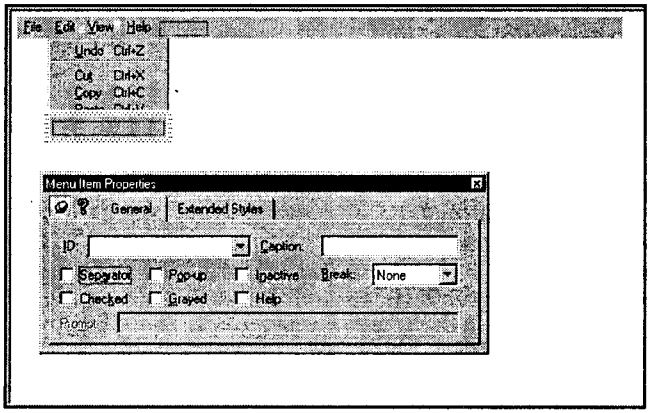


РИСУНОК 13.5.
Редактор меню и диалоговое окно Menu Item Properties.

2. В поле Caption введите "Clear &Drawing". (Амперсанд обеспечивает подчеркивание буквы "D", так что пользователи смогут выбрать команду меню, сначала нажав комбинацию Alt+E, а затем нажав D.)
3. В поле Prompt (Подсказка) введите строку "Erase the current drawing\nErase Drawing" (Стереть текущий рисунок/Стереть рисунок). Первая половина строки появляется в строке состояния во время перемещения по элементам меню при помощи мыши или клавиатуры, обеспечивая для пользователя поясняющую информацию. Вторая часть строки содержит текст подсказки. (Текст подсказки появляется над кнопкой панели управления, если эта кнопка имеет тот же идентификатор ресурса, что и выбранный элемент меню. Возможность отображения подсказки лучше всего всегда включать, даже если соответствующая кнопка в панели управления в данный момент не планируется. Впоследствии, при добавлении такой кнопки в панель управления, не придется лишний раз заниматься воспоминаниями.) Подсказка для элемента меню отделяется от подсказки для кнопки при помощи управляющей последовательности новой строки (\n).
4. Раскройте список идентификаторов. Вы увидите, что Menu Editor заблаговременно создал идентификатор с именем **ID_EDIT_CLEARDRAWING**, скомбинировав имена меню (Edit) и элемента меню. Если по каким-либо причинам предлагаемый идентификатор ресурса не устраивает, можно про-

сто ввести с клавиатуры свой собственный. В данном случае вполне подойдут значения, предлагаемые Visual C++. На рис. 13.6 показано заполненное диалоговое окно Menu Item Properties — постарайтесь привести свои данные в соответствие с этим рисунком.

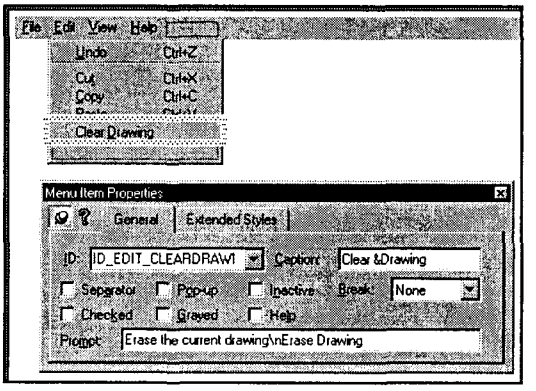


РИСУНОК 13.6.
Диалоговое окно Menu Item Properties для элемента меню Clear Drawing.

Добавление обработчика элемента меню

При добавлении нового элемента в шаблон меню путем ввода заголовка и идентификатора ресурса дополнительную работу выполнять не потребуется. Во время выполнения Windows отображает меню и обеспечивает взаимодействие с пользователем. Не нужно даже отслеживать работу мыши или клавиатуры — к сведению следует принять только факт выбора пользователем команды меню.

Когда пользователь выбирает элемент меню (посредством мыши или клавиатуры), Windows посылает программе сообщение **WM_COMMAND**. В этом сообщении передается идентификатор ресурса элемента меню, который сгенерировал сообщение.

Маршрутизация команд

В традиционном C-стиле Windows-программы Windows посылает сообщение **WM_COMMAND** в окно, содержащее меню. В рамках архитектуры DocView это создает проблемы. Класс рамочного окна (который содержит меню) не сможет выполнить обработку сообщения (поскольку это не класс документов) и должен будет передать его в класс, который способен это обеспечить. В результате сообщение дважды подвергается обработке. Например, чтобы выполнить обработку команды Clear Drawing (Удалить рисунок) в программе MiniSketch, необходимо реализовать обработчик внутри класса **CMainFrame**, отвечающий за прием сообщений от Windows, а затем написать другой обработчик в классе **CMSView**, который, собственно говоря, и выполняет основную работу.

MFC элегантно обходит упомянутое ограничение за счет поиска во всех классах приложения такого класса, который сможет обработать командное сообщение. Подобного рода подход, получивший название *маршрутизации команд*, работает только с сообщениями **WM_COMMAND** — с сообщениями, которые генерируются панелями инструментов и меню. С другими типами сообщений Windows он не работает.

Под управлением механизма маршрутизации команд сообщения **WM_COMMAND** сначала пересылаются в активное представление, затем в документ, после этого в объект главного рамочного окна, и в конце концов, в само

приложение. Если на этом пути какой-либо объект обработает сообщение, оно дальше не передается; если ни один объект не выполнит обработку сообщения, оно передается в Windows для обработки, принятой по умолчанию. Для командных сообщений приложения обработка по умолчанию означает (естественно) отсутствие каких-либо действий.

Использование Menu Editor

При помощи Menu Editor несложно связать обработчик действия или команды с конкретным элементом меню. Давайте поместим в программу обработчик команды Clear Drawing. Для этого потребуется выполнить следующие шаги:

1. Убедитесь в том, что проект MiniSketch загружен. Откройте меню **IDR_MAINFRAME** в Menu Editor и выберите элемент Edit|Clear Drawing. Щелкните на нем правой кнопкой мыши, а затем выберите ClassWizard из контекстного меню.
2. В окне ClassWizard выберите страницу свойств Message Maps. Идентификатор ресурса **ID_EDIT_CLEARDRAWING** должен появиться подсвеченным в списке Object IDs. Вы должны решить, какой класс должен выполнять обработку сообщения команды Clear Drawing. Поскольку эта команда затрагивает содержимое документа, поместим ее в класс документов. Выберите в списке Class Name элемент **CMSDoc**.
3. Сейчас необходимо решить, какой вид сообщений вы намерены обрабатывать. Как нетрудно видеть из списка Messages, можно выбрать либо **COMMAND**, либо **UPDATE_COMMAND_UI**. В данный момент вас интересуют только командные сообщения **WM_COMMAND**, поэтому выделите **COMMAND**. Сделайте так, чтобы ваш экран выглядел так, как рис. 13.7, а затем щелкните на Add Function.

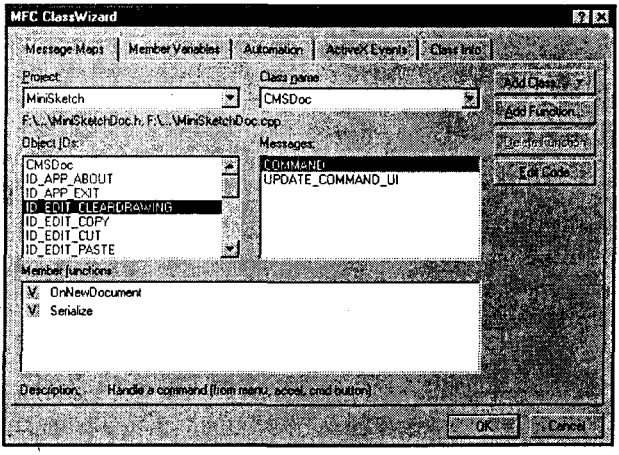


РИСУНОК 13.7.
Добавление обработчика команды Clear Drawing в ClassWizard

4. В диалоговом окне Add Member Function измените предложенное имя **OnEditCleardrawing** на **OnEditClearDrawing**, которое несколько легче читается. Щелкните на OK, после чего произойдет возврат в диалоговое окно ClassWizard. Щелкните на Edit Code (Редактировать код) и вы перейдете в текстовый редактор.

5. В текстовом редакторе введите код, выделенный на листинге 13.4.

Листинг 13.4. Обработчик сообщений команды Clear Drawing.

```
void CMSDoc::OnEditClearDrawing()
{
      // ЧТО СДЕЛАТЬ: Поместите здесь код обработчика вашей команды
      m_data.SetSize(0,128);
      SetModifiedFlag();
      UpdateAllView(NULL);
}
```

Завершив внесение изменений, откомпилируйте и запустите программу. Обратите внимание на то, что:

- В отличие от команды меню File|New, выбор Edit|Clear Drawing не приводит к выдаче подсказки, связанной с сохранением предыдущего документа.
- В случае выбора команды File|Save очищаемый документ сохраняется под существующим именем.
- Определенная строка подсказки отображается в строке состояния при выборе нового элемента меню.
- Команду Clear Drawing можно выбрать при помощи клавиатурной комбинации.

Функционирующая программа показана на рис. 13.8.

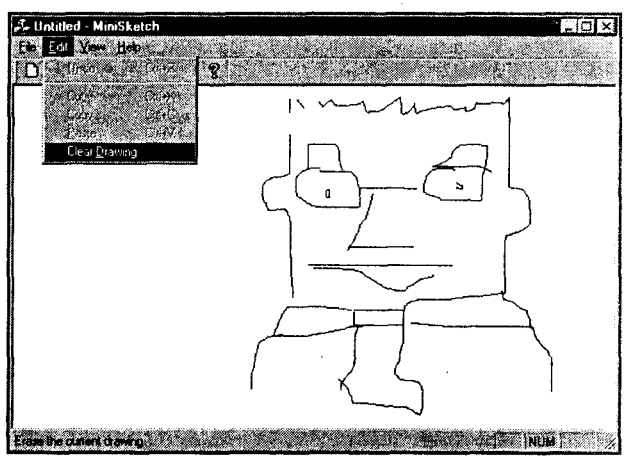


РИСУНОК 13.8.
Использование элемента меню Clear Drawing

За кулисами: что делал мастер ClassWizard

В большинстве случаев никого не интересует, что делает ClassWizard "в фоновом режиме". Однако в случае командного сообщения это следует знать, поскольку в программу потребуется поместить собственные обработчики команд. К счастью, делается это довольно просто.

Для добавления в меню новой команды ClassWizard должен решить три задачи. Давайте рассмотрим, что они собой представляют.

Прототип обработчика

ClassWizard начинает с помещения прототипа функции обработки команды в класс, отвечающий за обработку этой команды. ClassWizard заключает прототип в специальный комментарий, благодаря которому прототип становится собственностью ClassWizard (см. выделенные на листинге 13.5 строки). Прототип ClassWizard можно удалить, однако нельзя в него вносить изменения. Кроме того, нельзя помещать прототип, заключенный в специальный комментарий. Для добавления прототипа вставьте его за пределами специального комментария, а для внесения изменений, удалите существующий прототип и добавьте свой собственный вне скобок специального комментария.

Листинг 13.5. Часть листинга определения класса CMSDoc

```
class CMSDoc : public CDocument
{
// Код определения класса опускается
// Сгенерированные функции для карты сообщений
protected:
        //{{AFX_MSG(CMSDoc)
        afx_msg void OnEditClearDrawing();
        //}}AFX_MSG
DECLARE_MESSAGE_MAP()
private:
      CArray<CPoint, CPoint> m_data;
};
```

ПРИМЕЧАНИЕ

Для упрощения поиска сгенерированных функций для карты сообщений непосредственно перед ними MFC помещает префикс *AFX_MSG*. Если какой-то функции предшествует *AFX_MSG*, можно с уверенностью утверждать, что она присутствует в карте сообщений. Помимо этого префикс *AFX_MSG* ничего не делает, так что его можно безболезненно опустить при добавлении собственных функций в карту сообщений.

Тело обработчика

Во-вторых, ClassWizard помещает тело функции обработки в файл реализации (CPP) определенного класса. Обработчики простых команд не требуют аргументов и не возвращают значений, посему ClassWizard остается лишь предложить имя и далее использовать его при генерации соответствующей функции. Собственный обработчик сообщения **WM_COMMAND** можно назвать как угодно; тем не менее, его имя должно совпадать с именем, выбранным при определении класса.

Макрокоманда ON_COMMAND

И наконец, ClassWizard ассоциирует функцию обработки с идентификатором ресурса соответствующей команды. Это достигается за счет помещения макрокоманды **ON_COMMAND()** в карту сообщений соответствующего класса.

Как и в случае прототипа класса, ClassWizard заключает макрокоманду **ON_COMMAND()** в специальный комментарий. Собственная карта сообщений

должна добавляться за пределами этого комментария. В листинге 13.6 показан элемент карты сообщений для команды Clear Drawing.

Листинг 13.6. Карта сообщений класса CMSDoc

```
BEGIN_MESSAGE_MAP(CMSDoc, CDocument)
    //{{AFX_MSG_MAP(CMSDoc)
    ON_COMMAND(ID_EDIT_CLEARDRAWING, OnEditClearDrawing)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

Вновь реализованная функция соединяется с конкретным командным сообщением, подлежащим обработке, за счет помещения макрокоманды **ON_COMMAND** в карту сообщений класса. Макрокоманда **ON_COMMAND** требует двух аргументов: идентификатор ресурсов для обрабатываемой команды и имя функции обработки сообщений. Обратите внимание, что несмотря на сходство элемента карты сообщений с вызовом функции, точка с запятой не присутствует.

Теперь, когда известно, как заставить элементы меню реагировать требуемым образом, следует вернуться в Menu Editor и выполнить в нем дополнительные действия. Добавление в программу MiniSketch возможности выбора цвета и ширины пера позволит исследовать большинство элементов меню.

Еще раз обратимся к перьям

Через короткое имя вы приступите к процедуре добавления меню Pens (Перья). Но до того необходимо разобраться с меню Edit.

Еще недавно меню Edit счастливо сосуществовало с командами Undo, Cut, Copy и Paste, помещенными в него AppWizard.

В отличие от большинства элементов меню, добавляемых AppWizard, эти элементы не вызывают никаких действий. Для того, чтобы они работали, их придется подключить самостоятельно. В некоторых программах (подобных текстовому редактору) эти элементы меню активизируются достаточно просто. Однако в графических программах это требует значительно больших усилий. Например, для активизации Copy вначале потребуется обеспечить возможность выбора некоторого фрагмента изображения, что представляет собой далеко не тривиальную задачу.

В свете сказанного, мы намерены искать пути проще и удалить из меню Edit все элементы, за исключением Clear Drawing. Выберите верхний элемент (Undo), затем нажмите клавишу Delete и удерживайте ее в таком состоянии до тех пор, пока все элементы меню не будут удалены. После удаления содержимого меню Edit считайте, что вы готовы работать с меню Pens.

Добавление нового меню

Меню Pens — это новое меню верхнего уровня, которое следует поместить между меню View и меню Help. Щелчок на Pens приведет к открытию меню, тем самым обеспечивая возможность выбора цвета, толщины и стиля пера.

Начнем с помещения в программу самого меню Pens. Для этого потребуется выполнить следующие действия:

1. Загрузив проект MiniSketch, откройте меню **IDR_MAINFRAME** в Menu Editor. Новое меню можно добавить, выбрав пунктирный прямоугольник справа от

меню Help в верхней полосе Menu Editor. Поскольку меню Pens должно находиться слева от меню Help, выберите меню Help и перетащите его вправо от точки вставки. На рис. 13.9 показан Menu Editor во время перетаскивания меню Help.

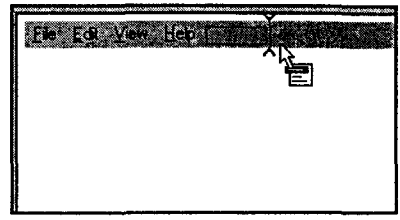


РИСУНОК 13.9. *Перетаскивание меню Help к новой позиции вставки*

2. Выберите при помощи мыши точку вставки и введите с клавиатуры заголовок "&Pens". Сразу же после начала ввода открывается диалоговое окно Menu Item Properties (см. рис. 13.10). Обратите внимание, что флажок Pop-up (Всплывающее меню) отмечен, а поле со списком идентификаторов недоступно — всплывающие меню не имеют (и не требуют) уникальных идентификаторов. При выборе меню Pens сообщение **WM_COMMAND** не генерируется; это имеет место только для элементов меню.

Цвет пера: включение опций в диалоговом окне

После добавления меню Pens под ним появляется прямоугольник выбора нового элемента меню. Именно в нем будут вводиться элементы меню Pens. Для начала добавляется элемент, обеспечивающий выбор нового цвета пера.

Соответствующие шаги показаны ниже:

1. Выберите новый элемент меню, после чего либо дважды щелкните на нем, либо начните вводить заголовок "&Color...". Для установки цвета пера будет использоваться общий диалог Color, как это делалось в программе PaintORama.
2. Заполните поле Prompt текстом "Выбрать текущее перо для рисования\nЦвет пера". Раскройте поле со списком идентификаторов и убедитесь, что выбран идентификатор **ID_PENS_COLOR**. Далее можно перейти к следующему шагу — подключению необходимых обработчиков. Экран должен выглядеть так, как показано на рис. 13.11.

Активизация диалогового окна выбора цвета

Вы уже знаете, как создавать обработчик командных сообщений, выбрав элемент меню и затем вызвав ClassWizard из контекстного меню (или через Ctrl+W). Рассмотрим другой путь добавления обработчика сообщений — с использованием WizardBar, с которым вы уже сталкивались в начальных главах.

Visual C++ часто предоставляет несколько путей для достижения цели, из которых можно выбрать наиболее удобный. Ниже представлен перечень действий, которые потребуется выполнить, чтобы добавить обработчик команд меню при помощи WizardBar:

1. Убедитесь в том, что проект MiniSketch загружен, а меню **IDR_MAINFRAME** открыто в Menu Editor. Выберите элемент меню Pens|Color.
2. В WizardBar (который находится поверх окна Workspace и окна редактирования) выберите класс **CMSView** из крайнего левого поля со списком. После этого крайнее правое поле будет содержать класс **CMSView**. Щелкните на выпадающем меню в WizardBar и выберите пункт Add Windows Message Handler. Экран должен иметь вид, как на рис. 13.12.

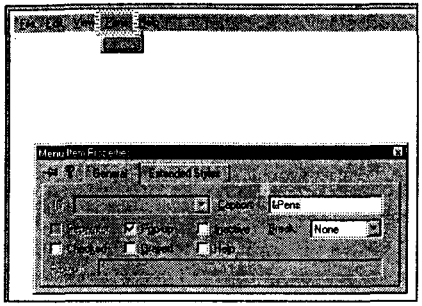


РИСУНОК 13.10. *Установка заголовка меню Pens*

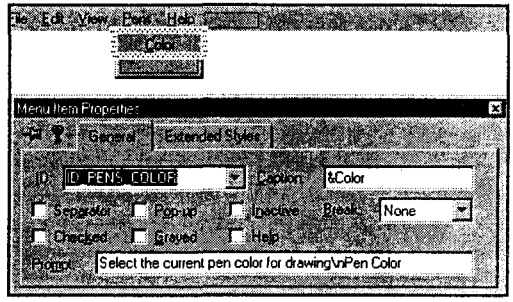


РИСУНОК 13.11. *Добавление элемента меню ID_PENS_COLOR*

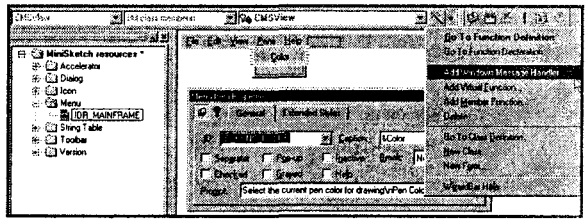


РИСУНОК 13.12.
Создание обработчика сообщений Windows из WizardBar

3. В появившемся диалоговом окне (с именем New Windows Message And Event Handlers For Class CMSView) из списка Class Or Object To Handle (класс или объект, подлежащий обработке) выберите **ID_PENS_COLOR**, а из списка New Windows Messages/Events — **COMMAND**. Когда экран приобретет вид, показанный на рис. 13.13, щелкните на Add and Edit.
4. В диалоговом окне Add Member Function выберите имя **OnPensColor()** и щелкните на OK. Вставьте строки, показанные в листинге 13.7.

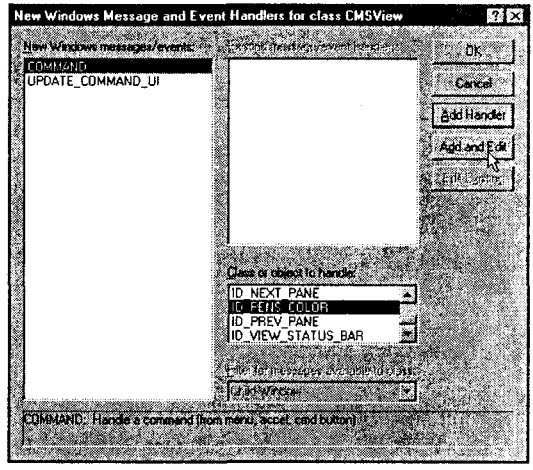


РИСУНОК 13.13. *Выбор опций для обработчика команд изменения цвета пера*

Листинг 13.7. Функция CMSView OnPensColor().

```
void CMSView::OnPensColor()
{
      // ЧТО СДЕЛАТЬ: Поместите здесь код
      // обработчика команд
      CColorDialog dlg(m_PenColor);
      if (dlg.DoModal == IDOK)
      {
            m_PenColor = dlg.GetColor();
            InitPen();
```

```
    }
}
```

Ядро поддержки

В коде функции **OnPensColor()** создается экземпляр общего диалога Color системы Windows и отображается при помощи функции **DoModal()**. Выбранный цвет запоминается в переменной **m_PenColor**, после чего вызывается функция **InitPen()** для изменения текущего пера.

К сожалению, приведенный код является источником нескольких проблем. Основная проблема заключается в том, что у вас еще нет переменной **m_PenColor** и функции **InitPen()**. Прежде чем можно будет воспользоваться новым меню, потребуется построить некоторую дополнительную инфраструктуру.

Вот что необходимо сделать:

1. Добавить элементы данных, показанные в таблице 13.1, в класс **CMSView**. Можно либо воспользоваться диалоговым окном Add Member Variable, либо вручную ввести их с клавиатуры в файле CMiniSketchView.h. Все переменные должны быть **private**. Скорее всего, вы должны помнить эти переменные, поскольку аналогичные применялись в программе PaintORama.

Таблица 13.1. Переменные для класса CMSView.

Тип	*Имя*	*Описание*
Cpen	m_Pen	Текущее перо, используемое в классе **CMSView**.
COLORREF	m_PenColor	Цвет текущего пера
int	m_PenStyle	Стиль текущего пера
int	m_PenWidth	Ширина текущего пера

2. Выполните инициализацию каждой переменной, поместив в конструктор **CMSView** код, представленный в листинге 13.8.

Листинг 13.8. Конструктор CMSView.

```
CMSView::CMSView()
{
      // ЧТО СДЕЛАТЬ: поместить сюда код конструктора
      m_PenColor = RGB(0,0,0);
      m_PenStyle = PS_SOLID;
      m_PenWidth = 1;
      m_Pen.CreatePen(m_PenStyle, m_PenWidth, m_PenColor);
}
```

3. Используйте диалоговое окно Add Member Function для добавления **public**-функции **InitPen()** в класс **CMSView**. Функция **void** не требует аргументов и не возвращает значение. Завершите создание функции, вставив в нее код, выделенный в листинге 13.9.

Листинг 13.9. Метод InitPen().

```
void CMSView::InitPen()
{
      m_Pen.DeleteObject();
      m_Pen.CreatePen(m_PenStyle, m_PenWidth, m_PenColor);
}
```

4. Добавьте показанную ниже строку в функцию **CMSView OnMouseMove()** непосредственно после строки, в которой выполняется создание **CClientDC**:

```
dc.SelectObject(&m_Pen);
```

Откомпилируйте программу и запустите ее. Как показано на рис. 13.14, выбор в меню команды Pen | Color приводит к открытию общего диалогового окна Color системы Windows. После выбора цвета программа использует для рисования новое перо.

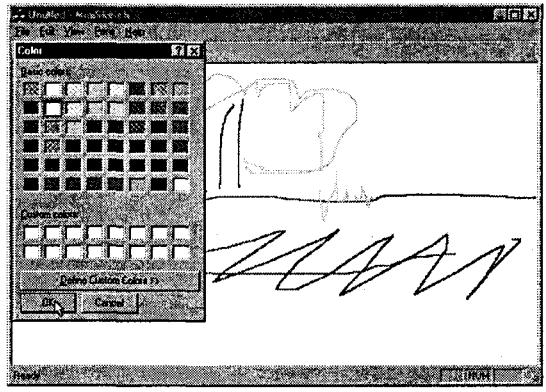


РИСУНОК 13.14. *Использование команды Pen|Color для изменения цвета пера*

Естественно, класс вашего документа не отслеживает цвета перьев, использованных для построения линий; таким образом, в случае изменения размеров либо сохранении и последующем восстановлении файла цветные изображения становятся черно-белыми. Данная проблема будет решаться в нескольких следующих главах по мере совершенствования классов документов и представлений.

Ширина пера: добавление подменю

После открытия меню Start (Пуск) в Windows не все опции видны сразу. Например, щелчок на Programs (Программы) приводит к отображению другого меню, которое, в свою очередь, воспроизводит еще одно меню и т.д. Последние носят название *подменю*. В Menu Editor среды Visual C++ подменю создаются достаточно просто.

В программе PaintORama для задания ширины пера использовался счетчик. Что касается программы MiniSketch, то вместо него будет применяться набор элементов меню: Thin (Тонкая), Medium (Средняя), Thick (Толстая) и Extra Thick (Очень толстая). Во избежание загромождения меню Pens эти элементы следует поместить в отдельное подменю. Затем для их подключения необходимо воспользоваться специальной формой макрокоманды **ON_COMMAND — ON_COMMAND_RANGE**. И наконец, используя акселераторы Windows, каждому из упомянутых элементов присваивается горячая клавиша.

Вы готовы? Двигаемся дальше.

Построение подменю

Во-первых, требуется создать подменю Pens|Width. Выполните следующие шаги (они отымут совсем немного времени):

1. Загрузите проект MiniSketch, при этом меню **IDR_MAINFRAME** должно быть открыто в Menu Editor. Откройте меню Pens, — перейдите в его команду и выберите новый элемент этого меню. Введите надпись "&Width" и отметьте флажок Pop-up. После этого Visual C++ запрещает доступ к полю со списком идентификаторов ресурсов. В заголовок подменю Windows автоматически поместит стрелку вправо. Как только ваш экран примет вид, показанный на рис. 13.15, можно считать, что вы готовы перейти к следующему шагу.

2. Во всплывающем меню, привязанном к Pens|Width, воспользуйтесь значениями из таблицы 13.2 для включения в меню четырех элементов. Обратите внимание, что каждый заголовок содержит определение горячей клавиши, следующее за символическим представлением табуляции (\t). Рисунок 13.16 иллюстрирует добавление последнего элемента меню.

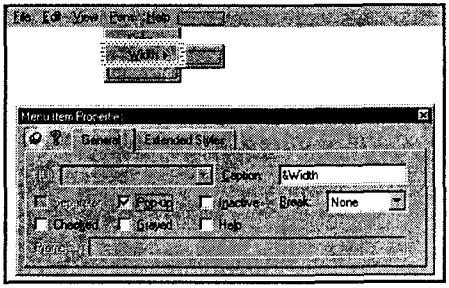


РИСУНОК 13.15. *Добавление подменю Pens|Width.*

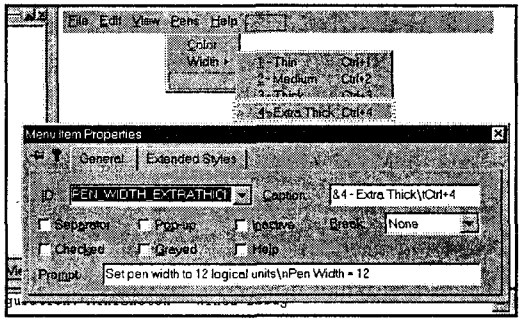


РИСУНОК 13.16. *Добавление элементов в подменю Pens|Width.*

Таблица 13.2. Элементы подменю Pens | Width

Идентификатор ресурса	*Заголовок элемента подменю*	*Строка подсказки*
ID_PEN_WIDTH_THIN	&1 – Thin\Ctrl+1	Установить ширину пера равной 1 пикселу\n Ширина пера = 1 пиксел
ID_PEN_WIDTH_MEDIUM	&2 – Medium\Ctrl+2	Установить ширину пера равной 4 логическим единицам\n Ширина пера = 4
ID_PEN_WIDTH_THICK	&3 – Thick\Ctrl+3	Установить ширину пера равной 8 логическим единицам\n Ширина пера = 8
ID_PEN_WIDTH_EXTRATHICK	&4 – Extra Thick\Ctrl+4	Установить ширину пера равной 12 логическим единицам\n Ширина пера = 12

После завершения этого шага можно откомпилировать и запустить программу. Однако все элементы подменю Pens|Width оказываются недоступными. Это связано с тем, что пока еще ни один обработчик команд не подключен. Решим эту проблему сейчас.

Обработка выбора ширины пера

Вы уже знакомы с одним способом обработки выбора элемента меню: использование ClassWizard для подключения четырех обработчиков **ON_COMMAND**. В

каждом таком обработчике устанавливается соответствующее значение **m_PenWidth** и вызывается **InitPen()**. Поскольку обработчик команды выбора ширины пера делает немногое, этот путь, по-видимому, столь же хорош, как и любой другой. Однако, раз мы уж занялись этим делом, рассмотрим другой вариант, предусматривающий использование метода **ON_COMMAND_RANGE**.

Метод **ON_COMMAND_RANGE()** работает при наличии упорядоченной последовательности идентификаторов меню. При помощи **ON_COMMAND_RANGE()** MFC пересылает полный набор команд в один обработчик вместо передачи каждой команды отдельному обработчику. Для уведомления о выбранном элементе меню Windows передает в функцию в качестве аргумента соответствующий идентификатор.

При создании набора элементов меню одного за другим, как это имело место в отношении меню Pens|Width, Windows присваивает им последовательные идентификационные номера. Для анализа номеров идентификаторов можно воспользоваться меню View|Resource Symbols. Как нетрудно убедиться, взглянув на рис. 13.17, идентификаторы ресурсов для элементов меню Pens|Width представляют собой последовательные номера от 32 773 до 32 776. (Фактические значения роли не играют.)

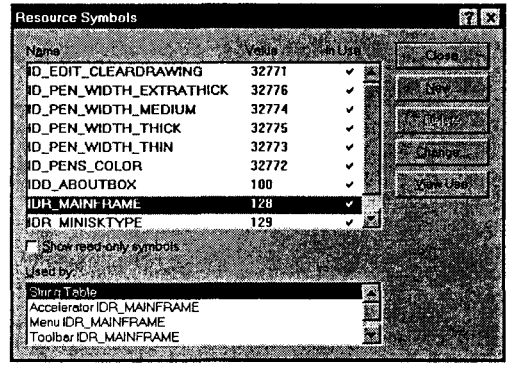


РИСУНОК 13.17. *Диалоговое окно Resource Symbols.*

Теперь, когда известно, что идентификаторы элементов меню принимают последовательные числовые значения, несложно добавить обработчики сообщений. Для этого потребуется выполнить такие действия:

1. Вставить прототип, показанный в листинге 13.10, в определение класса **CMSView**. Потребуется ввести лишь выделенную строку — все остальное служит чисто справочным целям.

Листинг 13.10. Прототип функции OnPensWidth().

```
// Сгенерированные функции карты сообщений
protected:
      //{{AFX_MSG(CMSView)
      afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
      afx_msg void OnLButtonUp(UINT nFlags, CPoint point);
      afx_msg void OnMouseMove(UINT nFlags, CPoint point);
      afx_msg void OnPensColor();
      //}}AFX_MSG
      afx_msg void OnPensWidth(UNIT nCmd);
DECLARE_MESSAGE_MAP()
```

2. Поместить выделенную строку из листинга 13.11 в карту сообщений класса **CMSView**.

Листинг 13.11. Карта сообщений класса CMSView.

```
BEGIN_MESSAGE_MAP(CMSView, CView)
      //{{AFX_MSG_MAP(CMSView)
```

```
        ON_WM_LBUTTONDOWN()
        ON_WM_LBUTTONUP()
        ON_WM_MOUSEMOVE()
        ON_COMMAND(ID_PENS_COLOR, OnPensColor)
        //}}AFX_MSG_MAP
        ON_COMMAND_RANGE(ID_PEN_WIDTH_THIN, \
                         ID_PEN_WIDTH_EXTRATHICK, \
                         OnPensWidth)
// Стандартные команды печати
        ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
        ON_COMMAND(ID_FILE_PRINT_DIRECT, CView::OnFilePrint)
        ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
END_MESSAGE_MAP()
```

3. Добавить тело функции **OnPensWidth()**, показанное в листинге 13.12, в файл MiniSketchView.cpp. Последнее следует выполнить вручную, а не через диалоговое окно Add Member Function, поскольку соответствующий прототип в определение класса уже вставлен. Обратите внимание, что ширина пера вычисляется путем вычитания из фактически выбранного идентификатора первого допустимого значения и последующего умножения результата на 4. В конечном итоге получается значение, составляющее 0, 4, 8 или 12. Значение, равное 0, указывает, что в любом случае перо получит ширину в один пиксел — именно то, что и требовалось.

Листинг 13.12. Обработчик OnPensWidth() диапазона командных сообщений.

```
void CMSView::OnPensWidth(UINT nCmd)
{
        m_PenWidth = (nCmd - ID_PEN_WIDTH_THIN) * 4;
        InitPen();
}
```

Откомпилируйте программу и протестируйте каждую опцию меню. Как показано на рис. 13.18, появляется возможность изменить как цвет, так и толщину пера.

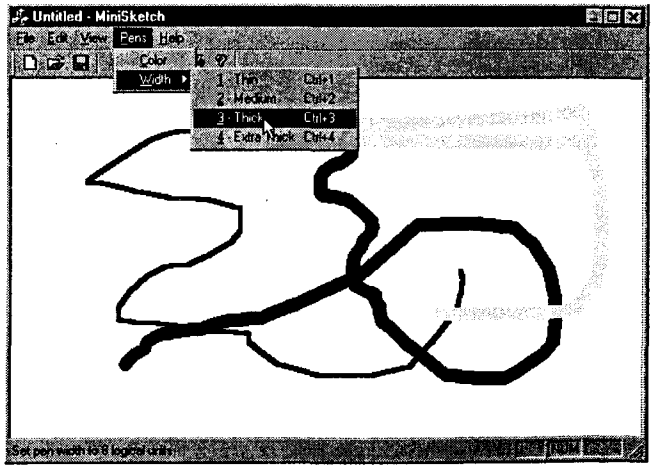


РИСУНОК 13.18.
Программа MiniSketch с добавленным меню Pens|Width.

ПРИМЕЧАНИЕ

По всей вероятности, вы обратили внимание на использование символа продолжения строки (\). Он присутствует в макрокомандах, а не в функциях; если отказаться от его применения, возникнут ошибки при компиляции программы.

Куда двигаться дальше?

На данный момент вы овладели практически всеми основными приемами работы с меню. Теперь вам известно, как добавлять в меню новые элементы, удалять неиспользуемые, устанавливать горячие клавиши и подключать функции к элементам меню при помощи ClassWizard. Вы знаете, как создавать подменю и меню, которые вызывают диалоговые окна. Кроме того, прояснился способ помещения подсказки в строку состояния, а также технология распознавания непрерывной последовательности команд меню в одном обработчике. Что же осталось?

Довольно своевременный вопрос. Что вы скажете о:

- *Проверке выбранного элемента меню* — Как уведомить пользователя, какая ширина пера им выбрана, не вычерчивая линий?
- *Разрешении и запрещении команд меню* — Например, если активны стилизованные перья, то меню Pens|Width больше не имеет смысла, поскольку все стилизованные перья имеют ширину, составляющую один пиксел.
- *Установке горячих клавиш, присвоенных элементам меню Pens|Width* — Для этого придется изучить таблицу акселераторов Windows.
- *Настройке панели инструментов с целью организации моментального доступа к часто используемым командам меню.*

Таким образом, дальнейшая картина ясна. Пройден длинный путь, однако цель еще не достигнута. Передохнем немного и займемся панелями инструментов.

Глава 14

Меню, панели инструментов и строки состояния

"Вы не знаете, чем вы обладаете, пока оно не будет потеряно," — поет Джони Митчел (Joni Mitchell) в своей композиции "Большое желтое такси". И это правда. Вот почему разработчики программного обеспечения никогда ничего не выбрасывают. Вы можете быть уверены, что каждая возможность первой версии программы в той или иной форме сохранится и в ее, скажем, тринадцатой версии.

Теневая сторона принципа "выживает самый толстый" применительно к программному обеспечению заключается в том, что в 1980 году самый современный по тем временам текстовый процессор помещался на дискете емкостью 160 Кб, а сегодня для такого процессора потребуется 60 Мб пространства. Даже несмотря на использование лишь малой толики всех возможностей, предлагаемых современным текстовым процессором, известно, что на коробке со следующей версией не будет написано: "Количество возможностей уменьшено!"

Точно так же как нельзя по достоинству оценить существующие возможности до тех пор, пока они не станут недоступными, так иногда не удается понять, чем может оказаться полезным то или иное функциональное средство, пока не поработать с ним некоторое время. Возьмем, например, выпадающее меню. Когда разработчики пользовательского интерфейса впервые реализовали выпадающие меню, они рассматривали их как всеобъемлющее решение всех аспектов использования компьютера. Они искренне верили, что найти другое средство, обеспечивающее большее удобство и быстроту управления, попросту невозможно. Однако чем большее распространение получали выпадающие меню, тем более очевидными становились их недостатки, в связи с чем разработчики изобретали новые средства.

В этой главе рассматриваются четыре вида усовершенствований базовой системы меню:

- *Командный UI.* Позволяет программистам обеспечивать для пользователей обратную связь за счет манипулирования *пользовательским интерфейсом (UI — User Interface) на основе меню.*
- *Клавиатурные акселераторы.* Предоставляют пользователям возможность обходить систему меню и выбирать команды путем ввода клавиатурных комбинаций.
- *Панели инструментов.* Обеспечивают возможность выбора часто используемых команд меню при помощи щелчка мышью на видимых и легко доступных кнопках. В совокупности клавиатурные акселераторы и панели инструментов выводят взаимодействие с приложением через систему меню на поверхность, а не хоронят его глубоко в недрах вложенных подменю.
- *Строки состояния.* Отображают для пользователя важную информацию, на которую необходимо обратить особое внимание.

Командный пользовательский интерфейс

Если вы достаточно наигрались с программой MiniSketch, то, должно быть, заметили, что для получения информации о текущем пере необходимо начертить совершенно ненужную линию. Все было бы гораздо проще, если бы в меню Pens | Width выбранная на текущий момент ширина пера каким-то образом отмечалась.

Windows API всегда включал в себя поддержку размещения отметок рядом с элементами меню. Перед отображением выпадающего меню Windows посылает сообщение **WM_INITPOPUP**. Приняв это сообщение, приложение может переходить с одного элемента меню на другой, добавляя или удаляя отметки и при необходимости разрешая или запрещая элементы меню.

Программирование такой операции в Windows не вызывает особых трудностей, однако оно достаточно однообразно и утомительно. К счастью, в MFC

эта операция столь же проста в реализации, как и реакция на команду меню. Используется довольно похожий метод — макрокоманда **UPDATE_COMMAND_UI**.

Макрокоманда **UPDATE_COMMAND_UI** поручает Windows вызвать указанную функцию непосредственно перед отображением соответствующего элемента меню. Во время вызова Windows передает указатель на объект **CCmdUI**, которым можно воспользоваться для разрешения запрещения элемента либо для установки или сброса отметки.

Командный UI системы меню можно обновить тремя способами:

- Связать каждый элемент меню с конкретным обработчиком командного UI.
- Использовать один и тот же обработчик командного UI для нескольких элементов меню.
- Связать некоторый диапазон элементов меню с одним и тем же обработчиком командного UI.

Кратко рассмотрим каждый из методов на примере меню Pens|Width. Цель заключается в том, чтобы поместить отметку рядом с актуальным значением ширины пера и удалить отметки у всех других значений ширины.

Метод 1: индивидуальные обработчики UI

Начнем с простейшего метода (он считается простейшим в силу того, что ClassWizard обеспечивает максимально возможную его поддержку). Имейте в виду, что простейший не всегда означает наилучший — такой метод может привести к появлению множества избыточных строк кода.

Потребуется выполнить такие шаги:

1. Откройте проект MiniSketch, а затем в Menu Editor загрузите меню **IDR_MAINFRAME**. Откройте меню Pens|Width и выберите в нем элемент Thin. Экран должен принять вид, показанный на рис. 14.1.
2. При помощи контекстного меню, доступного по щелчку правой кнопкой мыши, загрузите ClassWizard и выберите в нем **CMSView** из списка Class Name. Выберите **ID_PEN_WIDTH_THIN** из списка Object IDs и **UPDATE_COMMAND_UI** из списка Messages. Щелкните на Add Function, когда экран примет вид, показанный на рис. 14.2.
3. В диалоговом окне Add Member Function согласитесь с предложенным именем **OnUpdatePenWidthThin()** (см. рис. 14.3). Щелкните на OK, а затем после возврата в окно ClassWizard щелкните на Edit Code.
4. Поместите в функцию **OnUpdatePenWidthThin()** код из листинга 14.1. Метод **SetCheck()** помещает отметку рядом с элементом меню, если его аргумент принимает значение **true**, и удаляет отметку при значении аргумента, равном **false**. MFC обращается к функции **OnUpdatePenWidthThin()** непосредственно перед отображением элемента меню Pens|Width|Thin. Отметка помещается возле элемента меню, если переменная **m_PenWidth** принимает нулевое значение, что соответствует тонкому перу.

Листинг 14.1. Обработчик OnUpdatePenWidthThin() для тонкого пера.

```
void CMSView::OnUpdatePenWidthThin(CCmdUI* pCmdUI)
{
```

```
        // ЧТО СДЕЛАТЬ: Поместить здесь код обработчика командного UI
        pCmdUI->SetCheck(0 == m_PenWidth);
}
```

5. Повторите аналогичную процедуру для каждого из четырех оставшихся элементов меню Pens|Width. Используйте этот же код для каждой опции, однако осуществите проверку для, соответственно, 4, 8 и 12.

Завершив перечисленные выше шаги, откомпилируйте программу MiniSketch и протестируйте возможность установки любой ширины пера. По мере выбора различных элементов меню, можно будет наблюдать отображение отметок, сопровождающее каждый выбор (см. рис. 14.4).

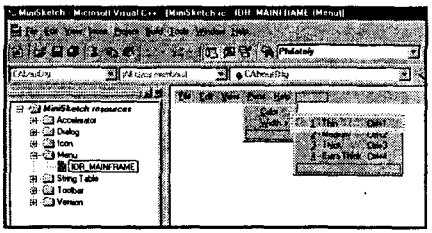


РИСУНОК 14.1. *Открывание меню Pens|Width в Menu Editor.*

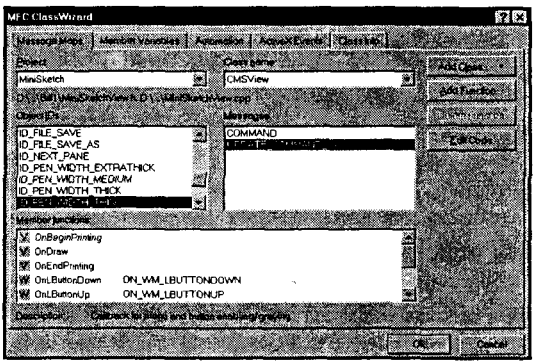


РИСУНОК 14.2. *Добавление обработчика сообщений ID_PEN_WIDTH_THIN при помощи ClassWizard.*

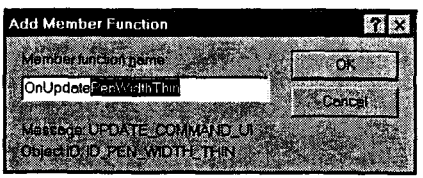


РИСУНОК 14.3. *Использование диалогового окна Add Member Function.*

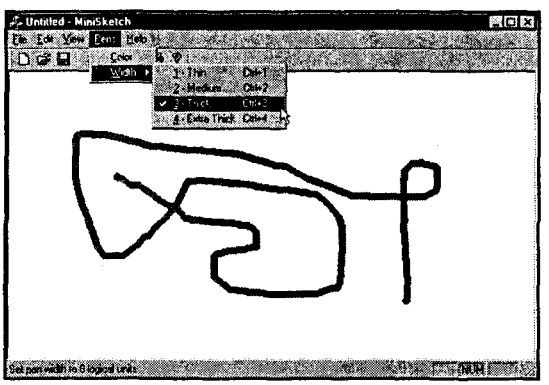


РИСУНОК 14.4. *Обработчики командного UI для меню Pens|Width в действии.*

Возможно, тот факт, что меню Pens|Width в MiniSketch после запуска вообще не отображает отметок, может вызвать удивление. Почему так происходит? Во время внимательного анализа кода конструктора класса **CMSView** обнаруживается, что переменная **m_PenWidth** инициализируется значением 1. К сожалению, 1 не есть значение, пригодное для обработчиков командных UI — допускаются

только значения 0, 4, 8 или 12. Измените код конструктора **CMSView** таким образом, чтобы переменная **m_PenWidth** инициализировалась одним из упомянутых значений, и вы увидите, что программа заработает.

Метод 2: множество макрокоманд, один обработчик

Поскольку все четыре обработчика командного UI работают сходным образом, все макрокоманды **UPDATE_COMMAND_UI** можно связать с одной и той же функцией. Карта сообщений будет иметь четыре входа, но каждый вход будет ссылаться на одну и ту же функцию.

Для решения этой задачи в рамках ClassWizard создайте новую копию программы MiniSketch из главы 13. (При необходимости воспользуйтесь сопровождающим книгу CD-ROM.) Затем выберите каждый элемент меню точно также, как это делалось ранее. Однако в диалоговом окне Add Member Function не соглашайтесь с предложенным именем — вместо него задайте одно и то же имя функции для всех четырех элементов.

В рассматриваемом случае имеет смысл выбрать имя **OnUpdatePensWidth()**, поскольку его несложно получить, отредактировав предложенную строку в диалоговом окне Add Member Function. Воспользуйтесь вычислениями, показанными в листинге 14.2, для установки значения **m_PenWidth**. Обратите внимание, что для определения исследуемой опции в этом коде используется конструкция **pCmdUI->m_nID**.

Листинг 14.2. Функция OnUpdatePensWidth().

```
void CMSView::OnUpdatePensWidth(CCmdUI * pCmdUI)
{
      int thisWidth = (pCmdUI->m_nID - ID_PEN_WIDTH_THIN) * 4;
      pCmdUI->SetCheck(m_PenWidth == thisWidth);
}
```

Метод 3: альтернативный обработчик ON_UPDATE_COMMAND_UI_RANGE

Даже в случае отображения нескольких элементов меню на один и тот же обработчик, код содержит несколько входов карт и сообщений. Избыточные входы карт можно безболезненно удалить.

С целью упрощения MFC предусматривает несколько расширенных макрокоманд обработчика UI. К сожалению, даже если вы предпочтете ими воспользоваться, ClassWizard не окажет особой помощи — писать код придется самостоятельно. Следует заметить, что разработка кода не вызывает особых трудностей, но в то же время появляются все возможности получить намного более элегантный результат.

В данном примере используется расширенный обработчик командного UI:

```
ON_UPDATE_COMMAND_UI_RANGE
```

Как и в случае **ON_COMMAND_RANGE**, упомянутая макрокоманда может применяться только в случае, если идентификаторы меню принимают последовательные значения (что имеет место в нашем примере).

Выполните следующие действия:

1. Как и ранее, начните с проекта, который был завершен в главе 13. Добавьте определение метода (прототипа) обработчика сообщений в заголовок класса **CMSView**. Для этого воспользуемся листингом 14.2:

```
afx_msg void OnUpdatePensWidth(CCmdUI * pCmdUI);
```

2. Добавьте макрокоманду **ON_UPDATE_COMMAND_UI_RANGE** в карту сообщений класса **CMSView**. Поместите новую макрокоманду в файл MiniSketchView.cpp между макрокомандами **BEGIN_MESSAGE_MAP** и **END_MESSAGE_MAP**. Ваша макрокоманда должна выглядеть так:

```
ON_UPDATE_COMMAND_UI_RANGE(ID_PEN_WIDTH_THIN, \
                           ID_PEN_WIDTH_EXTRATHICK, \
                           OnUpdatePensWidth)
```

   Убедитесь, что после закрывающей скобки отсутствует точка с запятой.

3. Добавьте функцию **OnUpdatePensWidth()** в файл класса реализации **CMSView**, воспользовавшись при этом листингом 14.2.

После компиляции программа должна функционировать так же, как и ранее.

Акселераторы

Меню Windows позволяют осуществлять доступ к их элементам при помощи клавиатуры, что определяется путем помещения амперсанда (&) в заголовок элемента меню. В программе MiniSketch такая возможность уже реализована. Например, пользователь MiniSketch может выбрать самое широкое перо, нажимая Alt+P (для открытия меню Pens), W (для открытия меню Width) и 4 (для выбора самого широкого пера).

Быстрые клавиши обеспечивают путь доступа к системе меню через клавиатуру, однако пользователь при этом все еще должен выполнять перемещения по меню. В противоположность этому акселераторы предлагают альтернативный способ ввода команд в программе — способ, не требующий навигации по системе меню.

При определении заголовков элементов меню Pens|Width, в них включались подсказки для клавиатурных акселераторов, которые следовали непосредственно за управляющим символом \t. В меню программы MiniSketch можно обнаружить Ctrl+1, Ctrl+2 и т.д. Самое широкое перо можно выбрать, просто нажав Ctrl+4. Тем не менее, наличие подсказок для акселераторов не активизирует сами акселераторы.

Ниже представлены шаги, при помощи которых решается проблема активизации акселераторов:

1. Имея дело с самой последней версией проекта MiniSketch, выберите окно ResourceView и раскройте папку Accelerator. Дважды щелкните на **IDR_MAINFRAME** с тем, чтобы на экране появилось окно Accelerator Editor (Редактор акселераторов). Экран должен иметь вид, показанный на рис. 14.5.

2. Прокрутите до конца список акселераторов и дважды щелкните на пустой позиции в конце списка. Откройте диалоговое окно Accel Properties (Свойства акселератора), показанное на рис. 14.6.

3. Отыщите **ID_PEN_WIDTH_THIN** в раскрывающемся списке идентификаторов.

4. Щелкните на Next Key Typed (Ввести клавиатурный акселератор), затем, когда откроется диалоговое окно Press Accelerator Key (Нажать клавишу акселератора) нажмите клавишу 1 (см. рис. 14.7).

5. Отметьте флажок Ctrl в диалоговом окне Accel Properties. Окончательно заполненное диалоговое окно должно принять вид, показанный на рис. 14.8.

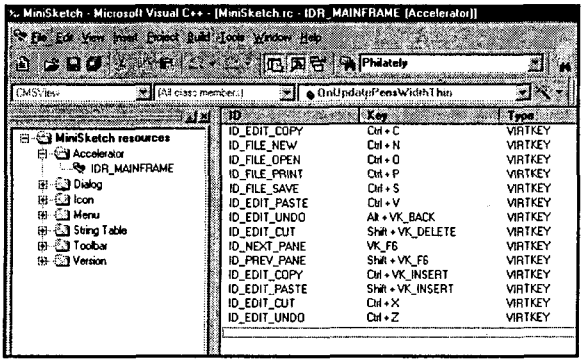


РИСУНОК 14.5. *Редактор акселераторов.*

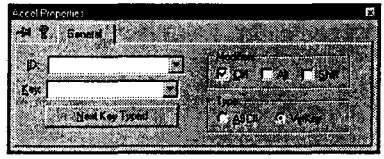


РИСУНОК 14.6. *Диалоговое окно Accel Properties.*

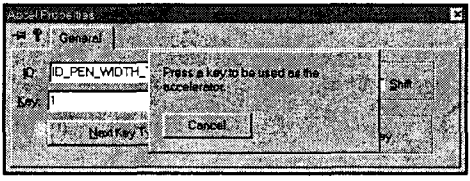


РИСУНОК 14.7. *Диалоговое окно Press Accelerator Key.*

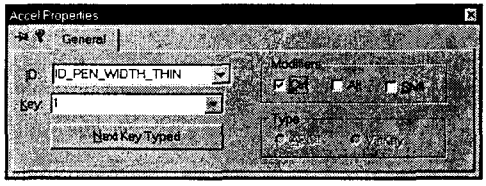


РИСУНОК 14.8. *Заполненное диалоговое окно Accel Properties.*

6. Выполните аналогичные действия по добавлению акселераторов для элементов меню Medium, Thick и Extra Thick, используя, соответственно, Ctrl+2, Ctrl+3 и Ctrl+4. В конечном итоге диалоговое окно приобретает вид, показанный на рис. 14.9.

Откомпилируйте и запустите на выполнение программу MiniSketch. Сейчас можно выбрать сверхтолстое или тонкое перо, не открывая меню Pens|Width.

Обратите внимание, что для одной и той же команды можно установить несколько акселераторов, например, это касается команды **ID_EDIT_CUT**. Кроме того, акселераторы можно определить и для команд, не включенных в меню. Команды с акселераторами, для которых нет ассоциированных идентификаторов меню, могут вводиться только с клавиатуры.

Панель инструментов

Панель инструментов состоит из маленьких кнопок, отображаемых под главным меню в верхней части экрана. Кнопки панели управления дублируют часто используемые элементы меню. Панели инструментов MFC обладают даже большей гибкостью — они допускают стыковку, так что пользователи могут перемещать их в любое место экрана. К тому же, сгенерированная AppWizard программа по умолчанию содержит элемент меню, отключающий панель инструментов.

ID	Key	Type
ID_PEN_WIDTH_THIN	Ctrl + 1	VIRTKEY
ID_PEN_WIDTH_MEDIUM	Ctrl + 2	VIRTKEY
ID_PEN_WIDTH_THICK	Ctrl + 3	VIRTKEY
ID_PEN_WIDTH_EXTRATHICI	Ctrl + 4	VIRTKEY
ID_EDIT_COPY	Ctrl + C	VIRTKEY
ID_FILE_NEW	Ctrl + N	VIRTKEY
ID_FILE_OPEN	Ctrl + O	VIRTKEY
ID_FILE_PRINT	Ctrl + P	VIRTKEY
ID_FILE_SAVE	Ctrl + S	VIRTKEY
ID_EDIT_PASTE	Ctrl + V	VIRTKEY
ID_EDIT_UNDO	Alt + VK_BACK	VIRTKEY
ID_EDIT_CUT	Shift + VK_DELETE	VIRTKEY
ID_NEXT_PANE	VK_F6	VIRTKEY
ID_PREV_PANE	Shift + VK_F6	VIRTKEY
ID_EDIT_COPY	Ctrl + VK_INSERT	VIRTKEY
ID_EDIT_PASTE	Shift + VK_INSERT	VIRTKEY
ID_EDIT_CUT	Ctrl + X	VIRTKEY
ID_EDIT_UNDO	Ctrl + Z	VIRTKEY

РИСУНОК 14.9.
Акселераторы для элементов меню Pens|Width.

Думайте о панелях инструментов как об акселераторах, предназначенных для пользователей, предпочитающих мышь перед клавиатурой. Выпадающие меню упрощают изучение программы и уменьшают риск неправомерных действий во время исследования возможностей программы. Акселераторы и панели инструментов, наоборот, усложняют изучение программы, однако упрощают ее использование.

Кнопки панели инструментов могут действовать тремя способами: как кнопки, как кнопки с фиксацией положения или как переключатели. Когда кнопка панели инструментов функционирует как простая кнопка, она выполняет действие, если на ней щелкнуть кнопкой мыши. Если это кнопка с фиксацией положения, она утапливается, когда на ней щелкнуть первый раз, и восстанавливает исходное положение при следующем щелчке. Хорошим примером фиксирующих кнопок могут служить кнопки Bold (Полужирный), Italic (Курсив) и Underline (Подчеркнутый) в текстовом процессоре Microsoft Word.

Если кнопка панели инструментов входит в состав группы кнопок, она может функционировать как переключатель. Например, в текстовом процессоре предусмотрены кнопки для выравнивания текста, такие как Left-Align (По левому краю), Center-Align (По центру), и Right-Align (По правому краю). Одна из них всегда находится в утопленном положении. Щелчок на утопленной кнопке не приводит к восстановлению ее положения, как это имеет место в случае фиксирующей кнопки, — в данном случае потребуется щелкнуть на другой кнопке группы.

Выбор цвета: обычная кнопка панели инструментов

Давайте добавим в панель инструментов кнопку обычного типа, которая обеспечит отображение диалогового окна Pens|Color. Чуть позже в панель инструментов будет помещен набор кнопок-переключатетей, предназначенных для выбора формы рисунка.

Для создания и манипулирования панелями инструментов в Visual C++ используется редактор панелей инструментов (Toolbar Editor). Упомянутый редактор представляет собой трехпанельный графический редактор, подобный Bitmap Editor или Icon Editor. В верхней панели находится представление всей панели инструментов целиком. Внутри Visual C++ панель инструментов рассматривается как один рисунок. В Toolbar Editor в каждый момент времени может редактироваться только одна кнопка; кроме того, можно свободно перемещать и удалять кнопки. Двойной щелчок на левой панели приводит к открытию диалогового окна Toolbar Button

Properties (Свойства кнопки панели инструментов). По завершении работ Toolbar Editor соберет фрагменты в единый рисунок, который будет использоваться в вашей программе.

Давайте познакомимся с Toolbar Editor. Выполним следующие действия:

1. Откройте проект MiniSketch и перейдите в окно ResourceView. Откройте папку Toolbar и дважды щелкните на **IDR_MAINFRAME** с целью загрузки Toolbar Editor. На рис 14.10 показан Toolbar Editor с панелью инструментов SDI по умолчанию, сгенерированной AppWizard.
2. Начните с удаления кнопок Cut, Copy и Paste. Выберите кнопку Cut (кнопка с пиктограммой, изображающей ножницы). Нажатие клавиши Delete или выбор из меню Cut не приводит к удалению этой кнопки — просто удаляется изображение. Для удаления собственно кнопки ее необходимо перетащить за пределы панели инструментов в любое другое место (не имеет значения, куда), что показано на рис. 14.11. Для удаления двух остальных кнопок проделайте то же самое.
3. Выберите крайнюю правую пустую кнопку, перетащите ее в позицию слева от кнопки Help. Захватите кнопку Help и сместите ее немного вправо. На рис. 14.12 показано, как должен выглядеть экран.
4. Выберите инструмент Pencil (Карандаш) из панели Graphics (графические инструменты). Щелкните левой кнопкой мыши на образце желтого цвета в палитре цветов (Colors). Начертите горизонтальную линию шириной в два пиксела, соблюдая отступы в два пиксела слева и справа и два пиксела сверху.

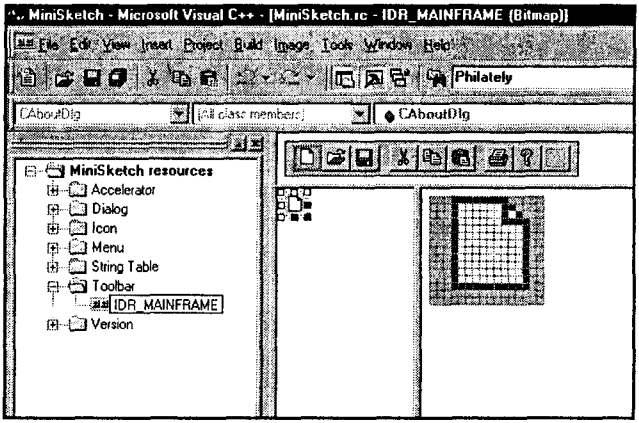


РИСУНОК 14.10. *Toolbar Editor с панелью инструментов SDI по умолчанию.*

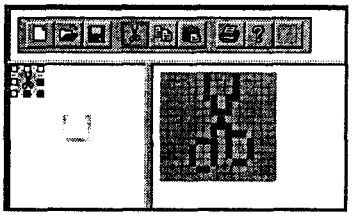


РИСУНОК 14.11. *Удаление кнопки панели инструментов.*

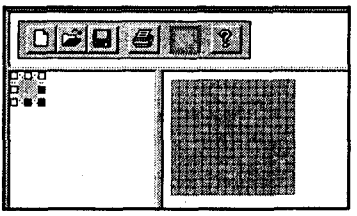


РИСУНОК 14.12. *Позиционирование новой кнопки выбора цвета пера.*

5. На один пиксел ниже желтой линии начертите похожую зеленую линию. Ниже зеленой линии начертите синюю, а затем и красную линии. Завершенный вид панели инструментов показан на рис. 14.13. (Обратите внимание, что на рисунке показаны также панели Graphics и Colors в виде плавающих окон.)
6. Дважды щелкните на левой панели Toolbar Editor (в любом месте растрового изображения) кнопки для открытия диалогового окна Toolbar Button Properties (Свойства кнопки панели инструментов). Обратите внимание, что идентификатор, отображаемый в раскрывающемся списке идентификаторов, представляет собой произвольное целое число. Откройте этот список и выберите идентификатор **ID_PENS_COLOR** (используемый для элемента меню Pens|Color), как показано на рис. 14.14. Других изменений в этом диалоговом окне делать не надо.

Откомпилируйте и запустите программу на выполнение. Теперь можно изменять цвет пера, воспользовавшись командой меню Pens|Color или кнопкой панели инструментов. Окно работающей программы представлено на рис. 14.15.

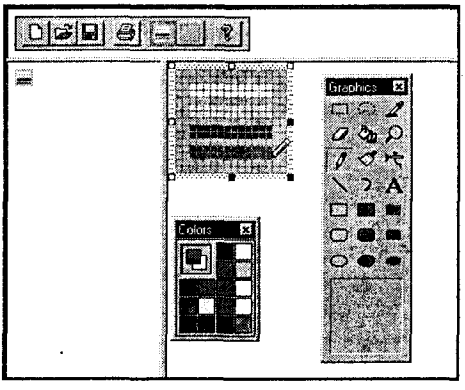


РИСУНОК 14.13. *Рисование линий в Toolbar Editor.*

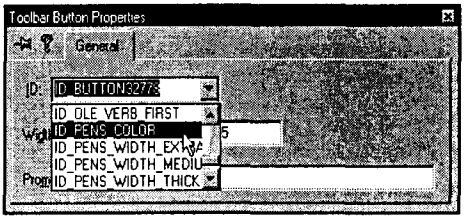


РИСУНОК 14.14. *Изменение идентификатора в диалоговом окне Toolbar Button Properties.*

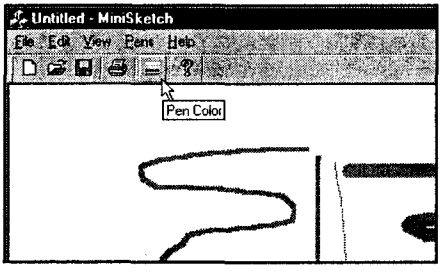


РИСУНОК 14.15. *Использование кнопки панели инструментов для выбора цвета пера в программе MiniSketch.*

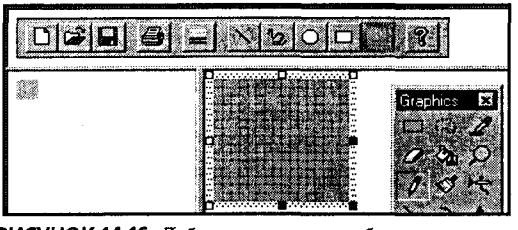


РИСУНОК 14.16. *Добавление кнопок выбора типа формы в панель инструментов.*

Добавление кнопок выбора типа формы: кнопки-переключатели панели инструментов

Для рисования форм в программе PaintORama применялось поле со списком **m_ShapesCombo**, представляющее все доступные виды форм. При этом значение,

хранящееся в поле со списком, присваивалось локальной переменной **drawMode**, значение которой анализировалось в операторе **switch** и на основе анализа производился выбор конкретной формы для рисования — линии, прямоугольника или эллипса.

Нечто подобное будет делаться и в программе MiniSketch, однако для переключения с одного стиля рисования на другой будут применяться кнопки панели инструментов. Когда пользователь щелкает на кнопке панели инструментов, соответствующее значение сохраняется в переменной **m_ShapeType**.

Для добавления кнопок выбора типа формы потребуется выполнить следующие шаги:

1. Поместите в панель инструментов еще четыре кнопки, организованные в виде группы. Нарисуйте для каждой из них соответствующие изображения: прямую линию, произвольную линию, овал и прямоугольник (см. рис. 14.16).

ПРИМЕЧАНИЕ

В данной главе разрабатывается лишь пользовательский интерфейс, обеспечивающий выбор конкретной формы. Вопросы, связанные с собственно рисованием форм рассматриваются в главах 15 и 16.

2. Для каждой из четырех добавляемых кнопок дважды щелкните на левой панели, чтобы открыть диалоговое окно Toolbar Button Properties. Для каждой кнопки создайте новый идентификатор: **ID_SHAPE_TYPE_LINE**, **ID_SHAPE_TYPE_FREEHAND**, **ID_SHAPE_TYPE_OVAL** и **ID_SHAPE_TYPE_RECTANGLE**. Определите соответствующие подсказки для строки состояния и помощи, как показано на рис. 14.17.

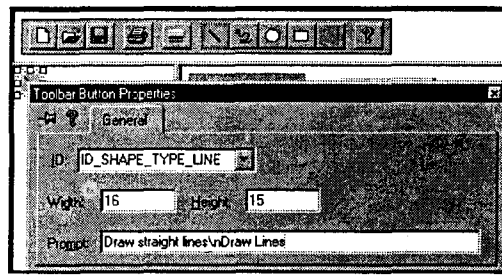


РИСУНОК 14.17.
Заполненное диалоговое окно Toolbar Button Properties.

3. Теперь, когда идентификаторы кнопок созданы, настало время решить проблемы обработки генерируемых ими сообщений. Запустите ClassWizard и выберите в поле со списком Class Name класс **CMSView**. В списке Object IDs найдите только что созданные четыре идентификатора (см. шаг 2). Добавьте соответствующие обработчики, как показано на рис. 14.18.
4. Поместите код, представленный в листинге 14.3, в четыре обработчика команд. Для экономии времени можно просто заменить код, сгенерированный ClassWizard.

Листинг 14.3. Обработчики команд для кнопок ID_SHAPE_TYPE панели инструментов.

```
// Обработка нажатия кнопок панели инструментов ID_SHAPE_TYPE_XX
void CMSView::OnShapeTypeLine()
{
      m_ShapeType = ID_SHAPE_TYPE_LINE;
}
```

```
void CMSView::OnShapeTypeOval()
{
      m_ShapeType = ID_SHAPE_TYPE_OVAL;
}
void CMSView::OnShapeTypeRectangle()
{
      m_ShapeType = ID_SHAPE_TYPE_RECTANGLE;
}
void CMSView::OnShapeTypeFreehand()
{
      m_ShapeType = ID_SHAPE_TYPE_FREEHAND;
}
```

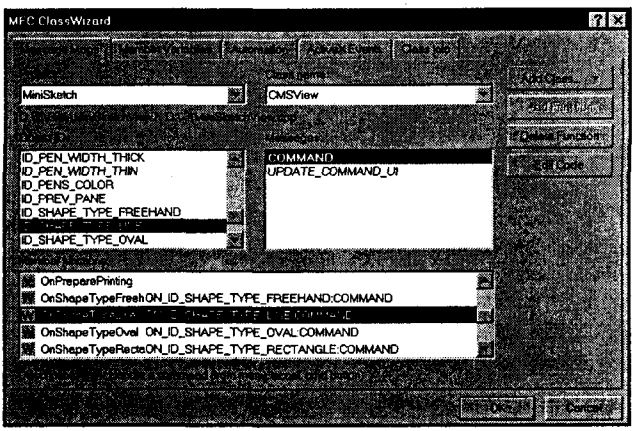


РИСУНОК 14.18.
Добавление обработчиков команд для идентификаторов ID_SHAPE_TYPE панели инструментов.

5. Определите в классе **CMSView** элемент данных **m_ShapeType** с типом **UINT** и доступом **private**.
6. Выполните инициализацию **m_ShapeType**, поместив в конструктор класса **CMSView** следующие строки:

```
// Инициализация типа формы
m_ShapeType = ID_SHAPE_TYPE_FREEHAND;
```

Обработчики кнопок выбора типа формы

Кнопки выбора типа формы функционируют не так, как другие кнопки панели инструментов. Когда пользователь щелкает на одной из них, программа выполняет соответствующее действие и оставляет кнопку в нажатом состоянии. Более того, если ранее была задействована другая кнопка выбора типа формы панели инструментов, то сейчас она должна вернуться в исходное состояние. Все это достигается за счет посылки выбранной кнопке панели инструментов сообщения **SetCheck()** в обработчике **UPDATE_COMMAND_UI**.

Рассмотрим необходимые шаги.

1. Откройте ClassWizard и выберите в списке Class Name класс представлений **CMSView**. Щелкните на **ID_SHAPE_TYPE_FREEHAND** в списке Object IDs, а затем выберите **UPDATE_COMMAND_UI** из списка Messages (Сообщения) и щелкните на кнопке Add Function. В открывшемся диалоговом окне Add

Member Function измените имя **OnUpdateShapeTypeFreehand** на **OnUpdateShapeType** (см. рис. 14.19). Щелкните на OK.

2. В ClassWizard выбирайте по одному остальные идентификаторы **ID_SHAPE_TYPE** и свяжите их с той же функцией. Для этого щелкните на Add Function, а затем в диалоговом окне Add Member Function введите **"OnUpdateShapeType"** в качестве имени функции. Экран должен приобрести вид, показанный на рис. 14.20.
3. Вставьте в функцию **OnUpdateShapeType()** код, показанный на листинге 14.4.

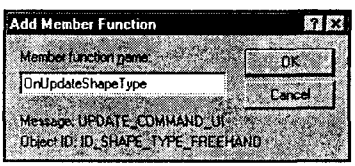


РИСУНОК 14.19. *Добавление обработчика OnUpdateShapeType().*

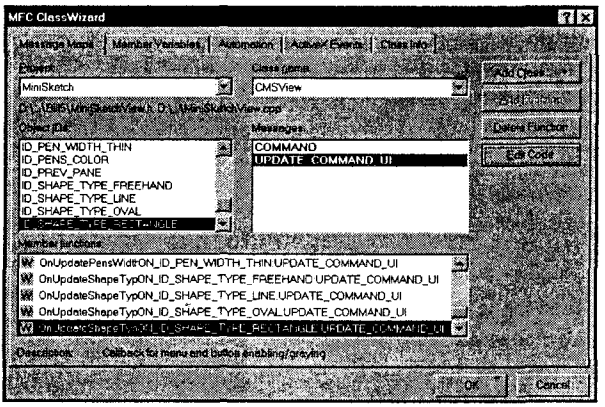


РИСУНОК 14.20. *Обработчики сообщений кнопок ID_SHAPE_TYPE панели инструментов.*

Листинг 14.4. Обработчик OnUpdateShapeType().

```
void CMSView::OnUpdateShapeType(CCmdUI* pCmdUI)
{
      pCmdUI->SetCheck(pCmdUI->m_nID == m_ShapeType);
}
```

Сейчас можно откомпилировать и запустить приложение. Теперь после щелчка на одной из кнопок выбора типа формы, она остается нажатой, тем самым обеспечивая простой и изящный способ управления, а также визуальную обратную связь. Новая возможность показана на рис. 14.21.

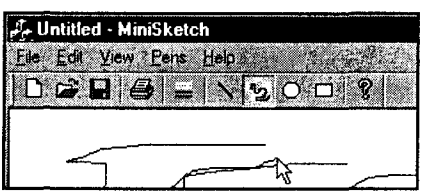


РИСУНОК 14.21. *Использование кнопок выбора типа формы панели инструментов.*

Естественно, код, который, выполняющий собственно рисование, пока не учитывает изменение значения **m_ShapeType**. Соответствующая реализация появится по мере совершенствования классов документов и представлений в нескольких последующих главах. Однако, несмотря на упомянутое несовершенство, приятно видеть, что панель инструментов работает.

Строки состояния

Последнее из рассматриваемых в данной главе усовершенствований пользовательского интерфейса — это строка состояний. Строка состояний не генерирует

команды подобно панелям инструментов, меню или акселераторам. Вместо этого она "разговаривает", вводя пользователя в курс дела и предоставляя ему важную информацию.

Как уже можно было убедиться, для создания строки состояния ничего делать не потребуется — необходимо лишь использовать ее. В этом разделе будет показано, как модифицировать встроенную строку состояния тремя различными способами:

- Удаляя индикаторы ScrollLock, NumLock и CapsLock.
- Добавляя индикатор ширины пера.
- Добавляя собственный индикатор, отображающий цвет пера.

Удаление нежелательных индикаторов

Удаление нежелательных индикаторов строки состояний — это самое простое из изменений, которые предстоит сделать. MFC сохраняет индикаторы в виде строковых ресурсов в строковой таблице приложения. Индикаторы клавиш CapsLock, NumLock и ScrollLock уже включены в строковую таблицу с идентификаторами **ID_INDICATOR_CAPS**, **ID_INDICATOR_NUM** и **ID_INDICATOR_SCR**.

Удалять упомянутые индикаторы из строковой таблицы не потребуется. Вместо этого следует удалить ссылки на них в коде, выполняющем создание строки состояния.

Конструктор **CMainFrame** создает строку состояния, ссылаясь на массив строковых идентификаторов **static**, помещенный в верхней части файла MainFrm.cpp:

```
static UINT indicators[] =
{
      ID_SEPARATOR,                  // индикатор строки состояния
      ID_INDICATOR_CAPS,
      ID_INDICATOR_NUM,
      ID_INDICATOR_SCRL,
};
```

Если заключить выделенные строки в комментарий (или удалить их), строка состояния опускает индикаторы CapsLock, ScrollLock и NumLock.

Добавление собственных индикаторов

Добавление своих индикаторов — не настоль просто, как удаление встроенных индикаторов, однако оно же и не слишком сложно. В этом разделе мы займемся дополнением строки состояния индикатором ширины пера. Такой индикатор должен отображать ширину пера в логических единицах: 0, 4, 8, или 12.

Для помещения нового индикатора необходимо выполнить трехшаговый процесс:

- Добавьте новый идентификатор ресурса для индикатора.
- Добавьте в строковую таблицу ресурс, связанный с новым идентификатором. Длина этой строки используется при определении размеров панели индикатора, так что может потребоваться дополнить строку пробелами.
- Добавьте обработчик **ON_UPDATE_COMMAND_UI** для конкретного идентификатора строки. Это делается вручную — ClassWizard не сможет помочь при работе с ресурсом строковой таблицы.

Давайте выполним перечисленные шаги.

Добавление нового идентификатора ресурса

Во-первых, следует определить новый идентификатор ресурса. Windows присвоит вашему ресурсу номер, однако вы должны указать имя ресурса.

Необходимо выполнить следующие шаги:

1. Выберите View|Resource Symbols (Просмотр|Ресурсные символы). Как только откроется окно Resource Symbols (см. рис. 14.22), щелкните на New.

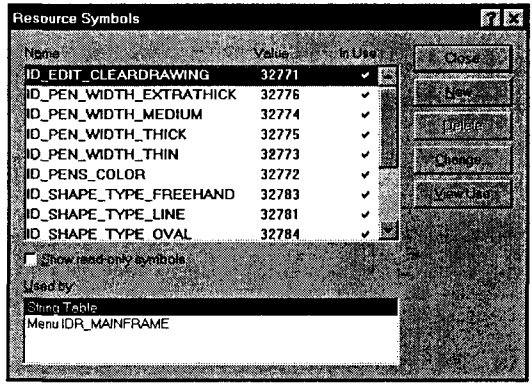


РИСУНОК 14.22.
Определение нового ресурсного символа в диалоговом окне Resource Symbols.

2. В диалоговом окне New Symbol (Новый символ) введите **"ID_STATUS_PEN_WIDTH"** и примите в качестве идентификатора ресурса значение, предлагаемое по умолчанию. После завершения операции закройте оба диалоговых окна.

Добавление ресурса в строковую таблицу

Второй шаг заключается в создании нового ресурса в строковой таблице. Все строковые константы, используемые в программе, можно сохранять в строковой таблице, а не встраивать их в код. В случае использования строковой таблицы появится возможность легко преобразовать интерфейс таким образом, чтобы можно было работать с различными языками, изменив всего лишь содержимое строковой таблицы.

Для помещения нового ресурса в строковую таблицу выполните следующие шаги:

1. Выберите панель ResourceView в окне Workspace, затем откройте папку String Table. Обратите внимание, что со строковой таблицей не связано ни одного идентификатора. Дважды щелкните на пиктограмме "abc", чтобы открыть String Table Editor (Редактор строковой таблицы), как показано на рис. 14.23.
2. Перейдите в конец строковой таблицы и дважды щелкните на последней (пустой) строке. В открывшемся диалоговом окне String Properties (Свойства строки) воспользуйтесь раскрывающимся списком для выбора **ID_STATUS_PEN_WIDTH** (только что созданного идентификатора), а затем введите для него текст по умолчанию. Добавьте в конец текста несколько пробелов, тем самым обеспечив место для строки, которая теперь будет отображаться на экране, поскольку панель индикатора во время выполнения расширяться не будет. Экран должен иметь вид, показанный на рис. 14.24.

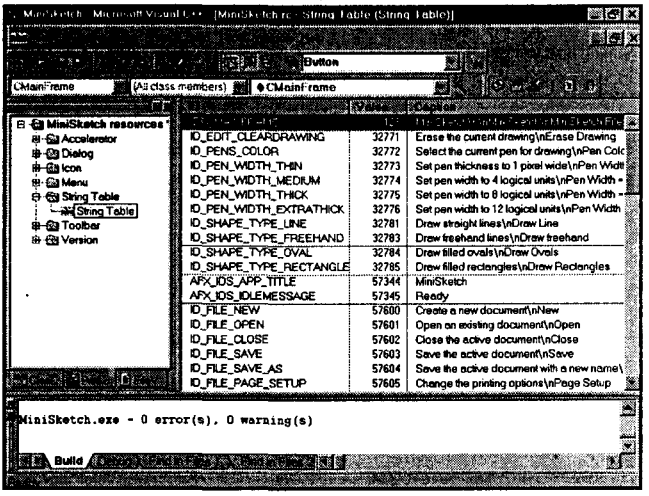


РИСУНОК 14.23.
Редактор строковой таблицы.

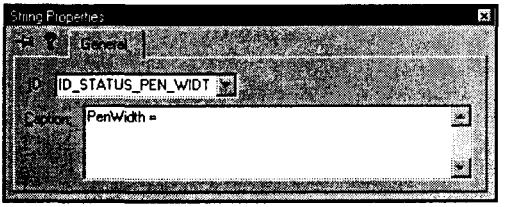


РИСУНОК 14.24.
Установка строкового значения для идентификатора ID_STATUS_PEN_WIDTH.

3. Измените определение массива **indicators** в файле CMainFrame.cpp. Введите **ID_STATUS_PEN_WIDTH** непосредственно за идентификатором **ID_SEPARATOR**.

После компиляции и запуска программы можно наблюдать новый индикатор. Он еще не производит никаких действий, даже если изменяется ширина пера — активизация индикатора представляет собой третий и последний шаг, который следует пройти.

Активизация индикатора

Для активизации индикатора потребуется всего лишь реализовать обработчик **ON_UPDATE_ COMMAND_UI**. Делается это вручную, поскольку ClassWizard не распознает идентификаторов, привязанных к ресурсам строковой таблицы.

К счастью, добавление обработчика выполняется достаточно просто. Все, что потребуется сделать, это:

1. В заголовок класса **CMSView** поместить следующий прототип:

```
afx_msg void OnUpdateUIPenWidthIndicator(CCmdUI * pCmdUI);
```

2. В карту сообщений класса **CMSView** включить следующую макрокоманду обработчика обновления индикатора (убедитесь в отсутствии точки с запятой в конце макрокоманды):

```
ON_UPDATE_COMMAND_UI(ID_STATUS_PEN_WIDTH,
   OnUpdateUIPenWidthIndicator)
```

3. Определить в качестве **CMSView** обработчик обновления индикатора с именем **OnUpdateUIPenWidthIndicator()**, реализация которого показана в листинге 14.5.

Листинг 14.5. Обработчик обновления индикатора OnUpdateUIPenWidthIndicator().

```
void CMSView::OnUpdateUIPenWidthIndicator(CCmdUI * pCmdUI)
{
      CString s;
      s.Format("Pen Width = %2d", m_PenWidth);
      pCmdUI->SetText(s);
}
```

Откомпилируйте и запустите приложение. Теперь при изменении ширины пера (то ли через меню, то ли с использованием акселераторов) индикатор функционирует, что демонстрирует рис. 14.25. Если число усекается, добавьте еще пробелы в строку, содержащуюся в строковой таблице.

Возможно, возникает вопрос, как макрокоманда **ON_UPDATE_COMMAND_UI** обновляет строку состояния, если последняя постоянно отображается на экране. В приложении с системой меню эта макрокоманда по приходу сообщения **WM_INITPOPUP** вызывает обработчик команд пользовательского интерфейса каждый раз, когда отображается меню. Что использует обработчик команд пользовательского интерфейса, когда он связан со строкой состояния?

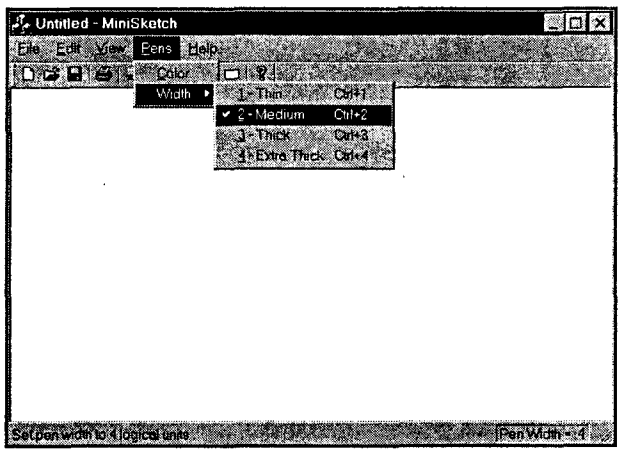


РИСУНОК 14.25. *Функционирующий индикаторширины пера в приложении Minisketch.*

Windows обращается к обработчикам строки состояния (равно как и к обработчикам команд панели инструментов) во время простоя. Состояние простоя имеет место, когда у приложения больше нет сообщений, требующих обработки, т.е. делать больше нечего. Может показаться, что это случается нечасто, тем не менее в действительности для большей части интерактивных приложений характерно наличие значительной доли времени простоя.

Однако, поскольку Windows вызывает обработчики панели инструментов и строки состояния во время простоя приложения, возможны ситуации, когда индикаторы строки состояния немного запаздывают. Такое никогда не имеет место в случае обработчиков команд меню.

Добавление индикатора цвета пера

Несмотря на то что встроенный класс **CStatusBar** поддерживает текстовые индикаторы, он не обеспечивает автоматическую поддержку графических индикаторов, которые как раз и нужны для отображения текущего цвета пера. Очень просто определить конкретный режим рисования форм — по нажатому состоянию одной из кнопок выбора формы. Имея в своем распоряжении индикатор ширины пера, можно четко сказать, какова текущая ширина пера. А что можно сказать в отношении цвета пера?

Конечно, можно было бы добавить еще один текстовый индикатор, который будет отображать значения RGB текущего цвета, однако это не принесет большой пользы. Неплохо иметь такой индикатор цвета пера, какой был реализован в программе PaintORama, причем желательно с улучшенными характеристиками. К счастью, реализовать это совсем несложно.

До сих пор вы в своей деятельности опирались на четыре класса, сгенерированные AppWizard. В них вносились необходимые изменения, добавлялись недостающие элементы данных и перекрывались виртуальные функции. Но чего мы еще *не делали — так это не создавали в проекте MiniSketch совершенно новый класс.* Что ж, приступим.

Когда AppWizard включает в проект строку состояния, он добавляет в класс **CMainFrame** защищенную переменную с именем **m_wndStatusBar**. Эта переменная относится к классу **CStatusBar**; она создается и связывается с главным рамочным окном в конструкторе **CMainFrame**.

Поскольку **m_wndStatusBar** представляет собой объект **CStatusBar**, а не подкласс, созданный специально для целей приложения, подобно **CMSView**, **CMSDoc**, **CMSApp** и **CMainFrame**, ни одну из виртуальных функций класса **CStatusBar** перекрыть нельзя. Хотя переменной **m_wndStatusBar** можно посылать сообщения, однако изменить то, как она работает, не представляется возможным. С другой стороны, если создать собственный подкласс класса **CStatusBar** и использовать его вместо **CStatusBar** при создании переменной **m_wndStatusBar**, то там можно вносить любые необходимые изменения. Именно этим мы и займемся, разработав качественный индикатор цвета пера в рамках строки состояния.

Создание класса CMSStatusBar

Вначале потребуется создать новый класс, а затем подключить его к приложению вместо класса **CStatusBar**. Первым делом будет создаваться класс **CMSStatusBar**, работающий точно так же, как и встроенный вариант. Далее в полученный класс вносятся все необходимые изменения.

Вот с чего следует начать:

1. Убедитесь, что проект MiniSketch открыт. Запустите ClassWizard (для этого нужно нажать Ctrl+W либо выбрать в меню команду View|ClassWizard). В диалоговом окне ClassWizard щелкните на Add Class, а затем в выпадающем меню выберите элемент New.
2. Введите **"CMSStatusBar"** в качестве имени класса и выберите из списка **CStatusBarCtrl** как базовый класс. (Фактически потребуется использовать **CStatusBar**, а не **CStatusBarCtrl**, но это исправится чуть позже.) Щелкните на OK, когда экран примет вид, показанный на рис. 14.26. Когда диалоговое окно ClassWizard отобразится вновь, закройте его.

3. Выберите панель ClassView в окне Workspace и дважды щелкните на классе **CMSStatusBar**. В заголовке этого класса измените ссылку **CStatusBarCtrl** на **CStatusBar**:

```
class CMSStatusBar : public
   CStatusBar
```

4. Откройте файл MSStatusBar.cpp и перейдите на определение карты сообщений для этого класса. Замените ссылку **CStatusBarCtrl** на **CStatusBar**. Исправленная карта сообщений должна выглядеть так:

```
BEGIN_MESSAGE_MAP(CMSStatusBar,
   CStatusBar)
      //
   {{AFX_MSG_MAP(CMSStatusBar)
      // ЗАМЕЧАНИЕ: Здесь ClassWizard включает и удаляет
      // макрокоманды отображения.
      //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

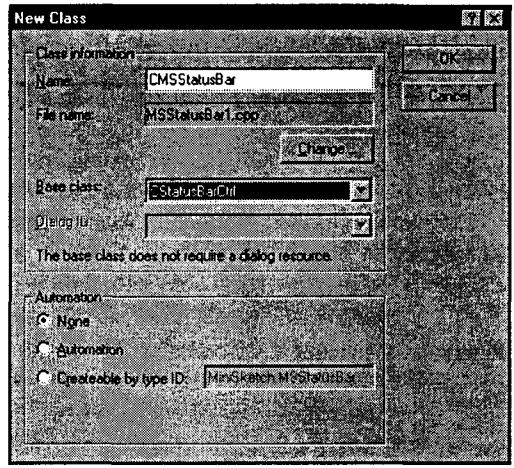


РИСУНОК 14.26. *Добавление нового класса CMSStatusBar.*

5. В панели ClassView откройте и разверните класс **CMainFrame**. Дважды щелкните на переменной **m_wndStatusBar**, чтобы перейти к определению переменной. В редакторе исходного кода (Source Code Editor) измените объявление переменной **m_wndStatusBar** так, чтобы упоминался **CMSStatusBar**, а не **CStatusBar**. Новое объявление должно приобрести вид:

```
CMSStatusBar m_wndStatusBar;
```

6. Включите ссылку на файл заголовков класса **CMSStatusBar** в файл заголовков, в котором объявлена переменная **m_wndStatusBar** (MainFrm.h). Просто добавьте перед определением класса **CMainFrame** следующую строку:

```
#include "MSStatusBar.h"
```

Откомпилируйте и запустите программу. Все должно работать точно так, как работало ранее.

Добавление индикатора цвета пера

Сейчас мы займемся помещением в строку состояния нового индикатора. Новый индикатор будет расположен после индикатора ширины пера. Индикаторы строки состояния представляют собой массив, индексы которого начинаются с 0, так что новый индикатор получит индекс 2 (этот индекс используется для доступа к индикатору).

Поскольку в намерения входит только закрашивание индикатора сплошным цветом, без вывода какого-либо текста, может показаться странным применение текстового индикатора, как это имело место в случае отображения ширины пера. Однако именно это и следует сделать.

Выполните приведенные ниже шаги (они в основном повторяют действия, совершаемые при построении индикатора ширины пера):

1. Используя команду меню View|Resource Symbols, определите новый ресурсный символ с именем **ID_STATUS_PEN_COLOR**. Примите идентификатор, предлагаемый по умолчанию.
2. Поместите в строковую таблицу новый строковый ресурс, воспользовавшись новым идентификатором. При определении нового строкового значения зарезервируйте 5–10 пробелов, тем самым обеспечив индикатору приемлемые размеры.
3. Включите новый идентификатор в массив **indicators**, определение которого находится в начале файла MainFrm.cpp. Новый массив **indicators** должен содержать следующих три строки:

```
static UINT indicators[] =
{
      ID_SEPARATOR,
   // индикатор строки
   // состояния
      ID_STATUS_PEN_WIDTH,
      ID_STATUS_PEN_COLOR,
};
```

Откомпилируйте и запустите программу на выполнение. Прежде чем двигаться дальше, убедитесь в том, что окно выглядит так, как показано на рис. 14.27.

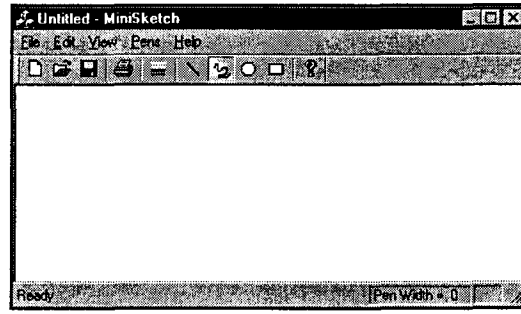


РИСУНОК 14.27. *Приложение MiniSketch после добавления индикатора цвета пера.*

Раскрашивание индикатора

Чтобы раскрасить индикатор цвета пера, потребуется перекрыть виртуальную функцию **DrawItem()** в классе **CMSStatusBar**. Windows вызывает **DrawItem()** для строки состояния по одному разу для каждой панели, которую необходимо перерисовать, передавая в функцию аргумент **LPDRAWITEMSTRUCT**. Для достижения нашей цели следует установить значения только трех полей этой структуры:

- ***itemID***. Определяет панель, подлежащую перерисовке. Поскольку индексы панелей в строке состояния начинаются с 0, для нашего индикатора **itemID** будет равно 2.
- ***rcItem***. Прямоугольник, содержащий "раскрашиваемую" область. Здесь определяются размеры индикатора.
- ***hDC***. Контекст устройства для строки состояния. Потребуется создать еще один объект **CDC**, а затем вызвать функцию **Attach()** — для его использования и **Detach()** — для освобождения.

В дополнение к перекрытию виртуальной функции **DrawItem()**, панели индикатора цвета пера необходимо придать стиль **SBT_OWNERDRAW**. Это делается за счет вызова метода **GetPaneInfo()** с целью получения базовой информации о панели, после чего выполняется вызов функции **SetPaneInfo()** с использованием операции поразрядного **OR** для придания панели стиля **SBT_OWNERDRAW**.

Для перекрытия функции **DrawItem()** следует выполнить такие шаги:

1. Поместите в класс **CMSStatusBar** следующее объявление:

```
virtual void DrawItem(LPDRAWITEMSTRUCT lpdis);
```

Вам не удается воспользоваться механизмом Add Virtual Function из ClassWizard, но зато можно применить диалоговое окно Add Member Function. Выберите в качестве возвращаемого типа **void** и отметьте флажок Virtual (Виртуальная). Не обращайте внимания ни на какие предупреждения, выдаваемые ClassWizard.

2. Включите в тело функции **DrawItem()** код, представленный в листинге 14.6.

Листинг 14.6. Виртуальная функция DrawItem() класса CMSStatusBar.

```
void CMSStatusBar::DrawItem(LPDRAWITEMSTRUCT lpdis)
{
// 1. Убедиться, что мы имеем дело с нужным элементом
      if (lpdis->itemID == 2)
      {
            // 2. Создать DC для рисования
            CDC dc;

            // 3. Ассоциировать его с DC lpdis
            dc.Attach(lpdis->hDC);

            // 4. Получить размеры прямоугольника, который требуется
            // раскрасить
            CRect rect(lpdis->rcItem);

            // 5. Раскрасить прямоугольник
            CBrush brush(RGB(255,0,0));   // Создать кисть красного
                                          // цвета
            dc.FillRect(rect, &brush);

            // 6. Отсоединить DC
            dc.Detach();

            // 7. Вернуться, нарисовав все
            return;
      }
      // Не наша панель? Пусть обработку выполнит CStatusBar
      CStatusBar::DrawItem(lpdis);
}
```

3. Присвоить панели цвета пера стиль **SBT_OWNERDRAW**, воспользовавшись функциями **GetPaneInfo()** и **SetPaneInfo()**. Поместить в функцию **CMainFrame::OnCreate()** следующие строки, непосредственно перед завершающим оператором **return**:

```
// Инициализировать пользовательскую строку состояния
UINT nID, nStyle;
int cx;
m_wndStatusBar.GetPaneInfo(2, nID, nStyle, cx);
m_wndStatusBar.SetPaneInfo(2, nID, nStyle |
   SBT_OWNERDRAW, cx);
```

Откомпилируйте и запустите завершенное приложение. Индикатор цвета пера должен быть закрашен в красный цвет — цвет, жестко закодированный в функции **DrawItem()**. Теперь останется лишь подключить индикатор таким образом, чтобы он использовал фактический цвет пера вместо жестко закодированной красной кисти.

Подключение переменной m_PenColor

В последнем шаге обеспечивается реакция на изменения значения переменной **m_PenColor**. Этот шаг труден, поскольку **m_PenColor** представляет собой приватную переменную класса **CMSView**, так что получить к ней доступ из класса **CMSStatusBar** нельзя. Если превратить переменную **m_PenColor** в **public** (весьма и весьма неудачная идея), возникнут большие сложности с соединением вашего объекта строки состояния с объектом представления.

По всей вероятности, наиболее простое решение заключается в создании копии переменной **m_PenColor** в классе **CMainFrame**, поскольку объект **CMainFrame** является *родительским окном* как для объекта представления, так и для объекта строки состояния. Если добавить в класс **CMainFrame** функции **SetPenColor()** и **GetPenColor()**, то и строка состояния, и представление смогут получить доступ к упомянутым функциям, воспользовавшись обращением к **GetParent()** для определения главного окна приложения.

Вот как это делается:

1. Добавьте приватную переменную **COLORREF** с именем **m_PenColor** в класс **CMainFrame**.
2. Определите в классе **CMainFrame** методы **SetPenColor()** и **GetPenColor()**, используя код из листинга 14.7. Функция **GetPenColor()** просто возвращает значение **CMainFrame::m_PenColor**. Функция **SetPenColor()** устанавливает значение **CMainFrame::m_PenColor**, а затем делает строку состояния недействительной (invalidate), в результате чего она будет перерисовываться всякий раз, когда выбирается новый цвет пера.

Листинг 14.7. Методы GetPenColor() и SetPenColor() класса CMainFrame.

```
COLORREF CMainFrame::GetPenColor()
{
      return m_PenColor;
}

void CMainFrame::SetPenColor(COLORREF color)
{
      m_PenColor = color;
      m_wndStatusBar.Invalidate();   // Перерисовать строку состояния
}
```

3. В функции **CMSStatusBar::DrawItem()** внесите изменения в строку, в которой создается красная кисть так, чтобы она приобрела вид

```
// 5. Раскрасить прямоугольник
CBrush brush((CMainFrame*)GetParent())->GetPenColor());
dc.FillRect(rect, &brush);
```

4. Поместите следующую строку в конец функции **CMSView InitPen()**:

```
((CMainFrame *)GetParent())->SetPenColor(m_PenColor);
```

5. При помощи ClassWizard добавьте новую функцию в класс **CMSView**, перекрыв унаследованную функцию **OnInitialUpdate()** кодом из листинга 14.8.

Листинг 14.8. Виртуальная функция CMSView::OnInitialUpdate().

```
void CMSView::OnInitialUpdate()
{
      CView::OnInitialUpdate();
      InitPen();
}
```

6. И наконец, поместите ссылку на файл заголовков MainFrm.h в начало каждого файла MiniSketchView.cpp и MSStatusBar.cpp после всех операторов **include**:

```
#include "MainFrm.h"
```

Откомпилируйте и запустите программу на выполнение. Всякий раз, когда цвет пера изменяется, эти изменения отображаются индикатором цвета пера в строке состояния.

На очереди другие работы

Итак, уважаемые, завершена еще одна большая глава. Можно перевести дух, однако успокаиваться не следует. Механизм **OnDraw()** еще сыроват — он, например, не обеспечивает выбора ширины и цвета пера. Кнопки выбора типа формы работают хорошо, но MiniSketch пока еще не может отрисовывать эти формы. А ведь еще есть печать! Даже не заикайтесь о печати.

Очевидно, что предстоит проделать большую работу, и наилучший способ приступить к ней — это попытаться выполнять обработку во время простоя. Windows-программы используют время простоя для выполнения тех важных рутинных работ, которые не делаются во время обработки сообщений. Итак, возьмите гамак и немного отдохните — завтра наступит очень скоро.

Глава 15

Сохранение в MiniSketch: работа с документами и файлами

В архитектуре "документ-представление" на класс документа возлагаются две обязанности. Во-первых, класс документа определяет метод сохранения данных программы в памяти. Во-вторых, класс определяет метод, с помощью которого программа сохраняет и получает данные, используя файлы на дисках и других второстепенных накопителях.

Предположим, что создается очень простая программа текстового редактора по технологии "документ-представление". Класс документа может хранить текст, используемый программой, несколькими способами:

- *Как двумерный массив символов.* Первым измерением будут строки текста, а вторым — отдельные символы.
- *Как одномерный массив* ***CString****.* Каждая строка текста будет сохранена в отдельном экземпляре **CString**, и каждый элемент массива будет строкой текста.
- *Как сплошной блок в памяти.* Конец каждой строки будет помечать символ новой строки.
- *Как двусвязный список массивов байтов.*

Варианты могут занять несколько страниц. При создании класса документа следует решить, каким образом его данные будут сохраняться в памяти. У каждого метода есть как преимущества, так и недостатки — необходимо определить, какая структура максимально подходит для поставленной задачи.

И это — лишь половина проблемы. Еще потребуется выбрать способ сохранения данных на диске, а также преобразования между постоянной, хранимой на диске, формой документа и находящейся в памяти его активной формой. И вновь вариантов решения несколько. Можно было бы сохранить данные в обычном текстовом файле, разделяя строки текста символами новой строки. Можно было бы записать данные как двоичный образ, который легко прочитать в память как единое целое. Несложно придумать альтернативную форму, объединяющую сжатие, шифрование или любые другие особенности дизайна.

При проектировании класса документа способ его представления в памяти часто влияет на выбор метода долгосрочного вторичного хранения. К примеру, если решено хранить текстовые данные в динамически распределяемом связанном списке **CString**, не будет возможности использовать двоичный метод поблочной записи на диск — при чтении обратно в память указатели списка уже не будут ссылаться на правильные адреса памяти.

При работе над классом документа MiniSketch **CMSDoc** будем придерживаться следующего плана:

- Начать следует с создания *элементарных классов* — классов, хранящих данные документа. Для программы MiniSketch будет разработана иерархия классов **Shape**, предназначенная для хранения одной линии, прямоугольника или овала.
- Затем мы рассмотрим поддержку в MFC *классов коллекций* — классов, предназначенных для сохранения больших объемов элементарных объектов.
- В заключение мы исследуем поддержку *сериализации* в MFC — метода сохранения и чтения экземпляров классов с диска. Тут же мы обсудим поддержку в MFC POFH — Plain Old File Handling*. Если сериализация покажется слишком сложной и запутанной, вы можете прийти к заключению, что система простого сохранения файлов значительно упростит программу. К счастью, обработка файлов в MFC даже проще, чем в C.

Итак, начнем работу с рассмотрения основных элементов, управляемых классом **CMSDoc**.

Точки и фигуры

При размышлениях над текстовым редактором, без труда определяется основной элемент документа: символ. Не так ли? В зависимости от приложения можно было бы еще отслеживать предложения, абзацы и разделы. А если разрабатывается текстовый процессор, а не простой текстовый редактор, также потребуется управлять шрифтами и информацией о форматировании.

С программой MiniSketch вы попадаете в подобное затруднительное положение. На первый взгляд сохранение точек кажется адекватным перерисовке изображений с чтением и записью на диск. Но как только вы начнете добавлять толщину пера и цвет (не говоря уже о различных типах фигур, для которых создавались кнопки в главе 14), простого сохранения точек окажется недостаточно. В этом случае следует построить более сложный документ.

Классы и атрибуты

Давайте подумаем, какую информацию должна сохранять MiniSketch:

- *Кривые (линии от руки).* Вы уже знаете, что кривые можно хранить в виде массива **CPoint**. Но сейчас необходимо сохранять также цвет и толщину пера для каждой кривой. Для сохранения кривой будет создан класс **Squiggle**.
- *Прямые линии.* Объект прямой намного проще объекта **Squiggle**. В объекте **Line** требуется сохранить две конечные точки линии, ширину пера и цвет.
- *Прямоугольники и овалы.* Объекты **Box** и **Oval** должны сохранять размеры ограничивающего прямоугольника, используемого для рисования фигуры, и перо, используемое для рисования ее контура. Кроме того, в связи с тем, что прямоугольники и овалы являются сплошными объектами, объекты также должны сохранять кисть, используемую для заливки внутренней области фигуры.

Создать иерархию классов, соответствующую этим требованиям, довольно легко. Поскольку каждому классу необходим доступ к толщине и цвету пера, можно создать базовый класс (назовем его **Shape**), содержащий только эти данные.

Кривые и отрезки являются разновидностями **Shape**, так что для создания производных от **Shape** классов **Line** и **Squiddle**, можно использовать прямое наследование (**public**). Каждый объект **Line** содержит данные о своих конечных точках, тогда как каждый объект **Squiggle** содержит массив объектов **CPoint**, что более походит на класс **CMSDoc** на данной стадии его проектирования.

В связи с тем что классы **Box** и **Oval** являются разновидностями сплошных фигур, им необходимы одинаковые элементы данных: атрибуты кисти, используемой для заливки внутренней части фигуры, и вершины ограничивающего прямоугольника. Но если **Box** и **Oval** имеют одинаковые элементы данных, чем же они тогда различаются? При запросе на визуализацию объект **Box** создает прямоугольник, а объект **Oval** — эллипс. Следовательно, из-за различия в поведении, и несмотря на идентичность данных, создается не один, а два отдельных класса — **Box** и **Oval**, унаследованные от **FilledShape**.

Класс **FilledShape**, в свою очередь, унаследован от **Shape** и содержит параметры кисти, используемой для заполнения фигуры. В классе **FilledShape** не следует сохранять размеры ограничивающего прямоугольника, поскольку позже может потребоваться создать нового наследника **FilledShape**, скажем, неправильную область или многоугольник, которым ограничивающий прямоугольник не нужен.

Окончательная иерархия классов **Shape** показана на рис. 15.1.

Поведение фигур

В предыдущем разделе было принято решение о том, что объекты в иерархии классов **Shape** должны отображать себя самостоятельно. Мы реализуем это требование, добавив к классу **Shape** виртуальную функцию **Draw()**, которая будет перекрываться в каждом производном классе. Функция **Draw()** будет принимать в качестве параметра контекст устройства, и, следовательно, будет полностью самостоятельной. Таким образом объекты **Box**, **Squiggle** и **Oval** смогут соответствующим образом визуализироваться.

И это еще не все! Функция **Draw()** — виртуальная, и следовательно, можно использовать полиморфизм, что существенно упростит логику рисования в программе. Вот как это делается.

В случае создания массива (или списка) указателей на **Shape**, как показано на рис. 15.2, C++ позволяет установить каждый их этих указателей на объект класса **Shape** или на *любой другой объект класса, порожденного от* ***Shape***.

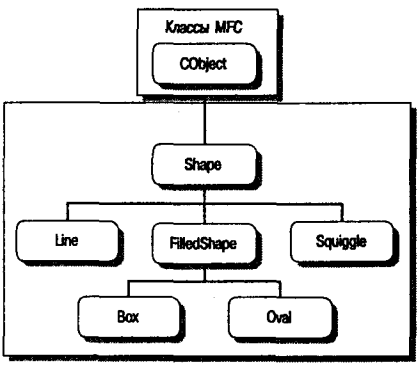


РИСУНОК 15.1. *Иерархия классов Shape.*

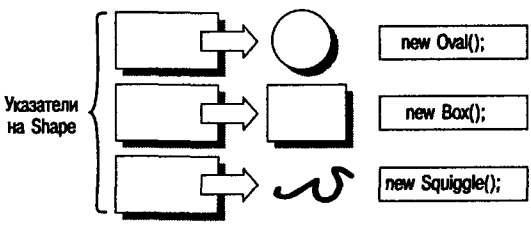


РИСУНОК 15.2. *Полиморфные указатели на Shape.*

Следовательно, показанный ниже код будет работать:

```
Shape * list[3];   // массив из трех указателей на Shape
list[0] = new Squiggle(...);
list[1] = new Oval(...);
list[2] = new Box(...);
```

Этот код не только корректен — он поразительно удобен; его можно использовать для реализации позднего связывания и полиморфизма в C++. Если по указателям **list[0]**, **list[1]** и **list[2]** обратиться к виртуальной функции, подобной **Draw()**, вызываются функции **Draw()** производных классов, а не класса **Shape**. Вот продолжение предыдущего примера:

```
list[0]->Draw();     // Вызывает Squiggle::Draw();
list[1]->Draw();     // Вызывает Oval::Draw();
list[2]->Draw();     // Вызывает Box::Draw();
```

Для корректности этого кода должны соблюдаться два условия. Во-первых, массив **list** должен содержать *указатели* на объекты базового класса, а не сами объекты базового класса. Во-вторых, функцию **Draw()** в базовом классе следует

объявить виртуальной — иначе код будет вызывать функцию **Draw()** класса **Shape** вместо функции **Draw()** классов **Line**, **Box**, **Squiggle** и **Oval**.

Определение классов фигур

Если вы еще раз взглянете на рис. 15.2, то увидите, что класс **Shape** порожден от базового класса MFC **CObject**. Это не жесткое требование. В данном случае мы просто хотим использовать предоставляемую классом **CObject** встроенную поддержку сериализации и динамического создания объектов. С другой стороны, использование класса **CObject** накладывает некоторые ограничения на способ объявления классов, как вскоре будет показано. К счастью, эти ограничения не особенно обременительны, так что нет смысла *не* наследовать от **CObject**.

Создание файлов классов

Сперва создадим основной набор файлов классов документов. После создания "каркасных" файлов их можно изменить, добавив требуемые характеристики и возможности.

Вместо создания файлов заголовков и реализации для каждого файла в отдельности (т.е. Shape.h, Oval.h, Squiggle.h и т.д.), все объявления классов помещаются в Shape.h, а все реализации методов — в Shape.cpp. Чтобы еще больше все упростить, объявления можно добавить к файлам, используемым классом **CMSDoc**. Умение создавать новые самостоятельные классы вам, естественно, пригодится, так что принимайтесь за работу.

Итак, вот шаги, которые необходимо выполнить:

1. Откройте проект MiniSketch. Из всплывающего меню WizardBar, показанного на рис. 15.3, выберите New Class либо выполните команду меню Insert|New Class.
2. В диалоговом окне New Class выберите тип класса Generic Class. (Эта опция недоступна, если диалоговое окно New Class вызывается из ClassWizard, а не из WizardBar.) Назовите класс **Shape** и породите его от **CObject** с использованием прямого наследования. В заключении диалоговое окно New Class должно выглядеть, как на рис. 15.4. Щелкните на OK.
3. Если появится предупреждающее сообщение, показанное на рис. 15.5, щелкните на OK. В связи с тем, что создан общий класс, а не класс MFC, ClassWizard не знает, где искать соответствующие заголовочные файлы. Вскоре этот вопрос будет решен.
4. Выберите в окне Workspace закладку ClassView и дважды щелкните на находящемся в ней новом классе **Shape**. Добавьте следующую строку непосредственно перед объявлением класса **Shape**, созданного ClassWizard (т.е. перед строкой, которая начинается с **class Shape**):

```
#include "stdafx.h"
```

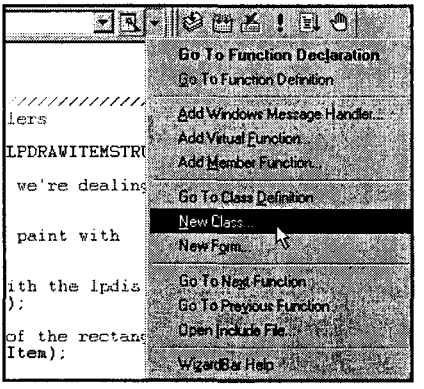


РИСУНОК 15.3. *Добавление нового класса из всплывающего меню WizardBar.*

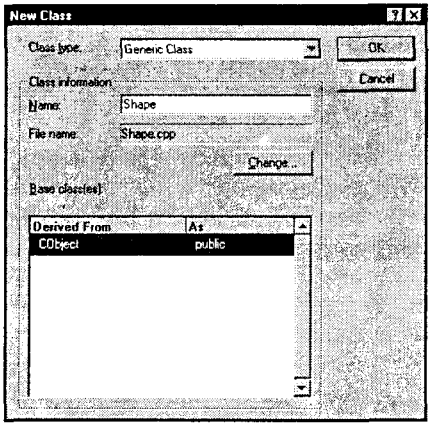


РИСУНОК 15.4. *Добавление класса Shape.*

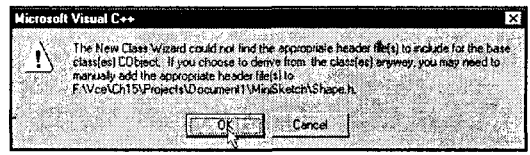


РИСУНОК 15.5. *Окно сообщения "Заголовок не найден".*

Эта строка исправит ошибку, упоминаемую в шаге 3. Откомпилируйте программу, просто чтобы убедиться, что все работает нормально.

СОВЕТ

Почему не проходит компоновка?

Если программа компилируется без ошибок, но после добавления класса Shape выдается ошибка связывания, повторно соберите проект, выбрав в меню Build|Rebuild All.

Visual C++ при создании проектов обычно использует инкрементное связывание, т.е. перекомпоновывается только код, который был изменен. Иногда, особенно после добавления новых классов, инкрементный компоновщик "выпадает в осадок". После выбора Build|Rebuild All Visual C++ повторно создает проект без старых ошибок.

Теперь у вас имеются файлы каркаса, необходимые для реализации каждого из производных классов. Сперва будет разрабатываться класс **Shape**, для которого с помощью ClassWizard создаются необходимые каркасные функции. Затем работа продолжится над производными классами.

Для каждого класса создаются два листинга. Объявления классов должны помещаться в файл Shape.h после объявления класса **Shape**, но до директивы **#endif**. Реализации методов добавляются в файл Shape.cpp.

Объявление класса Shape

Во время реализации классов потребуется решить, какие переменные им необходимы. При проектировании иерархии классов **Shape** мы определили, что все потомки **Shape** требуют перо; итак, класс **Shape** имеет элементы данных толщины и цвета пера. Назовем эти переменные **m_penColor** и **m_penWidth** и сделаем их защищенными (**protected**), чтобы потомки смогли получать к ним доступ без необходимости прибегать к функциям-посредникам.

Стандартный конструктор **Shape** нужен только механизму сериализации **CObject**, так что он не должен быть общедоступным (**public**). Перенесем его в раздел **protected**. Классу необходим конструктор для установки значений **m_penColor** и **m_penWidth**, так что добавим в раздел **public** еще один конструктор.

Класс **Shape** должен перекрыть виртуальную функцию **CObject::Serialize()**. Чтобы **Serialize()** заработала, в объявлении класса необходимо вызвать макрокоманду **DECLARE_SERIAL()**. (Не ставьте точки с запятой после макроса **DECLARE_SERIAL()**!) В заключение классу **Shape** необходимы две виртуальные функции: **Draw()** и **Update()**. Функция **Draw()** возвращает **void** и принимает в своем единственном параметре **CDC***; функция **Update()** также возвращает **void**, а в параметре принимает **CPoint**. Окончательное объявление класса **Shape** показано в листинге 15.1. Измените Shape.h так, чтобы он соответствовал приведенному здесь объявлению.

Листинг 15.1. Объявление класса Shape.

```
class Shape : public CObject
{
      DECLARE_SERIAL(Shape)
public:
      Shape(COLORREF color, int width);
      virtual ~Shape();
      virtual void Draw(CDC * pDC);
      virtual void Update(CPoint point);
      virtual void Serialize(CArchive &ar);
protected:
      Shape();

      COLORREF       m_PenColor;
      int            m_PenWidth;
};
```

Объявление класса Line

Объявление класса **Line** в целом повторяет объявление класса **Shape**. В нем содержатся те же функции, так что начать можно с копирования объявления класса **Shape** и последующей замены **Shape** на **Line**.

Затем внесите следующие изменения:

1. Убедитесь, что класс **Line** порожден от **Shape**, а не от **CObject**.
2. Удалите две переменных **m_PenColor** и **m_PenWidth**. **Line** унаследует их от **Shape**.
3. Добавьте две переменных типа **CPoint** для фиксирования конечных точек линии. Назовите их **m_LineStart** и **m_LineEnd** и сделайте защищенными.
4. Добавьте к рабочему конструктору два дополнительных параметра типа **CPoint** — **start** и **end**.

Окончательное объявление класса **Line** приведено в листинге 15.2. Добавьте его в заголовочный файл Shape.h, следом за объявлением класса **Shape**.

Листинг 15.2. Объявление класса Line.

```
// Класс Line
class Line : public Shape
{
      DECLARE_SERIAL(Line)
public:
      Line(COLORREF color, int width,
            CPoint start, CPoint end);
```

```
        virtual ~Line();
        virtual void Draw(CDC * pDC);
        virtual void Update(CPoint point);
        virtual void Serialize(CArchive &ar);

protected:
        Line();

        CPoint   m_LineStart;
        CPoint   m_LineEnd;
};
```

Объявление класса Squiggle

Squiggle и **Shape** отличаются лишь используемыми переменными. Скопируйте объявление класса **Shape** и замените все упоминания о **Shape** на **Squiggle**. Затем внесите следующие изменения:

1. Измените родительский класс с **CObject** на **Shape**.
2. Удалите два элемента данных, добавьте массив **CArray** объектов **CPoint** и назовите его **m_data**. Каждый объект **Squiggle** будет хранить массив точек.
3. Добавьте в конструктор третий параметр, который представляет начальную точку **Squiggle**. Установите его тип в **CPoint** и назовите **start**.

Окончательное объявление класса **Squiggle** показано в листинге 15.3. Добавьте его в заголовочный файл класса **Shape**.

Листинг 15.3. Объявление класса Squiggle.

```
// Класс Squiggle
class Squiggle : public Shape
{
        DECLARE_SERIAL(Squiggle)
public:
        Squiggle(COLORREF color, int width,
                        CPoint start);
        virtual ~Squiggle();
        virtual void Draw(CDC * pDC);
        virtual void Update(CPoint point);
        virtual void Serialize(CArchive &ar);

protected:
        Squiggle();

        CArray<CPoint, CPoint> m_data;
};
```

Объявление класса FilledShape

Классы **Box** и **Oval** не являются прямыми потомками класса **Shape**. Вместо этого они унаследованы от класса **FilledShape**.

Поскольку объекты **FilledShape** используют кисть, **FilledShape** определяет информацию, необходимую для создания кисти. С целью сокращения размеров проекта MiniSketch класс **FilledShape** содержит только переменную типа **COLORREF**, определяющую сплошные кисти. Впрочем, вы без труда сможете добавить информацию, необходимую для создания других видов кистей.

При сборке класса **FilledShape** следуйте той же процедуре, что и для **Line** и **Squiggle**.

1. Скопируйте определение класса **Shape** и замените все вхождения **Shape** на **FilledShape**.
2. Смените базовый класс с **CObject** на **Shape**.
3. Замените переменные класса **Shape** одной переменной **COLORREF** с именем **m_BrushColor**.
4. Добавьте в конструктор третий аргумент **brushColor**, предназначенный для инициализации переменной **m_BrushColor**.

Объявление класса **FilledShape** приведено в листинге 15.4. Добавьте его в заголовочный файл класса **Shape**.

Листинг 15.4. Объявление класса FilledShape.

```
// Класс FilledShape
class FilledShape : public Shape
{
      DECLARE_SERIAL(FilledShape)
public:
      FilledShape(COLORREF color, int width,
                  COLORREF brushColor);
      virtual ~FilledShape();
      virtual void Draw(CDC * pDC);
      virtual void Update(CPoint point);
      virtual void Serialize(CArchive &ar);

protected:
      FilledShape();

      COLORREF      m_BrushColor;
};
```

Объявления классов Box и Oval

Два последних класса в иерархии классов **Shape** имеют идентичные объявления, различающиеся только именем. (Несмотря на это, каждый класс *ведет себя* по-своему, как и обсуждалось ранее.) Далее будет создаваться объявление класса **Box**.

Для создания класса **Box** следуйте приведенным ниже шагам:

1. Скопируйте объявление класса **Shape** и замените все вхождения **Shape** на **Box**, как имело место для предыдущих классов.
2. Замените потомка класса с **CObject** на **FilledShape**. (Заметьте, что этот шаг отличается от аналогичного в предыдущем объявлении класса.)
3. Замените элементы данных **Shape** двумя переменными **m_ULCorner** и **m_LRCorner** типа **CPoint**. (Они будут содержать, соответственно, верхнюю правую и нижнюю левую границы фигур.)
4. Добавьте в конструктор три дополнительных параметра: **COLORREF brushColor**, передающийся **FilledShape** при создании объекта **Box**, а также **CPoint ulc** и **CPoint lrc**, инициализирующие элементы данных **Box**.

Окончательное объявление класса **Box** показано в листинге 15.5. Создание класса **Oval** выполняется аналогично с тем лишь отличием, что каждый экземпляр **Box** заменяется на **Oval**. (Можно также скопировать объявление **Box**, выделить текст

копии и воспользоваться командой меню Edit|Replace для замены **Box** на **Oval** в выделенном тексте.) Оба объявления поместите в заголовочный файл **Shape.h**.

Листинг 15.5. Объявление класса Box.

```
// Класс Box
class Box : public FilledShape
{
      DECLARE_SERIAL(Box)
public:
      Box(COLORREF color, int width,
            COLORREF brushColor,
            CPoint ulc, CPoint lrc);
      virtual ~Box();
      virtual void Draw(CDC * pDC);
      virtual void Update(CPoint point);
      virtual void Serialize(CArchive &ar);

protected:
      Box();

      CPoint   m_ULCorner;
      CPoint   m_LRCorner;
};
```

Реализация классов фигур

Теперь, создав заголовочный файл, перейдем к файлу реализации. ClassWizard уже создал необходимый файл, **Shape.cpp**, и в нем потребуется написать методы, объявленные в **Shape.h**.

Вместо того чтобы привести длинный листинг кода, мы будем последовательно работать над дополнением **Shape.cpp**. Сперва будут создаваться пустые функции (иногда называемые *заглушками*), чтобы убедиться в работоспособности основной структуры. Затем на протяжении этой и следующей глав заглушки заменяются реальными функциями.

Создание заглушек в Snape.cpp

Вот инструкции по созданию пустых функций:

1. Убедитесь, что проект MiniSketch открыт. Выберите в окне Workspace панель ClassView и дважды щелкните на классе **Shape**, чтобы открыть окно редактора исходного кода для Shape.h.
2. Выделите мышью объявление класса в Shape.h. Скопируйте его в буфер обмена с использованием Edit|Copy или по нажатию Ctrl+C (см. рис. 15.6).
3. Откройте файл Shape.cpp в панели FileView. Выделите мышью две функции, уже существующие в этом файле, а затем используйте Edit|Paste или нажмите Ctrl+V для копирования объявления класса из Shape.h в Shape.cpp. (Не обращайте внимания на то, что панель ClassView теперь содержит дубликаты каждого из классов **Shape** — это будет устранено на следующем шаге.)
4. Отредактируйте файл Shape.cpp, удалив все добавленное, за исключением прототипов методов, конструкторов и деструкторов. Добавьте оператор разрешения области видимости перед каждым именем функции, затем удалите все вхождения слова **virtual**. (Функция определяется виртуальной в

объявлении класса, а не в реализации функции.) В листинге 15.6 показано, как выглядит этот код для класса **Line**; другие классы выглядят сходным образом. Поместите реализацию для **Shape**, **Line**, **Squiggle**, **FilledShape**, **Box** и **Oval**.

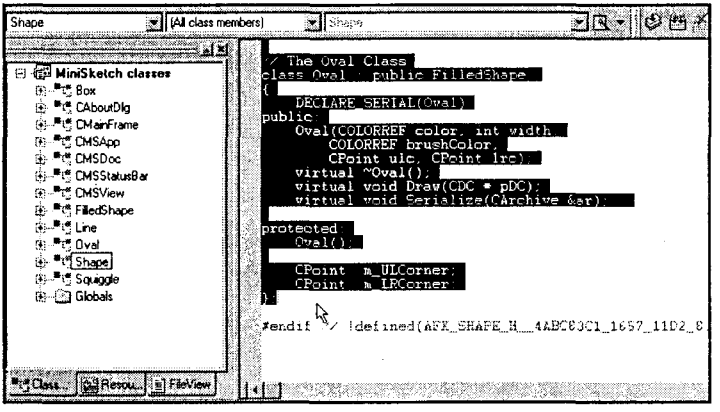


РИСУНОК 15.6.
Копирование объявлений класса Shape.

Листинг 15.6. Методы класса Line после редактирования.

```
// Класс Line
Line::Line(COLORREF color, int width, CPoint start,
   CPoint end);
Line::~Line();
void Line::Draw(CDC * pDC);
void Line::Update(CPoint point);
void Line::Serialize(CArchive &ar);
Line::Line();
```

5. После прототипа каждой функции удалите точку с запятой и добавьте пару фигурных скобок. В листинге 15.7 показано, как выглядит результат для класса **Line**; все остальные классы выглядят похоже.

Листинг 15.7. Класс Line после добавления пустых тел функций.

```
// Класс Line
Line::Line(COLORREF color, int width, CPoint start,
   CPoint end){ }
Line::~Line(){ }
void Line::Draw(CDC * pDC){ }
void Line::Update(CPoint point) { }
void Line::Serialize(CArchive &ar){ }
Line::Line(){ }
```

6. Добавьте в конец файла строки, приведенные в листинге 15.8. Эти макрокоманды добавляют поддержку создания каждого из объектов **Shape** с использованием сериализации.

Листинг 15.8. Макрокоманды сериализации для класса Shape.

```
// Макрокоманды сериализации
IMPLEMENT_SERIAL(Shape, CObject, 1)
IMPLEMENT_SERIAL(Line, Shape, 1)
```

```
IMPLEMENT_SERIAL(Squiggle, Shape, 1)
IMPLEMENT_SERIAL(FilledShape, Shape, 1)
IMPLEMENT_SERIAL(Box, FilledShape, 1)
IMPLEMENT_SERIAL(Oval, FilledShape, 1)
```

7. Запишите для каждого из классов рабочие конструкторы. За исключением **Squiggle**, ни один из конструкторов не нуждается в теле. Для начальной установки значений каждой из переменных класса можно воспользоваться синтаксисом инициализации. Окончательный вид конструкторов показан в листинге 15.9.

Листинг 15.9. Конструкторы иерархии классов Shape.

```
// Класс Shape
Shape::Shape(COLORREF color, int width)
      : m_PenColor(color), m_PenWidth(width) { }

// Класс Line
Line::Line(COLORREF color, int width, CPoint start, CPoint end)
      : Shape(color, width), m_LineStart(start), m_LineEnd(end) { }

// Класс Squiggle
Squiggle::Squiggle(COLORREF color, int width, CPoint start)
      : Shape(color, width)
{
      m_data.Add(start);
}

// Класс FilledShape
FilledShape::FilledShape(COLORREF color, int width,
   COLORREF brushColor)
      : Shape(color, width), m_BrushColor(brushColor) { }

// Класс Box
Box::Box(COLORREF color, int width, COLORREF brushColor,
            CPoint ulc, CPoint lrc)
      : FilledShape(color, width, brushColor),
        m_ULCorner(ulc), m_LRCorner(lrc) { }

// Класс Oval
Oval::Oval(COLORREF color, int width, COLORREF brushColor,
                CPoint ulc, CPoint lrc)
      : FilledShape(color, width, brushColor),
        m_ULCorner(ulc), m_LRCorner(lrc){ }
```

После добавления конструкторов и макрокоманд сериализации в Shape.cpp можно откомпилировать свой проект. Естественно, ни один из объектов **Shape** пока не используется — это будет нашим следующим шагом.

Использование классов иерархии Shape

Прежде чем начать строчить код, вернемся чуть назад и спланируем использование созданных классов **Shape**, что поможет избежать тупиковых ситуаций.

Вот общий план для классов **Shape**:

- Когда пользователь начинает рисование по нажатию левой кнопки мыши, в куче создается новый объект **Shape**. Состояние переменной **m_ShapeType** определяет, какой из объектов **Shape** необходимо создать.

- При перемещении мыши вызывается виртуальный метод **Update()** объекта **Shape**, чтобы сообщить объекту информацию о текущей позиции мыши. Для визуализации последней версии объекта **Shape** вызывается метод **Draw()**.
- Когда пользователь завершает рисование, отпуская кнопку мыши, текущий объект **Shape** сохраняется в классе документа.

Этот план кажется довольно простым, но есть некоторые вещи, на которые следует обратить особое внимание:

- Объекты **Shape** могут создаваться как классом документа, так и классом представления. Класс представления создает объекты **Shape** *явно* во время рисования, тогда как класс документа создает эти объекты *неявно* при сериализации.
- Класс документа отвечает за освобождение памяти, используемой всеми объектами **Shape**, независимо от того, кем они были созданы — классом документа или классом представления.
- Класс документа не может просто сохранять объекты, как считалось до сих пор. Вместо этого будут сохраняться *указатели на объекты*. (Если в объектах **Shape** применяется **CArray**, вся полиморфная информация, касающаяся объектов **Squiggle** и **Box**, исчезает после сохранения объекта **Shape** в документе.)

Это самый верхний уровень абстракции нашей задачи. В следующей главе мы поработаем над улучшением класса представления MiniSketch. А сейчас давайте переключимся на класс документа. Какие классы необходимы для сохранения рисунков? Для ответа на этот вопрос обратимся к *классам коллекций* MFC.

Коллекции MFC

Классы коллекций предназначены для хранения объектов — для коллекционирования их, если хотите. Возьмем, к примеру, класс **Array**. При создании **Array** всего лишь предоставляет пространство для хранения объектов одного типа. **Array** не обращает внимания на вид сохраняемых в нем данных, обеспечивая элементы одного вида.

Поведение связного списка похоже на поведение массива. Класс не интересуется видом хранимых данных — он просто предоставляет функции для добавления, удаления и поиска элементов. Подобные классы, называемые классами коллекций, обеспечивают управление данными в соответствии с конкретным способом.

Просмотр оперативной справки по слову *collections* может вызвать шоковое состояние. Вместо основных классов структур данных, т.е. связных списков, векторов, хэш-таблиц и бинарных деревьев, имеет место поразительный ассортимент — около двух дюжин классов, часть из которых выглядит идентично. Как узнать, какой из них использовать?

К счастью, ситуация не столь хаотична, как кажется на первый взгляд. 23 класса коллекций сгруппированы, что существенно упрощает выбор подходящего контейнера. Давайте рассмотрим их.

Нешаблонные (основанные на наследовании) классы

Первые версии библиотеки MFC были разработаны еще до включения шаблонов в язык C++. Таким образом, первые классы коллекций MFC были основаны на наследовании и предназначались для хранения объектов определенного типа

данных, скажем, **byte** или **int**. К этой ранней нешаблонной группе относится 17 классов коллекций MFC.

Классы, основанные на наследовании, — **CObArray** и **CObList** — хранят *указатели на объекты*, и эти объекты обязательно должны быть производными от **CObject**. Коллекции, основанные на наследовании, не являются *безопасными к преобразованию типов*. Компилятор не проверяет типы данных указателей на объекты, сохраняемые в массиве — достаточно того, чтобы они были унаследованы от **CObject***.

Рассмотрим пример. Предположим, что требуется хранить объекты **Line** и **Squiggle** в **CObArray**. Вот как это делается:

```
CObArray ar;                                    // Создаем массив объектов
Line * pLine = new Line(...);                   // Создаем новый Line
Squiggle * pSquiggle = new Squiggle(...);// Создаем новый Squiggle
ar.Add(pLine);                                  // Добавляем Line в массив
ar.Add(pSquiggle);                              // Добавляем Squiggle в
                                                   массив
```

Пока все выглядит отлично. Впрочем, получение информации из массива будет несколько сложнее. Если используется индексный доступ в **CObArray**, класс массива не вернет указатель на **Line** или **Squiggle** — вместо этого вы получите указатель на **CObject**. До использования необходимо преобразовать тип результата:

```
pSquiggle = (Squiggle *) ar[0];     // Получить из массива объект
                                       Squiggle
pLine = (Line *) ar[1];             // Получить из массива объект
                                       Line
pSquiggle->Draw(pDC);               // Нарисовать Squiggle
pLine->Draw(pDC);                   // Нарисовать Line
```

Внимательные читатели заметят в этом коде ошибку, связанную с неуклюжим приведением типов. При записи объекта **Line** в массив, он сохранялся в нулевом элементе, а объект **Squiggle** — в первом. При *выборке* объектов мы перепутали индексы, сославшись на объект **Squiggle** при помощи указателя на **Line**, и наоборот.

В связи с необходимостью приведения указателя, возвращаемого **CObArray**, компилятор не сможет выявить ошибку. Как-никак, приведения *предназначены* для указания компилятору не вмешиваться в их работу.

Коллекции, основанные на шаблонах

Шесть остальных классов являются классами шаблонов, и предоставляют таким образом безопасный к типам доступ к данным. Как правило, необходимо использовать как раз шаблонные классы.

Однако, шаблонным классам тоже присущи свои недостатки. При создании массива или списка одного из шаблонных классов коллекций, шаблон незаметно создает абсолютно новый класс, основанный на типах аргументов, используемых для реализации шаблона. Таким образом, при объявлении в программе шести объектов **CArray**, каждый из которых будет сохранять свой вид объектов, компилятор создаст шесть *различных* классов. Если же для создания массива применяется **CObArray**, программа будет содержать код только одного класса.

Шесть шаблонных классов MFC представлены в трех формах с двумя вариантами для каждой: классом, хранящим объекты, и классом, хранящим указатели на объекты. Вот эти три формы:

- *Массивы (arrays)* — подобно массивам, встроенным в большинство языков программирования, шаблонные классы MFC выполняют автоматическое увеличение при добавлении элемента. Доступ к отдельным членам коллекции производится через целочисленный индекс. Класс **CArray** хранит объекты, тогда как **CTypedPtrArray** — указатели на объекты.
- *Списки (lists)* — реализуемые в виде двусвязных списков классы **CList** (для простых объектов) и **CTypedListPtr** (для указателей) предоставляют неиндексированную структуру данных, пройтись по которой можно лишь последовательно. Можно перемещаться вперед и назад, но нельзя получить доступ к произвольно выбранному элементу. Списки подходят для использования в тех случаях, когда количество данных варьируется в широких пределах.
- *Карты (maps)* — иногда называемые *словарями*, карты работают в соответствии со своим названием. В обыкновенном словаре вы ищете слово (называемое *ключом*) и получаете определение (называемое *значением*). Карты позволяют сохранять любой вид значений и получать их с помощью любого вида ключа. Однако, при создании коллекции, потребуется определить типы ключа и значения. Если создается карта телефонной книги по именному ключу типа **CString**, карта не позволит получать или сохранять по ключу в виде телефонного номера. Двумя классами карт в MFC являются **CMap** и **CTypedPtrMap**.

Возможность сохранения в Minisketch

Итак, что же использовать для сохранения данных в классе **CMSDoc**? Начнем с исключения из списка кандидатур нешаблонных классов. Вам необходима поддержка компилятора для гарантии, что **Line*** случайно не приведено к **Squiggle***. С классами, основанными на наследовании, компилятор не сможет помочь.

Поскольку документ должен хранить полиморфные объекты, нельзя использовать классы, сохраняющие простые объекты, — здесь необходимо применить **CTypedPtrArray**, **CTypedPtrList** или **CTypedPtrMap**. Вполне уверенно можно исключить вариант с картой, и вам остается лишь выбрать между массивом и списком. Каждый из них имеет как преимущества, так и недостатки. Вариант со списком может оказаться более эффективным, но за счет более сложного программирования. В интересах простоты давайте выберем для хранения документа класс **CTypedPtrArray**.

Реализация класса CMSDoc

Теперь, когда уже решено, какую структуру использовать для хранения данных, приступим к реализации класса **CMSDoc**. Для получения работающего класса потребуется выполнить следующие шаги:

- Заменить тип переменной класса **CMSDoc m_data** на **CTypedPtrArray**.
- Заменить функции **AddPoint()**, **NumPoints()** и **GetPoint()**, которые работают только с объектами **CPoint**, набором функций, оперирующих объектами **Shape**. Назовите функции **AddShape()**, **NumShapes()** и **GetShape()**. Назначение их в основном сохраняется.

- Ввиду того что документ теперь использует указатели и динамическую память, следует написать код для очистки памяти. Для этого перекройте виртуальную функцию **CDocument::DeleteContents()**, которая вызывается как раз перед разрушением документа и перед очисткой текущего документа.
- В заключение необходимо заняться сериализацией. Для этого не нужно вносить изменения в класс **CMSDoc**. Указав объекту **m_data** класса **CMSDoc** сериализовать себя, он заставляет выполнить сериализацию каждый содержащийся в нем элемент. Кроме того, потребуется обеспечить, чтобы все объекты **Shape** могли сохраняться и восстанавливаться с диска.

Добавление объекта CTypedPtrArray

Как часто случается, первый шаг — самый простой. Все, что необходимо сделать — это сменить текущий тип **CArray** переменной **m_data** класса **CMSDoc** на **CTypedPtrArray**. Как и **CArray**, шаблон **CTypedPtrArray** получает два параметра реализации.

В случае **CTypedPtrArray**, первый параметр называется *базовым типом коллекции*. Если необходимо хранить указатели на объекты, используйте **CObArray** (что, собственно, и делается). Для сохранения простых (**void**) указателей используйте **CPtrArray**.

Вторым параметром шаблона является тип указателя, который вы собираетесь хранить. В этом случае вы собираетесь хранить указатели на объекты **Shape**. Окончательное объявление класса **CMSDoc**, которое необходимо поместить в конец объявления класса **CMSDoc** вместо предыдущего объявления **m_data**, выглядит следующим образом:

```
CTypedPtrArray<CObArray, Shape*> m_data;
```

Не пытайтесь перекомпилировать проект после внесения данного изменения. Методы, работающие с **m_data**, уже не работают — ведь теперь **m_data** содержит указатели на объекты **Shape**.

Замена функций доступа

Следующий шаг преобразования кода связан с удалением функций доступа к **CPoint** и заменой их функциями, работающими с указателями на **Shape**.

Для удаления функции необходимо удалить как ее объявление (содержащееся в объявлении класса), так и реализацию. В этом вам поможет среда Visual C++, удаляющая объявление функции помещающая ее реализацию в комментарий.

Выполните следующие шаги:

1. При загруженном проекте MiniSketch выберите в окне Workspace панель ClassView и откройте класс **CMSDoc**. Найдите в списке функцию **AddPoint()** и щелкните на ней правой кнопкой мыши. Выберите из появившегося контекстного меню команду Delete (см. рис. 15.7).
2. Отобразившееся диалоговое окно предупреждения, объясняющее, что тело фун-

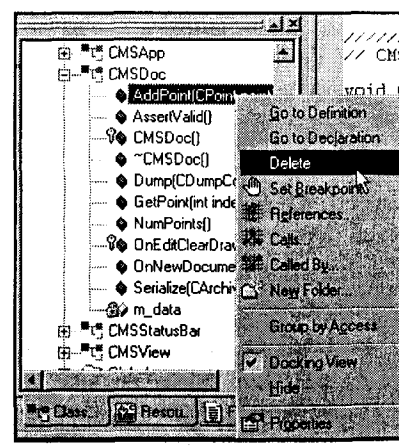


РИСУНОК 15.7. *Удаление метода.*

кции будет закомментировано, потребует вашего подтверждения (см. рис. 15.8). Щелкните на Yes.

3. Visual C++ удаляет из класса объявление функции, а в начале каждой строки функции ставит символы комментария, как показано на рис. 15.9.
4. Повторите аналогичные операции для функций **NumPoints()** и **GetPoint()**.

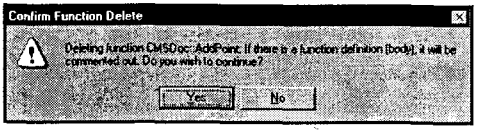


РИСУНОК 15.8. *Диалоговое окно подтверждения удаления функции.*

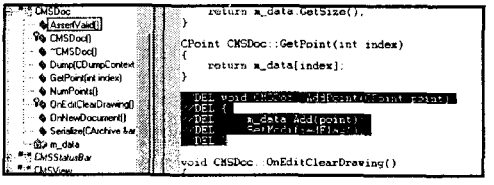


РИСУНОК 15.9. *Комментарии, помещенные в ваш исходный код.*

5. В панели ClassView щелкните правой кнопкой мыши на классе **CMSDoc** и выберите из контекстного меню команду Add Member Function. Добавьте три общедоступных функции, показанные в листинге 15.10.

Листинг 15.10. Функции CMSDoc для доступа к объектам Shape.

```
Shape * CMSDoc::GetShape(int item)
{
      return m_data[item];
}
int CMSDoc::NumShapes()
{
      return m_data.GetSize();
}
void CMSDoc::AddShape(Shape *pNewShape)
{
      m_data.Add(pNewShape);
}
```

Очистка памяти

Объекты **Shape** могут создаваться как классом документа, так и классом представления, однако очищать их все обязан объект документа. Для этого потребуется вызвать **delete** для каждого из элементов массива **m_data**. Для очистки памяти, используемой всеми указателями, вызовите функцию **RemoveAll()**.

Вот необходимые указания:

1. Выберите в окне ClassView класс **CMSDoc**. По щелчку правой кнопкой мыши откройте контекстное меню и выберите Add Virtual Function. В диалоговом окне New Virtual Function Override, показанном на рис. 15.10, выберите **DeleteContents()** и нажмите Add and Edit.
2. Введите код, показанный в листинге 15.11.

Листинг 15.11. Очистка памяти в классе документа.

```
void CMSDoc::DeleteContents()
{
      int nItems = m_data.GetSize();
```

```
        for (int i = 0; i < nItems; i++)
        {
              delete m_data[i];
        }
        m_data.RemoveAll();
}
```

Теперь, после очистки документа, можно просто использовать **SetSize(0,128)**, как это делалось при работе с классом **CArray**. Когда переменная **m_data** была массивом **CArray**, она хранила *объекты* **CPoint**, а не указатели, и, следовательно, могла автоматически управлять их памятью. Сейчас **m_data** имеет тип **CTypedPtrArray** и содержит *указатели* на **Shape**, а не объекты; следовательно, для каждого элемента коллекции потребуется вызывать **delete**. Выполнение **delete** для объекта приводит к вызову его деструктора, и объект получает возможность освободить выделенные ресурсы.

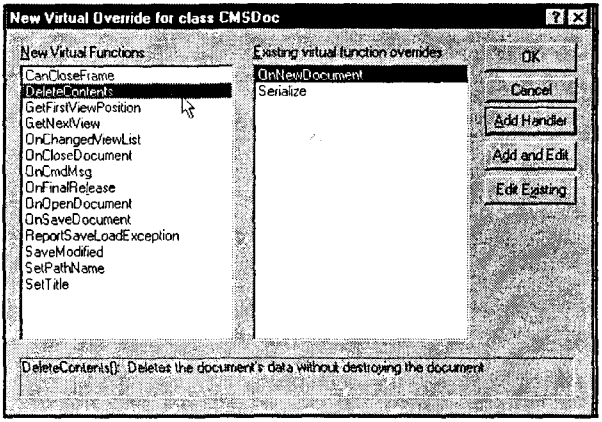


РИСУНОК 15.10. *Диалоговое окно New Virtual Function Override.*

Сериализация

В заключение необходимо завершить код сериализации для иерархии классов **Shape**. Прежде чем начать, давайте сконцентрируемся на верхнем уровне и разберемся, почему же сериализация столь важна и как она работает.

Предположим, что программа включает следующий код, где **pDoc** — это указатель на объект документа:

```
Shape *p = new Oval(RGB(255,0,0), 25,
                    RGB(0, 255, 0),
                    CPoint(0,0),
                    CPoint(100, 100));
pDoc->AddShape(p);
```

При выполнении этого кода в свободной памяти создается новый объект **Oval**, а конструкторы **CObject**, **Shape**, **FilledShape** и **Oval** инициализируют все элементы данных. Адрес нового объекта **Oval** сохраняется в переменной-указателе **p**, и по вызову функции **AddShape()** этот указатель добавляется к документу, как объяснялось ранее в главе. Затем **p** можно повторно использовать в каких-то других целях, поскольку документ позаботится об удалении объекта **Oval** из памяти. Такая структура памяти показана на рис. 15.11.

Предположим, что необходимо сохранить свой документ в дисковом файле. Что для этого должно произойти?

Вполне очевидно, что нельзя записать на диск значение **m_data**. **m_data** указывает на реальный массив указателей на **Shape**, находящихся где-то в куче. Точно

так же нет возможности просто записать на диски эти указатели на **Shape**. Указатели хранят всего-навсего позиции настоящих объектов в памяти. Вместо этого вам нужна информация, хранимая в каждом объекте: цвет пера, его толщина, цвет кисти и координаты ограничивающего прямоугольника. Тогда вы будете иметь *почти* все, необходимое для реконструкции объекта **Oval** при чтении с диска.

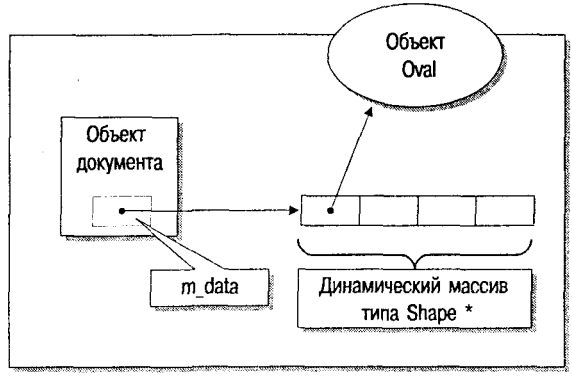


РИСУНОК 15.11. *Классы Shape в памяти.*

Отметьте, что мы сказали *почти*. Кроме значений элементов данных потребуется сохранить идентификационную информацию о типе времени выполнения (RTTI). При последующем чтении данных необходимо знать, что конструироваться должен новый объект **Oval**, а не новый объект **Box**. Когда механизм сериализации читает информацию RTTI, он находит определенный класс и создает новый объект, используя конструктор по умолчанию (без параметров). Остальная информация применяется для заполнения полей данных, чтобы они содержали в точности те же значения, что и до сериализации.

Детали сериализации в MFC

Класс MFC **CObject** имеет встроенную поддержку сериализации. Породив свои классы от **CObject**, можно автоматически записывать и получать объекты, соблюдая несколько простых правил:

- Класс должен быть потомком **CObject**.
- В объявлении класса должна вызываться макрокоманда **DECLARE_SERIAL()**, обеспечивающая упреждающее объявление функций динамического создания объектов.
- Файл реализации должен вызывать макрокоманду **IMPLEMENT_SERIAL()** для каждого сериализуемого класса. Эта макрокоманда добавляет функции, соответствующие упрежденно объявленным в макрокоманде **DECLARE_SERIAL()**.
- Для сохранения элементов данных на диске в классе должна быть перекрыта функция **Serialize()**.

Это все. О первых трех пунктах вы уже позаботились. Остается лишь написать функцию **Serialize()** для своих классов **Shape**.

Разработка Serialize()

Когда библиотека MFC вызывает функцию **Serilize()**, она передает ей ссылку на объект **CArchive**. Объект **CArchive** работает подобно стандартным потоковым объектам ввода-вывода **cout** и **cin**.

Для любого элементарного типа, такого как **int** или **float**, можно использовать перегруженный оператор включения (<<) для сохранения значений в архиве, или оператор исключения (>>) для получения значений из архива и сохранения их в переменных. Операторы включения и исключения также были перегружены для большинства простых объектов, не основанных на **CObject** (например, **CPoint** и **CRect**). Для некоторых элементарных типов Windows потребуется привести значение к общему примитиву. К примеру, значение **COLORREF** следует привести к типу **DWORD**.

Вместо отдельных объектов ввода и вывода, типа **cout** и **cin**, объект **CArchive** играет обе роли, но не одновременно. Перед использованием объекта включения необходимо вызвать его метод **IsStoring()**. Если **IsStoring()** возвращает **true**, архивный объект находится в режиме вывода и действует подобно **cout**. Если **IsStoring()** возвращает **false**, архивный объект находится в режиме ввода и действует подобно **cin**.

Давайте взглянем, как это работает, записав код **Shape::Serialize()**. Выполните приведенные ниже шаги:

1. В окне ClassView раскройте папку класса **Shape** и найдите функцию **Serialize()**. Дважды щелкните на ней для открытия редактора исходного кода.
2. В любом классе прежде, чем сериализовать вашу часть данных, следует предоставить возможность сериализовать данные базового класса. Следовательно, в первой строке **Shape::Serialize()** должен вызываться метод **CObject::Serialize()**. Передайте ему в качестве параметра объект **CArchive**.
3. Класс **Shape** имеет два элемента данных для сохранения: **int m_PenWidth** и **COLORREF m_PenColor**. Ни один из них не является потомком **CObject**, следовательно, необходимо использовать операторы включения и исключения в комбинации с функцией **IsStoring()**. Кроме того, потребуется привести **m_PenColor** от **COLORREF** к **DWORD**. Окончательный код функции **Shape::Serialize()** показан в листинге 15.12.

Листинг 15.12. Виртуальная функция Shape::Serialize().

```
void Shape::Serialize(CArchive &ar)
{
      CObject::Serialize(ar);
      if (ar.IsStoring())
      {
            ar << (DWORD)m_PenColor << m_PenWidth;
      }
      else
      {
            ar >> (DWORD)m_PenColor >> m_PenWidth;
      }
}
```

Если объект содержит в качестве полей потомки **CObject**, такие как поле **CArray m_data** в классе **Squiggle**, для их сериализации вы не должны использовать операторы включения и исключения. Вместо этого следует просто вызвать функцию **Serialize()** объекта, передав ей в качестве параметра полученный **CArchive**.

Функции **Serialize()** для остальных классов **Shape** могут быть найдены в листинге 15.13. Замените ими заглушки в Shape.cpp, и работу над классом **CMSDoc** можно считать завершенной.

Листинг 15.13. Виртуальные функции Serialize() для остальных классов иерархии Shape.

```
// Класс Line
void Line::Serialize(CArchive &ar)
{
      Shape::Serialize(ar);
      if (ar.IsStoring())
      {
            ar << m_LineStart << m_LineEnd;
      }
      else
      {
            ar >> m_LineStart >> m_LineEnd;
      }
}
// Класс Squiggle
void Squiggle::Serialize(CArchive &ar)
{
      Shape::Serialize(ar);
// Это CArray
      m_data.Serialize(ar);
}
// Класс FilledShape
void FilledShape::Serialize(CArchive &ar)
{
      Shape::Serialize(ar);

      if (ar.IsStoring())
      {
            ar << (DWORD) m_BrushColor;
      }
      else
      {
            ar >> (DWORD) m_BrushColor;
      }
}
// Класс Box
void Box::Serialize(CArchive &ar)
{
      FilledShape::Serialize(ar);
      if (ar.IsStoring())
      {
            ar << m_ULCorner << m_LRCorner;
      }
      else
      {
            ar >> m_ULCorner >> m_LRCorner;
      }
}
// Класс Oval
void Oval::Serialize(CArchive &ar)
{
      FilledShape::Serialize(ar);
      if (ar.IsStoring())
      {
            ar << m_ULCorner << m_LRCorner;
      }
      else
      {
```

```
            ar >> m_ULCorner >> m_LRCorner;
        }
    }
```

Что нового?

Вероятно, сейчас захочется откомпилировать **MiniSketch**, чтобы убедиться в отсутствии ошибок. К сожалению, две функции класса **CMSView** до сих пор ссылаются на старый документ, содержащий объекты **CPoint**. Закомментируйте код, использующий класс документа, в **CMSView::OnDraw()** и **CMSView::OnMouseMove()** и добавьте строку

```
#include "Shape.h"
```

в MiniSketchDoc.h. Теперь можно компилировать программу.

Она, естественно, пока не работает — за создание классов **Shape** для документа, который будет сохраняться, отвечает класс **CMSView**. На этом мы и сконцентрируемся в следующей главе. К концу главы 16 работа над приложением MiniSketch будет полностью завершена.

Глава 16

Совершенно новое представление: прокрутка и печать

Гуттенбергу не удалось существенно заработать на своем выдающемся изобретении. Вы же, изучив использование в MFC возможностей печати и прокрутки, получите значительную выгоду.

Иоганн Генсфляйх Цур-Ляден Цум-Гуттенберг (Johannes Gensfleisch Zur Laden Zum Gutenberg), известный истории просто как Гуттенберг, изобрел способ печати с помощью подвижной литеры. Метод Гуттенберга в XV веке оказался настолько передовым, что в процесс печати вплоть до XX века не вносилось никаких важных усовершенствований.

Подобно современным предпринимателям, Гуттенберг для финансирования разработки своего метода привлекал инвесторов. Один из инвесторов забеспокоился о перспективе финансовой выгоды и подал судебный иск против Гуттенберга. В 1455 году инвестор выиграл процесс, вынудивший Гуттенберга вернуть скромную деловую ссуду в 2200 гульденов. Позже инвестор получил контроль над печатью 42-строчной Библии, доходы от которой многократно превысили вложенные средства. Чтобы спасти Гуттенберга от нищеты, в 1465 году архиепископ Майнца присудил ему пенсию.

Не впадайте в панику — мы не собираемся вводить вас в финансовые крайности, прежде чем вы получите выгоду из наших уроков печати. В этой главе в приложение MiniSketch будут внесены окончательные усовершенствования. По завершении пользователи смогут применять вашу разработку для печати созданных ими шедевров. Они даже смогут создавать настенные фрески, слишком большие для представления в одном окне. Итак, приступим!

Что насчет цветов кисти?

Прежде чем сконцентрироваться на представлении документа, необходимо предоставить пользователю кнопку в панели инструментов, позволяющую изменять цвет кисти. Сделать это можно быстро и легко — считайте это кратким обзором главы 15.

Выполните приведенные ниже шаги:

1. Откройте проект MiniSketch и щелкните правой кнопкой мыши на классе **CMSView** в окне ClassView. Выберите из контекстного меню Add Member Variable. Создайте приватную переменную **m_BrushColor** с типом **COLORREF**. Диалоговое окно Add Member Variable показано на рис. 16.1.
2. С помощью этой же процедуры добавьте новый приватный объект **m_Brush** типа **CBrush**. В окне ClassView откройте в редакторе исходного кода конструктор **CMSView** и допишите в него следующие строки кода:

```
// Инициализируем цвет кисти
m_BrushColor = RGB(0, 0, 0);
m_Brush.CreateSolidBrush(m_BrushColor);
```

3. Переключитесь в секцию ResourceView окна Workspace, откройте папку Toolbar и дважды щелкните на ресурсе **IDR_MAINFRAME** для открытия Toolbar Editor. Перетащите пустую кнопку в панель инструментов в позицию за кнопкой цвета пера. На рис. 16.2 показано, как должен выглядеть ваш экран.
4. С помощью карандаша из набора средств рисования Toolbar Editor нарисуйте четыре заполненных разными цветами прямоугольника. Такой дизайн поможет отличать пиктограмму, используемую для изменения цвета кисти, от пиктограммы, предназначенной для изменения цвета пера. Экран должен иметь вид, представленный на рис. 16.3.

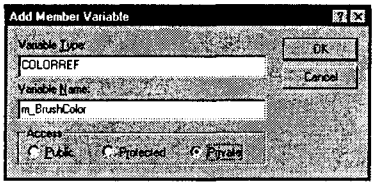


РИСУНОК 16.1. *Добавление переменной m_BrushColor в класс CMSView.*

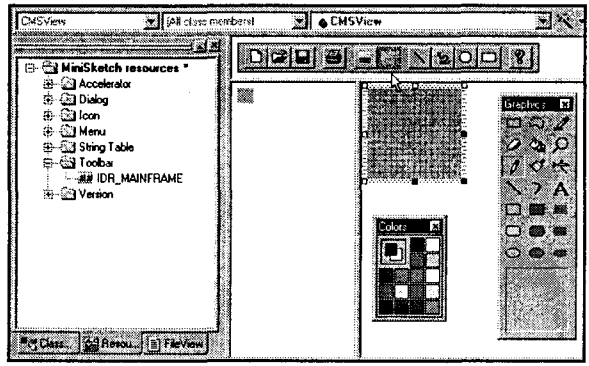


РИСУНОК 16.2. *Добавление в панель инструментов новой кнопки, управляющей выбором цвета кисти.*

5. Дважды щелкните в левой панели Toolbar Editor для загрузки диалогового окна Toolbar Button Properties. Введите новый идентификатор ресурса "**ID_BRUSH_COLOR**". Установите строку подсказки в "Установить цвет кисти для заполненных фигур\nЦвет кисти". Диалоговое окно должно выглядеть, как показано на рис. 16.4.
6. Из меню WizardBar выберите Add Windows Message Handler. В результате должно появиться диалоговое окно New Windows Message and Event Handlers for class CMSView. В списке Class or Object To Handle найдите и отметьте **ID_BRUSH_COLOR**. В списке New Windows Messages/Events выберите **COMMAND**. Как только окно приобретет вид, как на рис. 16.5, щелкните на Add And Edit.
7. Примите предлагаемое ClassWizard имя функции **OnBrushColor()**. Щелкните на OK, если окно выглядит, как показано на рис. 16.6.

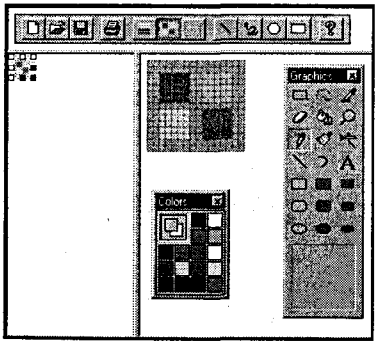


РИСУНОК 16.3. *Рисование кнопки, предназначенной для изменения цвета кисти.*

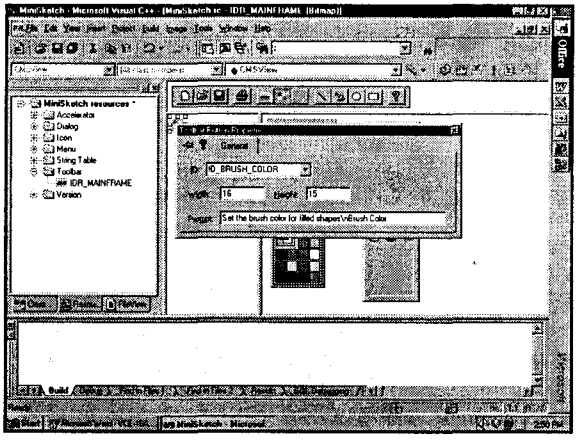


РИСУНОК 16.4. *Окончательный вид диалогового окна Toolbar Button Properties.*

8. К функции, сгенерированной ClassWizard, добавьте код из листинга 16.1.

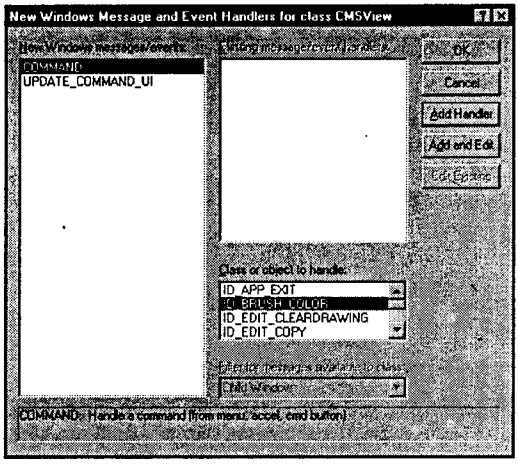


РИСУНОК 16.5. *Создание обработчика ID_BRUSH_COLOR.*

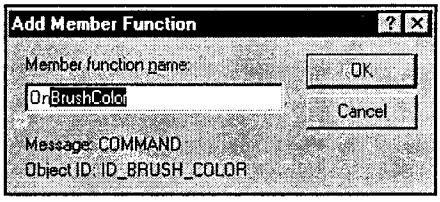


РИСУНОК 16.6. *Диалоговое окно Add Member Function.*

Листинг 16.1. Функция OnBrushColor().

```
void CMSView::OnBrushColor()
{
      // ЧТО СДЕЛАТЬ: Поместить здесь код обработки команды
      CColorDialog dlg(m_BrushColor);
      if (dlg.DoModal() == IDOK)
      {
            m_BrushColor = dlg.GetColor();
      }
}
```

После предоставления пользователю возможности изменения цвета кисти, можно было бы добавить в панель инструментов образец цвета кисти, аналогичный используемому для представления цвета пера. В интересах движения вперед, мы оставляем это в качестве упражнения и сконцентрируемся над тем, как получить работающее приложение MiniSketch.

Соединение документа и представления

Если вы помните общий план работ из предыдущей главы, то знаете, что сейчас необходимо изменить класс представления так, чтобы он создавал (при помощи **new**) объект **Shape**, когда пользователь совершит щелчок левой кнопкой мыши в клиентской области. При перемещении мыши за счет вызова его виртуальной функции **Update()** постоянно обновляется объект **Shape**. В заключение, когда пользователь отпустит кнопку мыши, окончательный объект **Shape** должен присоединиться к своим собратьям в классе **CMSDoc**.

Для этого потребуется переделать три функции **CMSView**: **OnLButtondown()**, **OnLButtonUp()** и **OnMouseMove()**. Начнем с функции **OnLButtonDown()**.

Создание новых фигур

В функции **OnLButtonDown()** следует создать новый объект **Shape** с типом, определяемым переменной **m_ShapeType**. Если, к примеру, **m_ShapeType** содержит значение **ID_SHAPE_TYPE_LINE**, создается новый объект **Line**; если же значение равно **ID_SHAPE_TYPE_RECTANGLE** — то новый объект **Box**. Стандартным типом фигуры будет **Squiggle**.

Поскольку новый объект **Shape** создается в куче с помощью оператора **new**, необходимо сохранить его адрес. Для этого предназначена новая переменная **m_pCurShape**, указывающая на рисуемую фигуру.

Создайте **OnLButtonDown()**, руководствуясь следующими шагами:

1. Откройте окно ClassView и найдите класс **CMSView**. Щелкните на нем правой кнопкой мыши и выберите из контекстного меню Add Member Variable. Добавьте переменную **private Shape*** с именем **m_pCurShape** и щелкните на OK, как только окно примет вид, показанный на рис. 16.7.

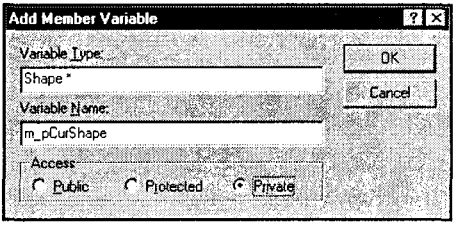


РИСУНОК 16.7. *Добавление переменной m_pCurShape.*

2. В окне ClassView откройте класс **CMSView** и найдите в списке функцию **OnLButtonDown()**. Откройте редактор исходного кода, дважды щелкнув на функции, и поместите в ее тело код, выделенный в листинге 16.2.

Листинг 16.2. Функция CMSView::OnLButtonDown().

```
void CMSView::OnLButtonDown(UINT nFlags, CPoint point)
{
      m_LineStart = point;
      SetCapture();
      switch (m_ShapeType)
      {
      case ID_SHAPE_TYPE_LINE:
            m_pCurShape = new Line(m_PenColor, m_PenWidth,
                                   point, point);
            break;
      case ID_SHAPE_TYPE_OVAL:
            m_pCurShape = new Oval(m_PenColor, m_PenWidth,
                                   m_BrushColor, point,
                                   point);
            break;
      case ID_SHAPE_TYPE_RECTANGLE:
            m_pCurShape = new Box(m_PenColor, m_PenWidth,
                                  m_BrushColor, point,
                                  point);
            break;
      case ID_SHAPE_TYPE_FREEHAND:
      default:
            m_pCurShape = new Squiggle(m_PenColor,
                                  m_PenWidth, point);
            break;
      }
}
```

Завершение фигуры

Ранее функция **OnLButtonUp()** просто освобождала захват мыши. В новой версии, когда пользователь отпускает кнопку мыши, потребуется выполнить еще три задачи:

- *Обновить окончательную фигуру, послав последнюю точку объекту* ***Shape***. Это достигается при помощи виртуальной функции **Update()** класса **Shape**.
- *Отправьте окончательный экземпляр* ***Shape*** *в объект документа для безопасного сохранения.* Для этого будет вызываться функция **AddShape()** класса **CMSDoc**.
- *Отобразить окончательный экземпляр Shape в текущем представлении.* Можно воспользоваться функцией **Invalidate()**, которая приведет к перерисовке всех объектов **Shape**, хранимых в документе. Впрочем, на самом деле в этом нет необходимости, поскольку непосредственное рисование оказывается намного быстрее. Теперь, после сохранения нового **Shape** в документе, он после обновления экрана пропадать не будет.

Для добавления упомянутых функциональных возможностей отыщите функцию **OnLButtonUp()**, как искали **OnLButtonDown()**. Добавьте в нее выделенный код из листинга 16.3.

Листинг 16.3. Функция CMSView::OnLButtonUp().

```
void CMSView::OnLButtonUp(UINT nFlags, CPoint point)
{
    CClientDC dc(this);
    m_pCurShape->Update(point);
    m_pCurShape->Draw(&dc);
    ReleaseCapture();

    CMSDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    pDoc->AddShape(m_pCurShape);
}
```

Резиновая нить

Последним членом триумвирата рисования вашего класса представления является функция **OnMouseMove()**. Как и в приложении PaintORama, текущая фигура рисуется по мере перемещения мыши, используя растягивание фигуры, или метод резиновой нити. При каждом изменении позиции мыши новый прямоугольник или овал отображается, а предыдущий — удаляется. Как и ранее, так следует поступать во всех случаях, *кроме* рисования от руки.

Вот что необходимо сделать:

- Если мышь перемещается, получите контекст устройства клиентской области, используя **CClientDC**.
- Если вы не рисуете от руки, смените режим рисования на **R2_NOT** и вызовите функцию **Draw()** текущего объекта **Shape**. Вспомните, что режим **R2_NOT** просто инвертирует текущий фон, игнорируя цвет пера. Поскольку координаты **Shape** еще не обновлялись, перо перерисует фигуру в режиме **R2_NOT**, тем самым стирая ее.

- Обновите координаты текущего объекта **Shape**, обратившись к функции **Update()** и передав ей в качестве параметра координаты мыши.
- Вызовите функцию **Draw()** текущего объекта **Shape()**, передавая в параметре полученный ранее контекст устройства клиента. Помните, если вы не рисуете от руки, это рисование должно выполняться в режиме **R2_NOT**.

Для внесения перечисленных изменений приведите функцию **CMSView::OnMouseMove()** в соответствие с листингом 16.4. В ней потребуется удалить несколько строк и добавить выделенные в листинге.

Листинг 16.4. Функция CMSView::OnMouseMove().

```
void CMSView::OnMouseMove(UINT nFlags, CPoint point)
{
      if (nFlags & MK_LBUTTON)
      {
            CClientDC dc(this);
            if (m_ShapeType != ID_SHAPE_TYPE_FREEHAND)
            {
                  dc.SetROP2(R2_NOT);
                  m_pCurShape->Draw(&dc);
            }
            m_pCurShape->Update(point);
            m_pCurShape->Draw(&dc);
      }
}
```

Рисование фигуры

Несмотря на то что работа с пользовательским интерфейсом приложения MiniSketch завершена, необходимо выполнить еще две задачи, прежде чем оно заработает.

- Научить классы **Line**, **Box** и **Oval** самостоятельно отображаться, реализовав их функции **Draw()**.
- Создать код для перерисовки всего изображения при вызове функции **CMSView::OnDraw()**.

Сперва взглянем на рисование каждого из классов **Shape**.

Самостоятельно визуализирующиеся фигуры

Код для визуализации каждого из классов **Shape** довольно очевиден. Каждый класс содержит виртуальную функцию **Draw()**, которая вызывается из класса представления через указатель **m_pCurShape**. В связи с тем что **m_pCurShape** является указателем на **Shape** (а не на **Line** или **Box**), функцию **Draw()** следует объявить виртуальной — иначе класс представления будет вызывать функцию **Draw()** класса **Shape()**, даже если **m_pCurShape** на самом деле указывает на объекты **Line** или **Box**.

При вызове **Draw()** класс представления передает объекту **Shape** указатель на DC, используемый при визуализации. Сперва каждый объект **Shape** должен создать свое собственное перо и выбрать его в DC. По завершении **Shape** должен восстановить в DC прежнее перо:

```
CPen pen, *oldPen;
pen.Create(PS_SOLID, m_PenWidth, m_PenColor);
oldPen = pDC->SelectObject(&pen);
// Далее следует код рисования
pDC->SelectObject(oldPen);
```

Переменные **m_PenWidth** и **m_PenColor** являются защищенными элементами данных, унаследованными от класса **Shape**. Если вы добавили код обработки стилизованных перьев, вместо строго определенного значения стиля **PS_SOLID** воспользуйтесь переменной **m_PenStyle**. Функция **SelectObject()** возвращает указатель на старое перо — последняя строка восстанавливает старое перо, удаляя в процессе новое перо из DC.

Классы, порожденные от **FilledShape** (в нашем примере — **Oval** и **Box**), должны подобным образом создавать кисть и выбирать ее в контексте устройства. По завершению рисования они должны восстановить предыдущую кисть.

Для изменения функций **Draw()** можно воспользоваться окном ClassView. Введите код, показанный в листинге 16.5. Заметьте, что в нем нет кода для **Shape::Draw()** и **FilledShape::Draw()** — ни один из этих классов не знает, как перерисовать *конкретную* фигуру. Тело каждой из функций должно состоять из пары фигурных скобок.

Листинг 16.5. Методы Draw() в иерархии классов Shape().

```
// Класс Line
void Line::Draw(CDC * pDC)
{
      CPen pen, *pOldPen;
      pen.CreatePen(PS_SOLID, m_PenWidth, m_PenColor);
      pOldPen = pDC->SelectObject(&pen);
      pDC->MoveTo(m_LineStart);
      pDC->LineTo(m_LineEnd);
      pDC->SelectObject(pOldPen);
}

// Класс Squiggle
void Squiggle::Draw(CDC * pDC)
{
      CPen pen, *pOldPen;
      pen.CreatePen(PS_SOLID, m_PenWidth, m_PenColor);
      pOldPen = pDC->SelectObject(&pen);
      int nPoints = m_data.GetSize();
      pDC->MoveTo(m_data[0]);
      for (int i = 1; i < nPoints; i++)
      {
            pDC->LineTo(m_data[i]);
      }
      pDC->SelectObject(pOldPen);
}

// Класс Box
void Box::Draw(CDC * pDC)
{
      CPen pen, *pOldPen;
      pen.CreatePen(PS_SOLID, m_PenWidth, m_PenColor);
      pOldPen = pDC->SelectObject(&pen);

      CBrush brush, *pOldBrush;
      brush.CreateSolidBrush(m_BrushColor);
```

```
        pOldBrush = pDC->SelectObject(&brush);
        pDC->Rectangle(CRect(m_ULCorner, m_LRCorner));
        pDC->SelectObject(pOldPen);
        pDC->SelectObject(pOldBrush);
}
// Класс Oval
void Oval::Draw(CDC * pDC)
{
        CPen pen, *pOldPen;
        pen.CreatePen(PS_SOLID, m_PenWidth, m_PenColor);
        pOldPen = pDC->SelectObject(&pen);
        CBrush brush, *pOldBrush;
        brush.CreateSolidBrush(m_BrushColor);
        pOldBrush = pDC->SelectObject(&brush);
        pDC->Ellipse(CRect(m_ULCorner, m_LRCorner));
        pDC->SelectObject(pOldPen);
        pDC->SelectObject(pOldBrush);
}
```

Реализация Update()

Помимо возможности самовизуализации, каждый объект **Shape** имеет виртуальную функцию **Update()**, вызываемую функцией **OnMouseMove()** класса представления. Поведение функций различается в зависимости от типа объекта. Объектам **Squiggle**, к примеру, требуется добавить новую точку во внутренний массив **CPoint**, объектам **Line** — обновить свою переменную **m_LineEnd**, а объектам **Box** и **Oval** — изменить свою переменную **m_LRCorner**.

Для добавления необходимого кода можно открыть Shapes.cpp из окна FileView либо посетить каждую из функций **Update()** через ClassView. Коды этих функций для классов **Line**, **Squiggle**, **Oval** и **Box** показаны в листинге 16.6. (Для классов **Shape** и **FilledShape** код вообще не требуется.)

Листинг 16.6. Функции Update() для иерархии классов Shape.

```
void Line::Update(CPoint newPoint)
   { m_LineEnd = newPoint;   }
void Squiggle::Update(CPoint newPoint){ m_data.Add(newPoint);   }
void Box::Update(CPoint newPoint)
   { m_LRCorner = newPoint; }
void Oval::Update(CPoint newPoint)
   { m_LRCorner = newPoint; }
```

Переделка OnDraw()

Если в этот момент откомпилировать и запустить программу (закомментировав код функции **OnDraw()**, который ссылается на **NumPoints()** и **GetPoint()**), все режимы рисования будут работать. Можно даже сохранить свою работу на диске, а затем прочитать ее. Правда, после сохранения работы вы уже не сможете ее увидеть. Несмотря на то что некоторые могут счесть это новой возможностью (Новинка! Первая программа, защищающая ваши художества от любопытных

глаз!), эту ошибку следует исправить. Последнее подразумевает написание новой функции **OnDraw()** в классе **CMSView**.

Новая функция **OnDraw()** проще предыдущей версии, работающей только с **CPoint**. Она ничего не рисует, а лишь получает от документа все объекты **Shape** и предлагает им перерисоваться. Что может быть проще?

Для добавления в MiniSketch новой функции **OnDraw()** найдите и откройте в окне ClassView класс **CMSView**. Отыщите в списке функцию **OnDraw()** и дважды щелкните на ней для открытия редактора исходного кода. Поместите в нее код, выделенный в листинге 16.7.

Листинг 16.7. Функция CMSView::OnDraw().

```
////////////////////////////////////////////////////////////////////////////
   /////
// Рисование в CMSView

void CMSView::OnDraw(CDC* pDC)
{
      CMSDoc* pDoc = GetDocument();
      ASSERT_VALID(pDoc);

      int nShapes = pDoc->NumShapes();
      for (int index = 0; index < nShapes; index++)
      {
            m_pCurShape = pDoc->GetShape(index);
            m_pCurShape->Draw(pDC);
      }
}
```

По завершении ввода **OnDraw()** можно откомпилировать и запустить программу. Попробуйте использовать различные перья и кисти. Выбирайте различные цвета. Сохраните свои художества на диске и загрузите их. Воспользуйтесь командами печати и предварительного просмотра. Как видите, приложение MiniSketch — завершенное и полнофункциональное. Причем окончательно!

Ну, более или менее.

Например, при открытии нового документа или при чтении документа из файла экран обновляется не сразу. Это можно исправить, вызвав **UpdateAllViews(NULL)** в конце функций **CMSDoc::OnNewDocument()** и **CMSDoc::Serialize()**. С печатью и предварительным просмотром связаны более серьезные проблемы. Они, конечно, работают, но взгляните на рис. 16.8, где показано одно и то же изображение, нарисованное в рабочем окне и в окне предварительного просмотра.

Очевидно, технология "Что вижу, то и получаю" (What-You-See-Is-What-You-Get, или WYSIWYG), применима к MiniSketch только в том случае, если вы считаете нормальным, что размер напечатанного изображения оказывается меньше размера изображения на экране.

К счастью, в Windows это делается просто, если вообще не элементарно. Приготовьтесь к исследованию очаровательного мира режимов отображений Windows.

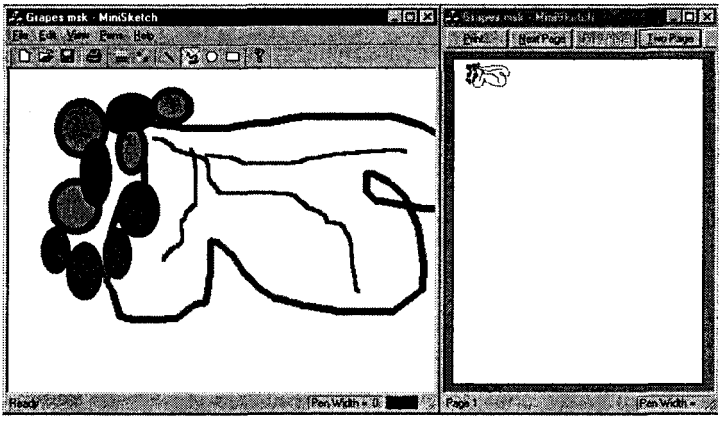


РИСУНОК 16.8. *Рисунок, выведенный MiniSketch на стандартном экране и в режиме предварительного просмотра.*

Альтернативные представления: режимы отображения

Одним из преимуществ Windows является ее независимость от устройств. Программисты не должны беспокоиться о конкретном разрешении дисплея пользователя — 640 × 480 или, скажем, 1600 × 1200. Программист не должен беспокоиться о том, куда направляется вывод: на струйный принтер с разрешением 300 dpi или на камеру с разрешением 4800 dpi. Программисты *вообще* не должны беспокоиться по поводу устройств.

Как было только что показано, независимость от устройств не является автоматической или встроенной. Очевидно, если вы используете команды

```
dc.MoveTo(100, 100);
dc.LineTo(400, 400);
```

то получите разительно отличающиеся результаты в случае записи на камеру (где линия будет иметь длину меньше одной десятой дюйма) и на струйный принтер (где она займет около 1.5 дюймов). Как GDI может позволить программе работать в обеих средах без какого-то специального кода для каждого устройства? Ответ кроется в различии между *координатами устройств* и *логическими координатами*.

Координаты устройства связаны с размерами пиксела или точки, используемой конкретным устройством. Когда вы говорите об экране с разрешением 640 × 480 dpi, вы имеете в виду координаты устройства. Большинство функций Windows (например, функции работы с мышью) работают с координатами устройства.

Функции вывода класса **CDC** являются исключением из этого правила. Функции контекстов устройств работают с логическими координатами; так, если записать **dc.MoveTo(100,100)**, то не обязательно произойдет перемещение на 100 пикселей вниз и вправо от верхнего левого угла экрана или принтера. Вместо этого выполнится перемещение на 100 логических единиц в положительном направлении оси *x* и на 100 логических единиц в положительном направлении оси *y*. Каков же размер логической единицы? Он изменяется и определяется текущим режимом отображения. Итак, прежде чем разобраться с логическими единицами, следует понять, что собой представляют режимы отображения.

Режим отображения MM_TEXT

Windows имеет 8 режимов отображения, перечисленных в табл. 16.1. Простейший из них, **MM_TEXT**, выступает как режим по умолчанию. Если вы ничего не делали, то режимом отображения DC будет **MM_TEXT**.

Режим отображения **MM_TEXT** имеет две определяющие характеристики. Во-первых, единица измерения эквивалентна единице устройства; следовательно, перемещение в режиме **MM_TEXT** на 100 логических единиц означает также перемещение на 100 единиц устройства, или пикселей. Вот почему вывод на печать в MiniSketch оказывается настолько меньше вывода на экран — принтер имеет больше точек на дюйм, нежели экран.

Таблица 16.1. Логические режимы отображения GDI.

Режим отображения	*Единица измерения*	*Направление оси ординат*
MM_TEXT	Пиксел	Положительное
MM_LOMETRIC	0.1 мм	Отрицательное
MM_HIMETRIC	0.01 мм	Отрицательное
MM_LOENGLISH	0.01 дюйма	Отрицательное
MM_HIENGLISH	0.001 дюйма	Отрицательное
MM_TWIPS	1/1440 дюйма	Отрицательное
MM_ISOTROPIC	Переменная	Переменное
MM_ANISOTROPIC	Переменная	Переменное

Во-вторых, в режиме **MM_TEXT** координаты интерпретируются так, как если бы читался обычный печатный текст на странице. Если представить себе страницу, состоящую из колонок и строк (как в электронной таблице), тогда строка с минимальным номером находится сверху, а колонка с минимальным номером — слева. В режиме **MM_TEXT** при увеличении значения *x* координата движется вправо, а при увеличении значения *y* — вниз экрана.

Помимо простоты использования, режим отображения **MM_TEXT** не отличается хорошими характеристиками, поэтому применять его не рекомендуется.

Режимы отображения, не зависящие от устройства

Режимы отображения **MM_ISOTROPIC** и **MM_ANISOTROPIC** позволяют целиком и полностью задать соотношение между логическими и физическими координатами, а также направление каждой из координатных осей. Режим **MM_ISOTROPIC** требует использования одинаковых характеристик отображения для обеих координатных осей. Режим **MM_ANISOTROPIC** позволяет задавать отдельные характеристики каждой оси.

Впрочем, в большинстве приложений будет применяться один из режимов отображения с постоянным масштабом. Каждый режим отображения с постоянным масштабом имеет предопределенный фактор масштабирования. Если вы, к примеру, нарисуете линию длиной в 100 пикселей в режиме **MM_LOENGLISH**, линия будет иметь длину около дюйма как на экране, так и на принтере, независимо от того, какое у принтера разрешение — 300 точек на дюйм или 3000. Остальные режимы отображения с постоянным масштабом работают сходным образом. Линия

длиной в 100 единиц в режиме **MM_HIENGLISH** будет иметь длину 0.1 дюйма, в режиме **MM_LOMETRIC** — 10 мм, в **MM_HIMETRIC** — 1 мм, а в **MM_TWIPS** — около 0.07 дюйма.

По умолчанию во всех режимах отображения за координату (0, 0) принимается верхний левый угол экрана или печатаемой страницы. При желании это можно изменить с помощью функций DC **SetViewPortOrg()** и **SetWindowOrg()**. **SetWindowOrg()** устанавливает новую логическую точку в качестве верхнего левого угла экрана, тогда как **SetViewPortOrg()** отображает логическую точку (0, 0) на координаты устройства, передаваемые в параметрах.

Здесь легко запутаться, поэтому сейчас будет приведен простой пример. Предположим, необходимо, чтобы верхним левым углом экрана стала логическая точка (100, 100), а не (0, 0). Для этого вызовите **SetWindowOrg(100, 100)**. С другой стороны, что делать, если требуется, чтобы логическая точка (0, 0) находилась на расстоянии в 100 единиц устройства вниз и влево? Просто вызовите **SetViewPortOrg(100, 100)**.

Все режимы отображения с постоянным масштабом для описания координатных осей *x* и *y* применяют *декартовы координаты*. Вспомните, вы с ними наверняка работали, изучая геометрию. В декартовой системе координат ось абсцисс пересекается с осью ординат в точке (0, 0). Ось абсцисс направлена вправо, ось ординат — вверх.

Это, конечно, отличается от режима **MM_TEXT**, где ось ординат направлена вниз. В режиме **MM_TEXT** вызов **dc.Rectangle(0, 0, 100, 100)** приведет к рисованию прямоугольника в верхнем левом углу экрана. Тот же вызов в одном из режимов отображения с постоянным масштабом приведет к невидимому изображению — прямоугольник будет располагаться над точкой (0, 0), а не под ней. Для рисования такого же прямоугольника в одном из режимов отображения с постоянным масштабом потребуется написать **dc.Rectangle(0, 0, 100, -100)**. Обратите внимание на отрицательное значение последнего параметра.

Отображение приходит в MiniSketch

Включение в программу MiniSketch режимов отображения не повлечет за собой написания массы нового кода, хотя будут некоторые детали, которые приведут неосторожных к допущению ошибок. Крупнейшие из этих деталей относятся к необходимости выполнения преобразований между координатами устройства и логическими координатами. Преобразование требуется в связи с тем, что за исключением функций вывода **CDC** и некоторых отдельных функций GDI, большинство координат Windows выражаются координатами устройств.

Эту проблему можно преодолеть — Windows предоставляет функцию **DPtoLP()** для преобразования точек устройства в логические и функцию **LPtoDP()** для обратного преобразования. Остается один вопрос — где выполнять преобразование? Сохранить координаты устройства в классе документа, а затем преобразовать их в логические координаты при выводе? Или же сохранить в документе логические координаты и преобразовывать щелчки и перемещения мыши в логические координаты? Можно воспользоваться любым из перечисленных способов, но давайте пока отложим это.

В MiniSketch будет использоваться режим **MM_LOENGLISH**, в котором каждая точка имеет размер 0.01 дюйма как на экране, так и на принтере. Выполните следующие шаги:

1. Откройте проект MiniSketch и отыщите в окне ClassView класс **CMSView**. Щелкните на нем правой кнопкой мыши и выберите из контекстного меню Add Virtual Function. В диалоговом окне New Virtual Override найдите функцию **OnPrepareDC()**. Как только экран примет вид, показанный на рис. 16.9, щелкните на Add And Edit.

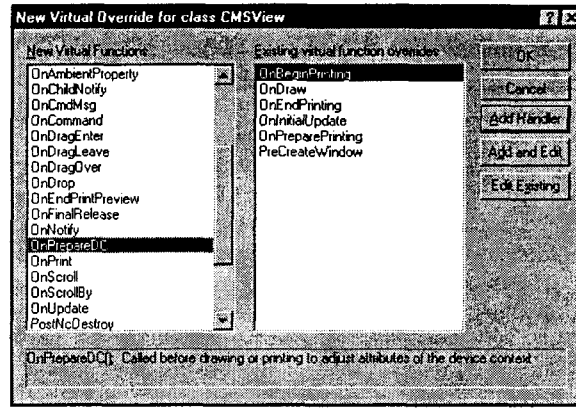


РИСУНОК 16.9. *Добавление функции OnPrepareDC().*

2. Выделенные строки кода в листинге 16.8 добавьте в функцию **OnPrepareDC()**, сгенерированную ClassWizard. Функция **SetMapMode()** устанавливает режим отображения **MM_LOENGLISH**. MFC автоматически вызывает функцию **OnPrepareDC()** перед вызовом **OnDraw()**.

Листинг 16.8. Функция CMSView::OnPrepareDC().

```
void CMSView::OnPrepareDC(CDC* pDC, CPrintInfo* pInfo)
{
      // ЧТО СДЕЛАТЬ: Поместить здесь специализированный код
      // и/или вызвать метод базового класса
      pDC->SetMapMode(MM_LOENGLISH);
      CView::OnPrepareDC(pDC, pInfo);
}
```

3. Автоматический вызов **OnPrepareDC()** перед вызовом **OnDraw()** не означает, что MFC автоматически будет вызывать эту функцию перед другими функциями, рисующими в контексте устройства, такими как **OnMouseMove()**. В таких случаях потребуется модифицировать функции обработки событий мыши. Начните с добавления в начало функции **OnLButtonDown()** трех строк, показанных в листинге 16.9. Вызов **OnPrepareDC()** после создания контекста устройства гарантирует работу в режиме отображения **MM_LOENGLISH**, а вызов **DPtoLP()** гарантирует, что координаты мыши, хранимые в **point**, перед использованием корректно преобразуются в логические единицы.

Листинг 16.9. Изменения в функции CMSView::OnLButtonDown.

```
CClientDC dc(this);
OnPrepareDC(&dc);
dc.DPtoLP(&point);
```

Такие же строки необходимо добавить в функции **OnMouseMove()** и **OnLButtonUp()**. В каждую из функций добавьте две выделенных строки из листинга 16.10 непосредственно после строки, в которой создается контекст устройства.

Листинг 16.10. Изменения в функциях OnMouseMove() и OnLButtonUp().

```
CClientDC dc(this);
OnPrepareDC(&dc);
dc.DPtoLP(&point);
```

После внесения этих изменений откомпилируйте и запустите программу. Теперь изображение на экране имеет примерно те же размеры, что и на принтере. (На самом деле размер изображения на экране немного больше его размера на принтере. Линия длиной в 100 единиц в режиме **MM_LOENGLISH** будет иметь длину на принтере ровно в 1 дюйм, но на экране она будет длиной около 1.5 дюймов. Последнее связано с тем, что дисплеи используют логический дюйм, оптимизированный с целью улучшения четкости стандартного 10-точечного текста до четкости 12-точечного.)

Прокрутка окон представления

Все же MiniSketch еще имеет один небольшой недостаток. В случае отсутствия дисплея с высоким разрешением нет возможности создать изображение, которое смогло бы заполнить страницу форматом 8.5x11 дюймов. Обычно эта проблема решается за счет механизма прокрутки, позволяющего видеть в каждый конкретный момент времени только часть изображения.

Добавление прокрутки в MiniSketch даже проще изменения режима отображения с **MM_TEXT** на **MM_LOENGLISH**. Если бы прокрутка планировалась заранее, можно было бы указать AppWizard включить прокрутку автоматически, использовав в качестве базового класса для **CMSView** класс **CScrollView**, а не **CView**.

К счастью, даже при таком упущении обеспечение прокрутки в приложении не составляет особого труда. Просто выполните следующие шаги:

1. Отыщите в окне ClassView класс **CMSView** и дважды щелкните на нем, чтобы открыть объявление класса. В заголовке класса измените **CView** на **CScrollView**:

   ```
   class CMSView : public CScrollView
   ```

2. Откройте файл реализации **CMSView** — MiniSketch.cpp, и найдите в начале файла две следующих строки:

   ```
   IMPLEMENT_DYNCREATE(CMSView, CView)
   BEGIN_MESSAGE_MAP(CMSView, CView)
   ```

   Замените **CView** на **CScrollView**, как показано ниже:

   ```
   IMPLEMENT_DYNCREATE(CMSView, CScrollView)
   BEGIN_MESSAGE_MAP(CMSView, CScrollView)
   ```

3. Измените функцию OnPrepareDC(), закомментировав обращение к **SetMapMode()** и заменив **CView::OnPrepareDC()** на **CScrollView::OnPrepareDC()**. Окончательно функция должна выглядеть, как показано в листинге 16.11.

Листинг 16.11. Измененная функция OnPrepareDC().

```
void CMSView::OnPrepareDC(CDC* pDC, CPrintInfo* pInfo)
{
    // ЧТО СДЕЛАТЬ: Поместить здесь специализированный код
    // и/или вызвать метод базового класса
```

```
    pDC->SetMapMode(MM_LOENGLISH);
    CScrollView::OnPrepareDC(pDC, pInfo);
}
```

4. В программе, использующей **CScrollView**, режим отображения в функции **OnPrepareDC()** не устанавливается. Вместо этого он устанавливается в функции **OnInitialUpdate()** при помощи вызова **SetScrollSizes()**. **SetScrollSizes()** принимает несколько параметров, но передать следует только первых два. В первом параметре задается режим отображения — вы используете **MM_LOENGLISH**, как и ранее. Второй параметр определяет размеры документа в логических единицах. Установите создание страниц формата 8.5 × 11 дюймов, т.е. размер 800 × 1050. В **MM_LOENGLISH**, где 1 единица равна 0.01 дюйма, такие размеры будут соответствовать ширине в 8 дюймов и высоте в 10.5 дюймов, допуская 0.25-дюймовый отступ с обеих сторон и 0.5-дюймовый отступ сверху и снизу. Окончательная функция **OnInitialUpdate()** приведена в листинге 16.12, причем новые строки в ней выделены.

Листинг 16.12. Функция OnInitialUpdate() для установки прокрутки.

```
void CMSView::OnInitialUpdate()
{
    InitPen();
    SetScrollSizes(MM_LOENGLISH, CSize(800, 1050));
}
```

Запустив программу после компиляции, вы сможете рисовать на "виртуальном" холсте форматом 8.5 × 11 дюймов, получая доступ к различным его областям с использованием полос прокрутки. Рисунок 16.10 демонстрирует, что изображения в режиме предварительного просмотра действительно занимают весь экран.

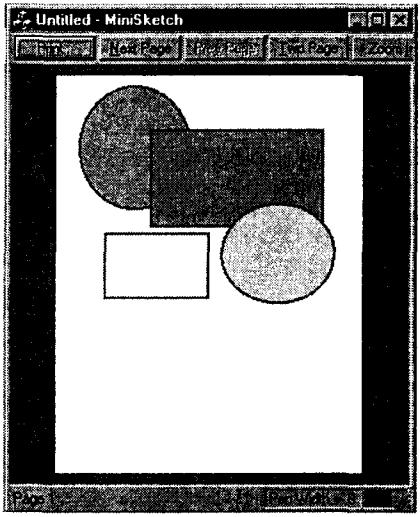


РИСУНОК 16.10. *Окно предварительного просмотра при использовании прокрутки.*

Если приложению требуется одностраничный вывод, как это имеет место в случае MiniSketch, то комбинация класса **CScrollView**, режимов отображения, не зависящих от устройств, и функции **OnDraw()** сможет удовлетворить все потребности в печати. Впрочем, для реализации более сложных требований к печати, необходимо узнать о том, как печать обрабатывается в MFC. Это рассматривается в следующем разделе.

Печать и предварительный просмотр

Будем предельно честными: печать в Windows просто отвратительна. Это не только предмет получения контекста устройства принтера и последующих вызовов GDI. Потребуется позаботиться о нумерации страниц, спулинге, написании про-

цедур обратного вызова аварийного завершения, разрешении и запрете главного окна и еще о массе деталей, которые не перечислить и за день. В MFC этот процесс реализован неплохо, но это еще не означает, что его можно назвать привлекательным.

Первым, что необходимо понять о печати в Windows независимо от того, печатаете вы в API или в MFC — то, что печать ориентирована на страницы. Постраничная печать начинается при вызове **StartPage()** и заканчивается при вызове **EndPage()**. (MFC делает это за вас.) Между вызовами для составления страниц используются те же функции GDI, что и для вывода на экран. Так каким же образом Windows знает, когда прекращать печать текущей страницы и переходить к следующей? Ответ прост: никаким. Составление каждой страницы, сообщение принтеру о необходимости перехода на следующую страницу и упорядочивание текста или графики для печати в нужной позиции на нужной странице полностью находится в вашей компетенции. Как и было сказано — это не очень хорошая картина.

В MFC вы не должны беспокоиться о низкоуровневых функциях печати, наподобие **StartPage()** или **EndPage()**. Также не потребуется получать контекст устройства для принтера, обрабатывать очередь печати или прерывание печати. Главным образом все, что необходимо сделать — это переопределить одну или несколько виртуальных функций, показанных на рис. 16.11. Давайте их рассмотрим.

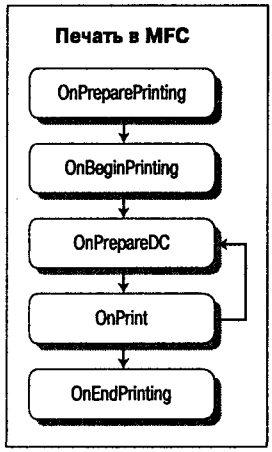


РИСУНОК 16.11. *Функции печати в MFC.*

Функции печати MFC

При использовании AppWizard для добавления поддержки печати и предварительного просмотра, им осуществляется перекрытие трех из пяти функций печати MFC. Ниже рассматривается назначение этих 5 функций:

- ***OnPreparePrinting()***. Выводит диалоговое окно печати и дает возможность разбить документ на страницы.
- ***OnBeginPrinting()***. Вызываемая прямо перед началом печати, эта функция дает возможность создать шрифты, перья и кисти, которые будут использоваться во всем документе с тем, чтобы не создавать их каждый раз для каждой страницы.
- ***OnPrepareDC()***. Вызываемая прямо перед началом печати, эта функция изменяет начало печати, чтобы можно было печатать выбранные страницы по нескольку раз.
- ***OnPrint()***. Выполняет нормальную печать. Стандартная реализация функции содержит простой вызов **OnDraw()**. Ее можно перекрыть, оставив вызов **OnDraw()** и добавив собственные заголовки, отступы и номера страниц.
- ***OnEndPrinting()***. Вызываемая после того, как напечатана последняя страница, эта функция позволяет освободить кисти, перья и другие ресурсы, которые выделялись в **OnBeginPrinting()**.

Функция OnPreparePrinting()

AppWizard автоматически перекрывает функцию **OnPreparePrinting()**, которая обращается к функции **DoPreparePrinting()**. Вот стандартная реализация **OnPreparePrinting()**:

```
BOOL CMSView::OnPreparePrinting(CPrintInfo *pInfo)
{
     return DoPreparePrinting(pInfo);
}
```

Функция **DoPreparePrinting()** выводит диалоговое окно печати (Print) и возвращает контекст устройства используемого принтера. Закрытие диалогового окна Print либо возврат функцией **OnPreparePrinting()** значения **FALSE** прерывает печать.

Когда каркас вызывает **OnPreparePrinting()** (и когда вы вызываете **DoPreparePrinting()**), в качестве параметра в нее передается объект **CPrintInfo**. На протяжении всего цикла печати используется один и тот же объект **CPrintInfo**. Им же можно воспользоваться для получения состояния печати, например, определить страницу, печатаемую в текущий момент.

Главное предназначение **OnPreparePrinting()** — это предоставление средств для сообщения процессу печати о том, сколько страниц должно быть напечатано. Сделать это можно при помощи функций **SetMinPage()** и **SetMaxPage()** класса **CPrintInfo**. После установки минимального и максимального номеров страниц механизм предварительного просмотра будет использовать для определения количества выводимых страниц числа, введенные в диалоговом окне Print.

Как же сообщить, сколько страниц содержит ваш документ? Это весьма неприятная часть работы. Не существует быстрого или автоматического способа. Если у вас, например, текстовый файл, потребуется вычислить, сколько строк помещается на одной странице (при определенном шрифте) и сколько строк находится во всем файле. Затем только можно вычислить количество страниц. Нелегкая задача. Если вы не знаете этих данных (как часто случается), воспользуйтесь флагом в объекте **CPrintInfo**, чтобы сообщить процессу печати, переходить ли к следующей странице.

Функция OnBeginPrinting()

В случае удачного завершения функция **DoPreparePrinting()** создаст контекст устройства принтера и указатель на него, вместе с указателем на описанный ранее объект **CPrintInfo**, который передается в **OnBeginPrinting()**.

OnBeginPrinting() и сопутствующая функция **OnEndPrinting()** позволяют распределить ресурсы, которые будут использоваться на протяжении всего процесса печати. Кроме того, они обеспечивают вторую возможность установки количества страниц в документе.

В функции **OnPreparePrinting()** доступ к контексту устройства принтера отсутствует, поэтому вы не сможете узнать размер печатной области. Получить эту информацию можно, использовав контекст устройства принтера в функции **OnBeginPrinting()** и вызвав из нее функцию **GetDeviceCaps()** с параметрами **VERTRES** или **HORZRES**. Функция **GetDeviceCaps()**, как и подсказывает ее имя, возвращает размер печатной области в пикселах. Для вычисления количества страниц потребуется преобразовать эти числа в логические координаты.

Функция OnPrepareDC()

Вы уже сталкивались с функцией **OnPrepareDC()**, которую MFC вызывает автоматически перед выводом каждой страницы, независимо от того, куда направляется вывод — на принтер или на дисплей. Функция **OnPrepareDC()** принимает те же два параметра, что и **OnBeginPrinting()**: указатель на DC и указатель на объект **CPrintInfo**. Если вывод выполняется на экран, объект **CPrintInfo** принимает значение **NULL**. Можно проверить это условие, однако проще вызвать функцию **CDC IsPrinting()**, сообщающую о том, куда направляется вывод — на экран или на принтер.

До сих пор вы уже, вероятно, поняли, почему следует побеспокоиться о том, куда направляется вывод — на экран или на принтер. Так будет ли использоваться один и тот же код для обоих выводов? И да, и нет. Если вывод направляется на принтер, то **OnPrepareDC()**, а также последующая функция **OnPrint()** будут вызываться для каждой из выводимых страниц. В функции **OnPrepareDC()** будет изменяться позиция печати и, вероятно, устанавливаться прямоугольник отсечения, чтобы обеспечить вывод корректной страницы. Для этого потребуется воспользоваться переменной **m_nCurPahe** объекта **CPrintInfo** и произвести соответствующую настройку.

Функция OnPrint()

Последней функцией из окружения печати, которую мы рассмотрим, будет **OnPrint()**. Стандартная версия **OnPrint()** просто вызывает **OnDraw()**. Если вам не нужны заголовки, отступы или нумерация страниц (и если в **OnPrepareDC()** установлена область просмотра и область отсечения), нет смысла перекрывать **OnPrint()**.

С другой стороны, если необходимо установить область отсечения, заголовки и отступы — сделайте это здесь. AppWizard не перекрывает эту функцию — сделайте это самостоятельно при помощи ClassWizard.

Пример MiniSketch

Итак, вы его получили. Просто и очевидно? Нет — мы полагаем нет. Но еще есть время на краткий пример. Это не сделает процесс более миловидным, но поможет лучше его осмыслить.

В последней версии MiniSketch с использованием **CScrollView** создавался документ форматом 8x10.5 дюймов. Давайте сделаем документ MiniSketch *потенциально* допускающим формат 24x21 дюйм. Вы будете распечатывать документы максимального размера — шесть страниц, т.е. три в ширину и две в высоту. Впрочем, сперва следует вычислить актуальный размер документа, а затем уже соответственно разбить его на страницы.

Для этого выполните следующие шаги:

1. Создание большого виртуального документа — довольно простая часть работы. Откройте проект MiniSketch и отыщите в окне ClassView функцию **CMSView::OnInitialUpdate()**. Щелкните на ней дважды для открытия окна редактора исходного кода и измените параметры, передаваемые в **SetScrollSizes()**, как показано в листинге 16.12.

Листинг 16.13. Функция CMSView::OnInitialUpdate().

```
void CMSView::OnInitialUpdate()
{
```

```
        InitPen();
        SetScrollSizes(MM_LOENGLISH, CSize(2400, 2100));
}
```

2. Для поддержки разбиения на страницы потребуется определить, сколько страниц из шести возможных на виртуальном холсте содержат фигуры. Если рисунки содержит только первая страница, нет необходимости печатать пять остальных. Добавьте в класс **CMSDoc** функцию **GetDocDimensions()**. Эта функция, показанная в листинге 16.14, проходит по каждому объекту **Shape**, хранимому в документе, и получает координаты наиболее удаленной от начальной точки документа фигуры. На основании полученной информации от каждого объекта Shape обновляются переменные **x** и **y**. (При обновлении **y** используется функция **min()**, поскольку в режиме отображения **MM_LOENGLISH** направление оси ординат — отрицательное.)

Листинг 16.14. Функция CMSDoc::GetDocDimensions().

```
CSize CMSDoc::GetDocDimensions()
{
      int x = 0, y = 0;
      CSize temp;
      int nItems = m_data.GetSize();
      for (int i = 0; i < nItems; i++)
      {
            temp = m_data[i]->GetMaxSize();
            x = max(x, temp.cx);
            y = min(y, temp.cy);
      }
      return CSize(x,y);
}
```

3. К каждому классу в иерархии **Shape** добавьте виртуальную функцию **GetMaxSize()**, возвращающую **CSize**. Для классов **Shape** и **FilledShape** верните значение **CSize** (0,0). Для других классов необходимо определить максимальное пространство, занимаемое **Shape**. В листинге 16.15 приведен код функции **GetMaxSize()** для класса **Line**. Листинги других функций можно найти на сопровождающем книгу CD-ROM.

Листинг 16.15. Функция Line::GetMaxSize().

```
CSize Line::GetMaxSize()
{
      int x = max(m_LineEnd.x, m_LineStart.x);
      int y = min(m_LineEnd.y, m_LineStart.y);
      return CSize(x,y);
}
```

4. После получения размеров документа, его можно использовать для вычисления необходимого количества страниц. Для хранения этой информации добавьте в класс **CMSView** следующие приватные переменные типа **int**:

   - **m_PageWidth** — Содержит ширину печатной страницы, получаемую из контекста устройства принтера.

- **m_PageHeight** — Содержит высоту печатной страницы, получаемую из контекста устройства принтера.
- **m_nPagesWide** Содержит количество страниц в ряду.
- **m_nPagesHigh** — Содержит количество рядов печатных страниц.

5. Вычислите, сколько страниц печатать. Сделать это в функции **OnPreparePtinting()** нельзя, поскольку у вас пока нет контекста устройства принтера. Посему, определение и установка количества страниц должно выполняться в **OnBeginPrinting()**.

   Вычисление количества страниц достаточно очевидно. Сперва необходимо узнать размер каждой страницы. (Эта величина сильно варьируется в зависимости от используемого принтера и выбранной ориентации бумаги.) Информация о размерах страницы получается из вызова **GetDeviceCaps()** и сохраняется в переменных **m_PageWidth** и **m_PageHeight**. Как только станет доступным размер каждой страницы, можно вычислить размер всего документа. Эти вычисления осуществляются в только что созданной функции **GetDocDimensions()** класса **CMSDoc**. Поскольку **GetDeviceCaps()** возвращает координаты устройства, а документ сохраняется в логических координатах, необходимо выполнить преобразование координат устройства в логические. Затем с помощью переменных **m_nPagesWide** и **m_nPagesHigh** можно вычислить количество необходимых строк и колонок. Далее вызывается функция **CPrintInfo SetMaxpage()** для сообщения механизму печати количества необходимых страниц.

   Добавьте в функцию OnBeginPrinting() код, показанный в листинге 16.16.

Листинг 16.16. Функция OnBeginPrinting().

```
void CMSView::OnBeginPrinting(CDC* pDC, CPrintInfo*
   pInfo)
{
      m_PageWidth   = pDC->GetDeviceCaps(HORZRES);
      m_PageHeight  = pDC->GetDeviceCaps(VERTRES);

      CSize docSize = GetDocument()->GetDocDimensions();
      CSize pageSize(m_PageWidth, m_PageHeight);
      pDC->SetMapMode(MM_LOENGLISH);
      pDC->DPtoLP(&pageSize);
      m_PageWidth   = pageSize.cx;
      m_PageHeight  = -pageSize.cy;

      m_nPagesWide  = 1 + docSize.cx / m_PageWidth;
      m_nPagesHigh  = 1 + ((docSize.cy) / m_PageHeight);

      pInfo->SetMaxPage(m_nPagesWide * m_nPagesHigh);
}
```

6. После уведомления механизма печати MFC о количестве печатных страниц, он обращается к функции **OnPrepareDC()** и **OnPrint()** по одному разу для каждой страницы. При вызове MFC одной из функций переменная **CPrintInfo m_nCurPage** инкрементируется. Эта переменная используется для определения номера печатаемой страницы. Перед обращением **OnDraw()** вызывается **SetWindowOrg()** для установки позиции печати. (Следует заметить, что этот способ печати не особо эффективен. При каждом вызове **OnDraw()**

печатается весь документ, хотя в контексте устройства принтера видима только часть, попадающая внутрь прямоугольника отсечения.)

Для переопределения виртуальной функции **OnPrint()** отыщите в окне ClassView класс **CMSView**. Щелкните правой кнопкой мыши для вызова контекстного меню, и выберите в нем Add Virtual Function. Поместите в тело функции код из листинга 16.17, и работа будет завершена.

Листинг 16.17. Функция OnPrint().

```
void CMSView::OnPrint(CDC* pDC, CPrintInfo* pInfo)
{
      int row = ((pInfo->m_nCurPage - 1) / m_nPagesWide);
      int col = ((pInfo->m_nCurPage - 1) % m_nPagesWide);
      pDC->SetWindowOrg(col * m_PageWidth, row *
         m_PageHeight);

      OnDraw(pDC);
}
```

Вот так. Откомпилируйте и запустите программу, и вы в ней сможете рисовать на виртуальном холсте в 24 дюйма шириной и в 21 дюйм высотой. Распечатка будет занимать столько страниц, сколько необходимо для отображения всего рисунка. При изменении размера бумаги или ее ориентации изменится количество необходимых страниц, но печать все же будет функционировать. На рис. 16.12 показано знакомое "неореалистическое" полотно "Яйца", выведенное в окне предварительного просмотра MiniSketch. Предположительно, это полотно первоначально создавалось по заказу кинорежиссера Альфреда Хичкока для включения его в финальную сцену фильма *"The Birds" ("Птицы")*. Как сообщают, Хичкок счел эту идею настолько ужасной, что художество, долго томившееся в пыли, вдруг вдохновило на создание таких фильмов, как *"Aliens" ("Чужие")*, и более свежую киноленту позже — *"Godzilla" ("Годзилла")*. Это все, конечно, молва, но вот оригинал полотна "Яйца" находится на сопровождающем книгу CD-ROM. Им можно воспользоваться для испытания возможностей печати в MiniSketch.

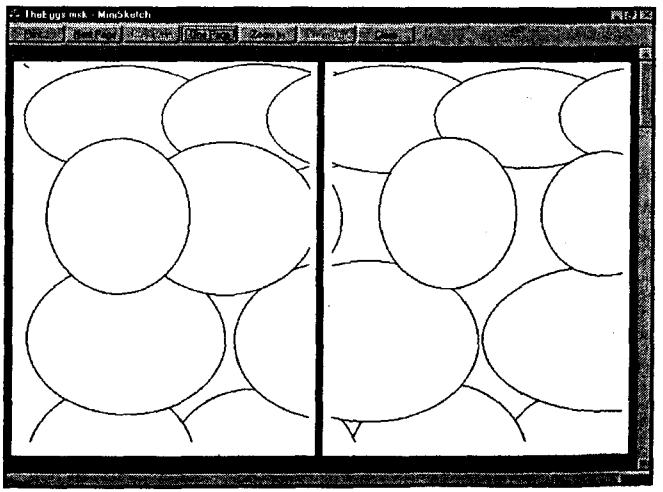


РИСУНОК 16.12.
Картина "Яйца" в окне предварительного просмотра MiniSketch.

Теперь о чем-то совершенно другом

В MiniSketch еще можно внести много улучшений, например, использовать стилизованные кисти и перья, добавить к прокрутке еще и масштабирование с помощью режимов отображения, или встроить различные диалоговые окна. Вы способны самостоятельно внести огромное количество улучшений — и мы надеемся, что вы сделаете это.

Для нас же наступил момент прощания с MiniSketch. В следующей главе мы вернемся к диалоговым окнам и рассмотрим остальные элементы управления Windows. Будет исследоваться работа с текстом и использование форматируемого текста. В конечном итоге мы перейдем к элементам управления ActiveX, базам данных и Internet.

Ну что, пошли?..

Глава 17

Повторное использование программного обеспечения: сборка приложения из компонентов

Если сфера деятельности программирования компьютеров не настолько молода, что может предъявлять права на любые "старые добрые пословицы", тогда древнейшим и почтеннейшим является, несомненно, предостережение "Не изобретай велосипед!". В течение многих лет уважение этой традиции было большей частью пустословием — казалось, что каждое новое поколение программистов было вынуждено вручную создавать новые варианты тех же наборов компонентов. Visual Basic все это изменил.

Шел 1990 год. Профессиональные программисты Windows высмеивали идею написания Windows-программ на Basic, "игрушечном языке, бесплатно поставляемом с PC". Как они выражались, если бы Богу было угодно, чтобы Windows-программы разрабатывались на Basic, Windows API не написалось бы на C. Кроме того, программисты Visual Basic вовсе не программируют в обычном смысле этого слова. Они просто перетаскивают маленькие пиктограммы, называемые компонентами, по экрану, и в диалоговых окнах устанавливают их свойства. Только в крайних случаях они прибегают к написанию кода, да и то маленькими обрывками. Профессиональные программисты не были в восторге.

Впрочем, две группы *были* под впечатлением. Сперва многие программисты бизнес-приложений, которые применяли продукты dBASE, Clipper или Business Basic для создания приложений для торговли, видеомагазинов и зубоврачебных кабинетов, сочли Visual Basic своим билетом в мир Windows. Затем несколько профессиональных программистов Windows увидели Visual Basic не как соперника, но как возможность поставки специализированных компонент программного обеспечения для растущего числа прикладных программистов на Visual Basic.

Трудно сказать, кто из них был первым — программисты бизнес-приложений, переключившиеся на Visual Basic, или разработчики программного обеспечения, предлагающие новые и усовершенствованные компоненты, — но эта комбинация привела к взрыву. Если бы Visual Basic не имел большого количества простых в использовании инструментальных средств программирования для Windows, огромный рынок компонент никогда не был бы развит. И если бы Visual Basic не был открытой системой, позволяющей компаниям писать работающие в нем компоненты, программисты никогда бы не приняли этот язык.

И, вероятно, в завершение можно сказать, что к успеху Visual Basic привели не масса программистов и не изобилие коммерческой поддержки — это сделали компоненты.

Первые элементы управления Visual Basic — называемые *элементами управления VBX* — были 16-разрядными и предназначенными для работы в Windows 3.1. Элементы управления VBX были тесно связаны с Visual Basic. С переходом операционных систем Microsoft от 16-разрядных к 32-разрядным Windows NT, Windows 95 и Windows 98, базовая архитектура элементов управления Visual Basic изменилась. Элементы управления VBX были заменены элементами управления OCX, известными сейчас как элементы управления ActiveX. Элементы управления VBX отличались от ActiveX многим, но главное различие состояло в том, что элементы управления ActiveX не были привязаны к Visual Basic. Основанные на модели многокомпонентных объектов (COM), элементы управления ActiveX теперь существуют во множестве средств разработки программного обеспечения для Windows.

В главе 1 было сказано, что Visual C++ не является средой визуального программирования типа "укажи и щелкни", в отличие от Visual Basic или Delphi. Вероятно, следует подчеркнуть, что Visual C++ является *не только* средой визуального программирования, поскольку революция компонентов, порожденная Visual Basic, оказала огромное воздействие и на Visual C++.

В этой и следующей главах мы намерены показать, как Visual C++ поддерживает разработку программного обеспечения компонентов. Visual C++ предоставляет более богатую среду, чем Visual Basic, поскольку поддерживает много видов раз-

работки. Это дважды верно в области разработки компонентов, где Visual C++ поддерживает два следующих вида компонентов:

- *Компоненты Visual C++*. Обеспечивают простое добавление в проект MFC компонентов программного обеспечения, таких как экраны-заставки, индикаторы хода работ и листы свойств.
- *Элементы управления ActiveX*. Работают в Visual C++ так же, как и в Visual Basic.

Начнем наше путешествие с галереи Visual C++ (Visual C++ Gallery).

Исследование галереи

Галерея элементов управления и компонентов Visual C++ предоставляет место для сохранения компонентов с целью их использования в дальнейшей работе. Впрочем, в галерее слово *компонент* имеет более широкое значение, чем объясненное нами до сих пор. Компонентом галереи может быть элемент управления ActiveX, но таковым может являться и Мастер, графический файл, шаблон диалогового окна или класс, реализованный для повторного использования. Можно приобрести исходный код компонентов Visual C++, специально предназначенных для работы с галереей.

При первоначальной установке Visual C++ галерея содержит две папки: одну, содержащую связи со всеми элементами управления ActiveX, зарегистрированными в системе, и другую, содержащую связи с компонентами Visual C++. При добавлении собственных классов или ресурсов потребуется создать для их хранения новую папку. Связи, содержащиеся в папках компонентов ActiveX и Visual C++, динамически обновляются при каждом открытии галереи.

Начнем путешествие по галерее с некоторых компонентов, содержащихся в папке Visual C++ Components. Чтобы помочь в освоении, вызовем на "бис" почтенную программу MiniSketch. После отправки в отставку MiniSketch мы начнем новый проект WordZilla, использующий элемент управления форматированным текстом.

Совершенствование MiniSketch

Visual C++ Gallery содержит компоненты, предназначенные для добавления к существующим компонентам. В этом смысле галерея более похожа на ClassWizard, чем на AppWizard. С другой стороны, галерея по определению является однонаправленным средством. Например, после добавления в проект панели заставки вы не сможете откатить эту операцию с помощью средств Gallery. Панель заставки придется вручную удалить из проекта. В этом смысле Gallery больше походит на AppWizard, чем на ClassWizard.

Мы намерены добавить в программу MiniSketch новую возможность — динамически разбиваемые окна. Полосы разбивки позволяют открывать второе, третье и даже четвертое представление для одного документа. Так как все они будут одним и тем же классом преставления, нет необходимости вносить в приложение новый код.

Вы готовы? Давайте приступим к разбивке. (Тяжкий вздох здесь не только разрешен, но и обязателен.)

Добавление полос разбивки

Как вы уже знаете, программа MiniSketch является SDI-программой, т.е. одновременно можно открывать не более одного документа. MiniSketch содержит один документ и одно представление. Полосы разбивки позволяют автоматически добавлять дополнительные представления в SDI-программу.

Поддержка полос разбивки осуществляется с помощью компонента Splitter Bar из Visual C++ Gallery. Для этого выполните следующие шаги:

1. Откройте проект MiniSketch. Полученный из галереи компонент автоматически добавляется в текущий проект. Если проект не открыт, то нельзя открыть и галерею компонентов.

2. Найти галерею компонентов непросто — она скрывается на несколько уровней ниже главного меню. Откройте ее, выполнив команду меню Project|Add To Project|Components and Controls. На рис. 17.1 показан путь к этой команде.

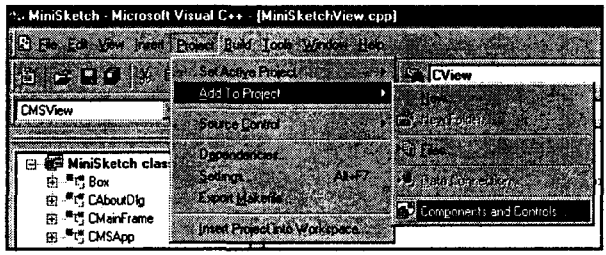


РИСУНОК 17.1. *Открытие галереи компонентов Visual C++.*

3. Открытие диалогового окна галереи компонентов и элементов управления займет некоторое время — Visual C++ просканирует систему и создаст список зарегистрированных компонентов Visual C++ и элементов управления ActiveX. Найдя компонент, Visual C++ сохраняет связь с ним в папках Registered ActiveX Controls или Visual C++ Components. По завершении откроется диалоговое окно, показанное на рис. 17.2.

4. Дважды щелкните на папке Visual C++ Components, чтобы вывести список компонентов, которые можно автоматически добавить в приложение. При поиске компонента Splitter Bar можно пользоваться полосами прокрутки (см. рис. 17.3).

СОВЕТ

Получение информации о компонентах

Следует отметить, что во время выбора компонента подсказка в нижней панели диалогового окна Components and Controls Gallery описывает выделенный элемент. Для большинства компонентов также доступна оперативная справка. Если кнопка More Info активна, то щелчок на ней приводит к выводу оперативной справки. На рис. 17.4 показана оперативная справка для компонента Splitter Bar.

5. После изучения оперативной справки вернитесь в диалоговое окно Components and Controls Gallery и щелкните на кнопке Insert. После этого Visual C++ запустит встроенную программу, которая запишет код, обеспечивающий добавление полосы разбивки в проект. Впрочем, прежде чем программа запишет код, ей необходимо получить некоторую информацию. На

рис. 17.5 показано диалоговое окно Splitter Bar. Как видите, можно добавить вертикальную (Vertical), горизонтальную (Horizontal) или обе (Both) полосы разбивки. Выберите переключатель Both.

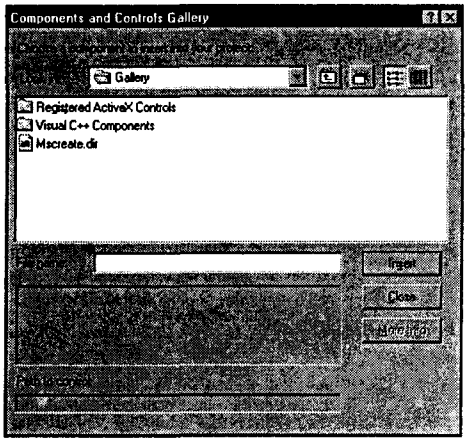


РИСУНОК 17.2. *Диалоговое окно Components And Controls Gallery.*

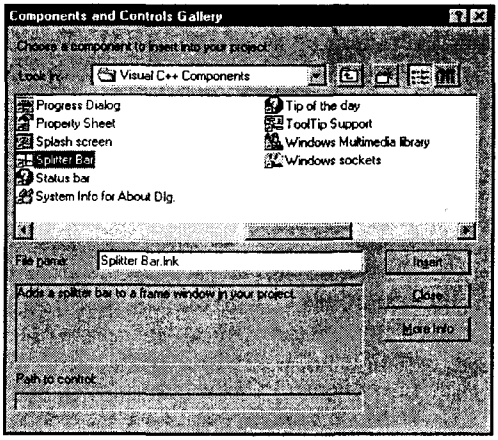


РИСУНОК 17.3. *Выбор компонента Splitter Bar из галереи компонентов.*

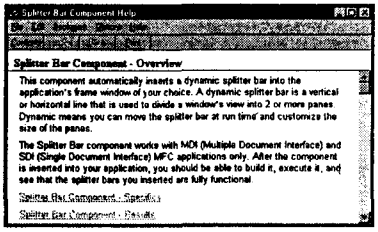


РИСУНОК 17.4. *Оперативная справка для компонента Splitter Bar.*

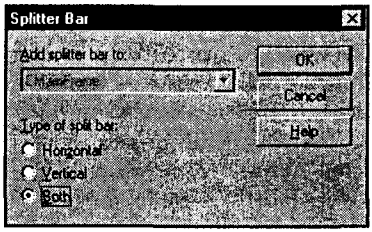


РИСУНОК 17.5. *Диалоговое окно Splitter Bar.*

Внутри компонента Splitter Bar

После щелчка на Insert Visual C++ делает две вещи. Во-первых, в класс **CMainFrame** добавляется защищенная переменная **m_wndSplitter**, представляющая собой объект **CSplitterWnd**. Затем перекрывается функция **CMainFrame::OnCreateClient()**, которая вызывается при создании клиентской области, т.е. представления. В добавленном Gallery коде создается объект **CSplitterWnd**, находящийся поверх окна **CMSView**, как показано в листинге 17.1.

Листинг 17.1. Создание CSplitterWnd в функции OnCreateClient().

```
BOOL CMainFrame::OnCreateClient(LPCREATESTRUCT lpcs,
    CCreateContext* pContext)
{
      // CG(галерея компонентов): Нижеследующий блок кода
      // добавлен компонентом Splitter Bar.
      {
            // ЧТО СДЕЛАТЬ: уточнить количество строк и колонок
            if (!m_wndSplitter.Create(this, 2, 2,
```

```
            // ЧТО СДЕЛАТЬ: уточнить минимальный размер панели
            CSize(10, 10),
                        pContext))
        {
            TRACE0("Failed to create splitter bar ");
            return FALSE;       // Ошибка создания
        }
        return TRUE;
    }
}
```

Несмотря на то что часть кода после "ЧТО СДЕЛАТЬ" (TODO) кажется позволяющей установить произвольное количество строк и колонок, на самом деле доступны только три варианта выбора: одна колонка и две строки, две колонки и одна строка, а также две колонки и две строки. Любая другая комбинация для динамически разделяемых окон, создаваемых галереей компонентов, некорректна. Gallery основывает начальные значения на активном переключателе в диалоговом окне Splitter Bar.

Окончательная доводка кода

После того как Gallery завершит добавление кода к проекту, ее диалоговое окно можно закрыть и без проблем перекомпилировать свою программу. К сожалению, если в этот момент попытаться выполнить программу, она не будет работать. За этим скрывается ошибка — но не в коде Gallery, а в некоторых упущениях, которые вы сделали при добавлении в панель состояния индикатора цвета пера.

Функция **InitPen()** в классе **CMSView** содержит такую строку:

```
((CMainFrame *)(GetParent())->SetPenColor(m_PenColor);
```

Этот код прекрасно работал до помещения полосы разбивки, однако впоследствии родителем объекта **CMSView** уже не является окно **CMainFrame**, поскольку в **OnCreateClient()** добавляется объект **CSplitterWnd**. Для устранения этой проблемы замените вызов функции **GetParent()** на **AfxGetMainWnd(),** которая всегда будет работать в приложениях SDI:

```
((CMainFrame *)AfxGetMainWnd())->SetPenColor(m_PenColor);
```

После внесения этого уточнения запустите программу MiniSketch. Маленькие полосы, появившиеся непосредственно над вертикальной полосой прокрутки и слева от горизонтальной, представляют собой полосы разбивки. Кроме того, при перемещении через полосу прокрутки изменяется курсор мыши. Если перетащить полосу при измененном виде курсора, как показано на рис. 17.6, можно разделить окно представления на вертикальные и горизонтальные панели.

Перетащив горизонтальные и вертикальные полосы разбивки, в документе можно создать четыре полунезависимых области экрана. Они являются только полунезависимыми, поскольку нет вертикальных и горизонтальных полос прокрутки для каждой из четырех панелей — панели, расположенные по вертикали, используют одну горизонтальную полосу прокрутки, а расположенные по горизонтали — одну вертикальную полосу прокрутки. На рис. 17.7 показана разбивка при работе с нашим рисунком. Каждая из четырех панелей на рисунке отображает один из четырех углов документа.

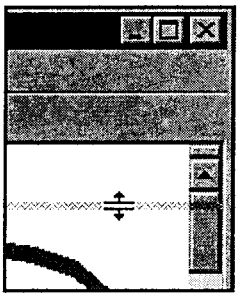


РИСУНОК 17.6. *Использование полосы разбивки.*

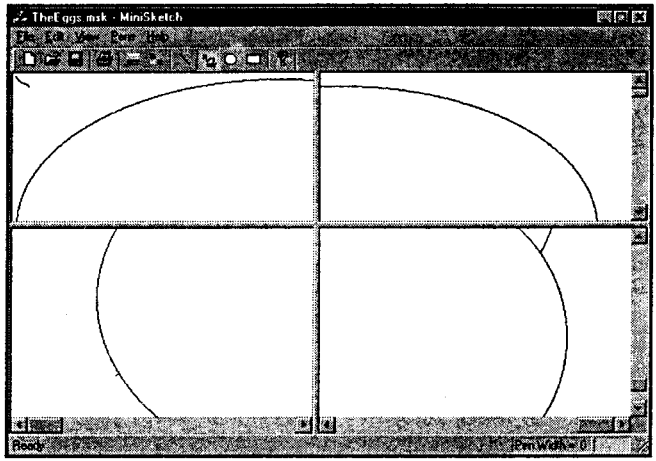


РИСУНОК 17.7. *Разделение картины "Яйца" на четыре части.*

Потратьте время на исследование компонентов Visual C++. На сопровождающем книгу CD-ROM вы найдете другую версию MiniSketch, с добавленной панелью заставки и системной информацией по компонентам Gallery. Теперь давайте рассмотрим кое-что новое.

Когда это кажется безопасным: WordZilla

К галерее компонентов Visual C++ мы вернемся в следующей главе. А сейчас рассмотрим другой метод повторного использования компонентов: приложения, построенные с помощью элементов управления Windows.

На самом деле вы уже встречались с приложением, которое расширяет один из встроенных элементов управления: программа NotePod была основана на классе **CEditView**, базирующемся на компоненте **CEdit**, который в свою очередь является просто оболочкой над элементом редактирования Windows.

Приложения, основанные на элементах управления отличаются от других приложений, построенных по технологии "документ/представление", следующим: вместо сохранения документа в виде внешней структуры данных, каждый элемент управления выступает как автономный модуль, хранящий свои собственные данные. В приложениях "документ/представление", основанных на компонентах, класс документа все еще существует, но его роль теперь состоит в передаче данных, а не в их хранении.

Давайте создадим многофайловый текстовый редактор наподобие NotePod, над которым вы работали в главе 1. Вместо класса **CEditView**, используемого в NotePod, возьмем класс **CRichEditView**, чтобы документ мог включать в себя различные стили текста и шрифты.

В связи с тем что наша программа будет более или менее напоминать WordPad, мы, вероятно, должны были бы следовать традиции, установленной в главе 1, и назвать ее WordPod. Программа NotePod добавила несколько новых возможностей к стандартной программе NotePad, и поэтому имя NotePod для нее было подходящим. Однако наш новый редактор будет не только использовать элемент фор-

матируемого текста — он будет выступать в роли экспериментального тестового стенда во время исследования мира элементов управления ActiveX. Если быть честными, наша программа, вероятно, будет казаться большим, отвратительным мутантом. Даже имя ее в таких обстоятельствах кажется походящим: WordZilla.

Сборка WordZilla: проект RichEditView

Создайте в AppWizard новый проект MFC и назовите его WordZilla. Затем выполните следующие шаги:

1. В окне MFC AppWizard — Step 1 определите построение проекта с многодокументным интерфейсом и поддержкой архитектуры "документ/представление".
2. В окне MFC AppWizard — Step 2 примите стандартные установки — без поддержки баз данных.
3. В окне MFC AppWizard — Step 3 выберите переключатель Container, щелкните на Yes для поддержки файлов составных документов и убедитесь, что флажок ActiveX Controls отмечен. Когда окно примет вид, показанный на рис. 17.8, щелкните на Next. (Если не добавить в приложение поддержку контейнеров составных документов, нельзя будет использовать элемент форматируемого текста.)

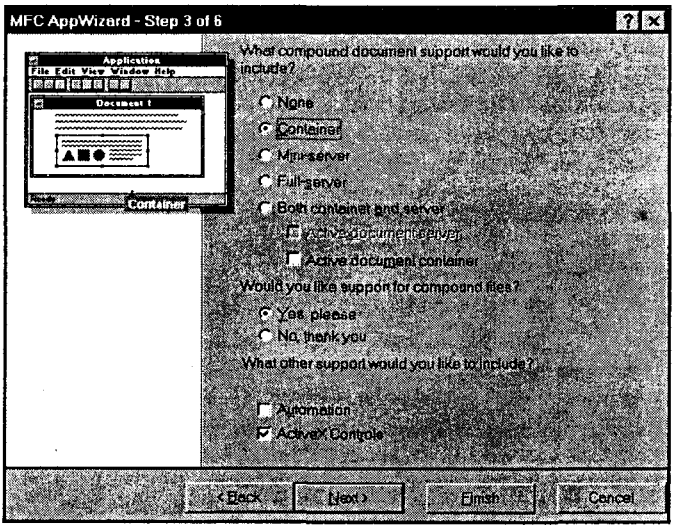


РИСУНОК 17.8.
Окно MFC AppWizard — Step 3 при создании WordZilla.

4. В окне MFC AppWizard — Step 4 примите стандартные значения, но щелкните на Advanced для открытия диалогового окна Advanced Options. Перейдите на вкладку Document Template Strings и введите wzi в поле File Extension. Установите Document Type Name в "WZilla", а все остальные вхождения "WordZi" замените на "WordZilla". Когда диалоговое окно будет выглядеть, как показано на рис. 17.9, щелкните на Close для возврата в окно Step 4. Затем щелкните на Next.
5. Примите стандартные установки в окне MFC AppWizard — Step 5.
6. В окне MFC AppWizard — Step 6 установите базовый класс представления в **CRichEditView**, используя выпадающий список Base Class. В именах клас-

сов слово "WordZilla" замените сокращением "WZ". Имена классов должны оказаться следующими: **CWZView**, **CWZDoc**, **CWZApp** и **CWZCntrlItem**. Имена классов **CMainFrame** и **CChildFrame** не изменяйте. Окно должно принять вид, как на рис. 17.10. В этом случае щелкните на Finish, а затем на OK.

Исследование WordZilla

После генерации AppWizard кода для WordZilla откомпилируйте и запустите программу. Как показано на рис. 17.11, программа работают практически так же, как и NotePod.

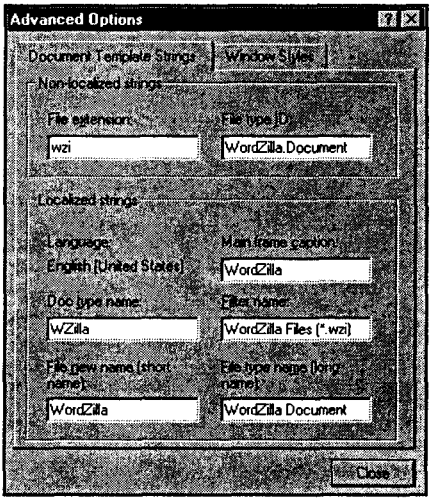


РИСУНОК 17.9.
Установки в диалоговом окне Advanced Options для проекта WordZilla.

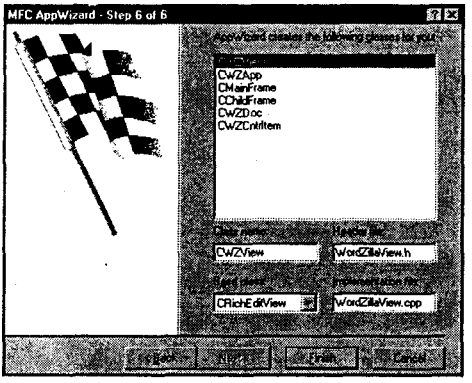


РИСУНОК 17.10. *Окно MFC AppWizard — Step 6.*

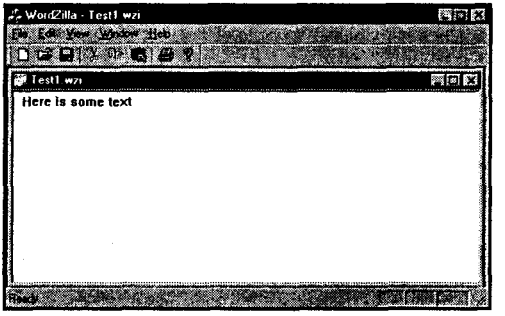


РИСУНОК 17.11. *Программа WordZilla в работе.*

Появляется возможность открывать несколько файлов, каждый в своем дочернем окне. Работает печать, предварительный просмотр, а также меню Window и Edit. Можно сохранять и открывать сохраненные документы. Все функциональные возможности NotePod в этом приложении уже присутствуют. Однако несмотря на внешнюю схожесть, WordZilla сильно отличается от NotePod.

Одно из различий связано с типом документа. NotePod, как и базовый его элемент **CEdit**, работают с обычным текстом ASCII. Элемент форматируемого текста, формирующий основу WordZilla, работает со специальным форматом документа, называемом Rich Text Format (RTF). Если открыть RTF-файл в обычном текстовом редакторе (см. рис. 17.12), можно заметить, что структура документа куда сложнее используемой в NotePod.

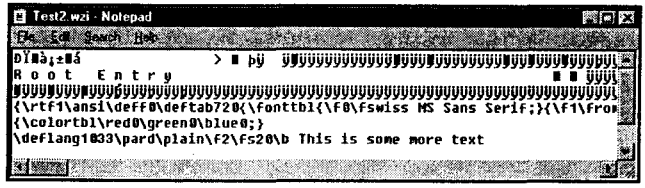


РИСУНОК 17.12. *Вид файла WZI в Notepad.*

Второе различие между NotePod и WordZilla можно увидеть, сравнив содержимое меню Edit. Тогда как меню NotePod содержит только опции Cut, Copy, Paste и Undo, меню WordZilla включает несколько других секций — команды Select All и Paste Special с опциями Find и Replace. Чуть ниже находится ключ к истинной мощи WordZilla: элемент меню Insert New Object, показанный на рис. 17.13.

В действительности элемент форматируемого текста содержит не только форматируемый текст. Он может содержать любой вид внедряемого объекта COM или OLE. Например, можно внедрить электронную таблицу Excel или рисунки Paint. При выборе команды Insert New Object, WordZilla ищет в системном реестре приложения, которые зарегистрировали свои типы документов. В окне Insert New Object приложение позволяет пролистать список доступных типов документов — это показано на рис. 17.14, где выбран документ Visio 5. Диалоговое окно Insert Object позволяет создать новый объект Visio непосредственно в вашем документе WordZilla. Также можно импортировать существующий файл, выбирая переключатель Create From File.

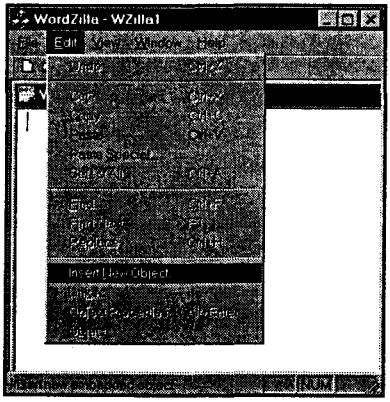


РИСУНОК 17.13. *Выбор опции Insert New Object.*

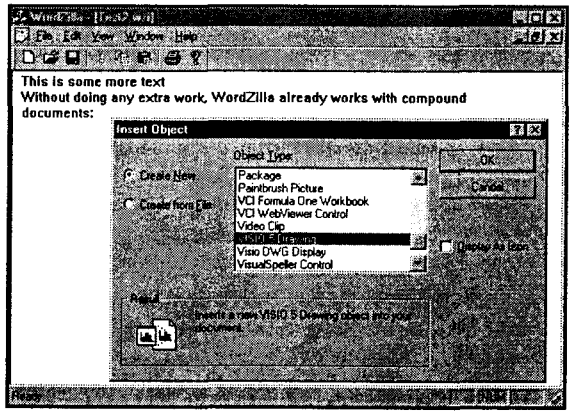


РИСУНОК 17.14. *Вставка нового рисунка Visio 5 в документ WordZilla.*

По нажатию OK диалоговое окно Insert Object запускает выбранное приложение внутри используемой вами программы. При вставке нового документа Visio в WordZilla запустится Visio. На время редактирования объекта меню Visio временно заменят обычные меню WordZilla (см. рис. 17.15).

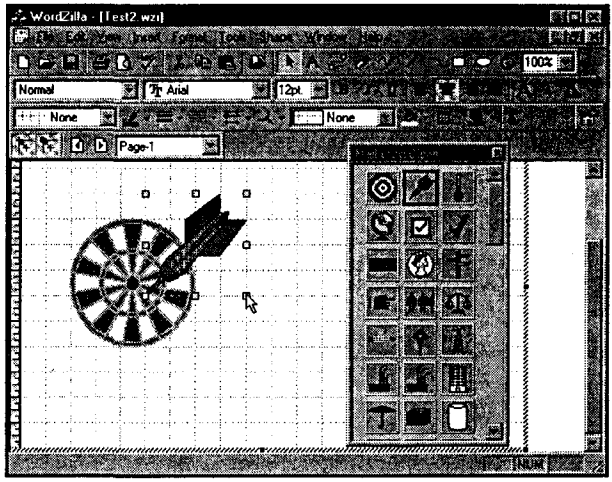


РИСУНОК 17.15.
Редактирование объекта Visio в приложении WordZilla.

По завершении редактирования COM-объект становится частью документа. Во время записи документа этот объект запишется как его часть. Объект можно распечатать и вообще относиться к нему так, как будто он создан вашим приложением. По двойному щелчку на внедренном объекте запускается его родное приложение и объект можно редактировать на месте. Как показано на рис. 17.16, форматировать, масштабировать, перемещать и выравнивать внедренный объект можно даже без вмешательства родного приложения.

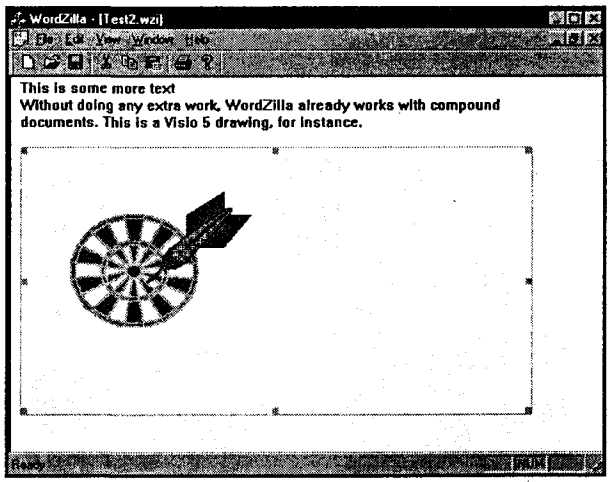


РИСУНОК 17.16.
Форматирование документа Visio 5 в приложении WordZilla.

Усовершенствование WordZilla

Несмотря на впечатляющие возможности, особенно возможность внедрения объектов, WordZilla страдает некоторыми недостатками. В нем используется такая же глупая система, как и в NotePod. Хотя можно вырезать, копировать и вставлять, нет способа изменить атрибуты выделенных символов. Кроме того, нельзя изменить выравнивание абзацев.

Прежде чем добавлять новые компоненты, давайте внесем в WordZilla следующие улучшения:

- Установим более привлекательный шрифт по умолчанию, например Arial.
- Добавим возможность выбора шрифта
- Разрешим пользователю изменять атрибуты выделенных символов — полужирный, курсив и подчеркнутый.
- Позволим устанавливать выравнивание абзацев — по правому краю, по левому краю и по центру.
- Добавим возможность добавления маркеров к абзацу.

Может показаться, что на осуществление этих возможностей уйдет масса времени, и это на самом деле так, если начинать все с самого начала. Но так как вы используете заготовленный компонент — элемент форматируемого текста — 99% работы уже сделано. Потребуется всего лишь подключить некоторые идентификаторы и написать немного кода.

Итак, займемся поочередно этими пунктами.

Установка шрифта по умолчанию

Написание программы, основанной на **CRichEditView**, довольно очевидно, однако следует обращать внимание на массу деталей. К сожалению, в документации непросто увидеть, какая информация более важна, а какая — менее. Несмотря на то что там есть вся необходимая информация, большинство ее скрывается за полем зрения и затеняется целым томом материалов.

Для изменения атрибутов нового текста, введенного в **CRichEditView**, служит метод **SetCharFormat()**. Если вызвать его из метода **OnInitialUpdate()**, установленные характеристики станут характеристиками по умолчанию.

Из-за большого количества текстовых опций **SetCharFormat()**, как и многие другие функции класса **CRichEditView**, принимают в качестве аргумента структуру **CHARFORMAT**. В структуре **CHARFORMAT** есть три важных поля, которые необходимо установить:

- **cbSize**. Определяет размер структуры в байтах. Вы должны установить это поле.
- **dwMask**. Сообщает, какие из оставшихся полей являются допустимыми. Например, если **dwMask** содержит значение **CFM_FACE**, то поле **szFaceName** — допустимо. Если **dwMask** не содержит значения **CFM_FACE**, функция **SetCharFormat()** проигнорирует значение поля **szFaceName**. Несколько значений в **dwMask** задаются с использованием операции **OR**.
- **dwEffects**. Определяет характеристики текста, такие как полужирный, курсив и подчеркнутый. Значение в **dwEffects** включает или отключает эти характеристики. Например, если **dwMask** имеет значение **CFM_BOLD**, указывающее на допустимость значения полужирного, хранимого в **dwEffects**, можно установить **dwEffects** в **CFE_BOLD** для разрешения характеристики полужирного или же в 0 — для ее отключения.

Для WordZilla используется 10-точечный шрифт Arial. Поле **dwMask** должно иметь значение **CFM_FACE | CFM_SIZE | CFM_BOLD**, что означает принятие во внимание имени шрифта, размера и характеристики полужирного, которая будет передана. Все остальное, за исключением **cbSize**, можно спокойно проигнорировать.

Добавьте код, выделенный в листинге 17.2, в метод **CWZView::OnInitialUpdate()**, чтобы каждый новый документ с самого начала использовал 10-точечный Arial.

Листинг 17.2. Функция CWZView::OnInitialUpdate().

```
void CWZView::OnInitialUpdate()
{
    CRichEditView::OnInitialUpdate();

    // Установить начальный шрифт
    CHARFORMAT cfm;
    cfm.cbSize = sizeof(cfm);
    cfm.dwMask = CFM_FACE | CFM_SIZE | CFM_BOLD;
    cfm.dwEffects = 0;   // Отключить полужирный
    cfm.yHeight = 200;   // Размер в TWIP (1/20)
    strcpy(cfm.szFaceName,"Arial");
    SetCharFormat(cfm);

    // Установить границы печати; 720 TWIP = 1/2 дюйма.
    SetMargins(720, 720, 720, 720);
}
```

Поле **dwEffects** установлено в 0 для запрета характеристики полужирного, которая включается по умолчанию. Эта установка необходима только в связи с тем, что унаследованный стандартный документ имеет включенную характеристику полужирного. Поле **yHeight** устанавливается в 200 TWIP. Каждый TWIP — это 1/20 точки, т.е. **yHeight** определяет 10-точечный размер шрифта.

Выбор нового шрифта

Установка шрифта по умолчанию была несложной, а обеспечение для пользователя возможности выбора любого желаемого шрифта даже еще проще. Необходимые функциональные возможности уже существуют в **CRichEditView** — потребуется лишь правильно их подключить. Добавим в панель управления кнопку, позволяющую изменять шрифт и цвет шрифта. Сейчас рассмотрим, как будет работать эта кнопка.

Когда пользовать нажимает кнопку выбора шрифта, появляется стандартное диалоговое окно Font, позволяющее выбрать любой установленный шрифт. Если пользователь выделит блок текста, а затем выберет новый шрифт, последний будет применен только к выделенному блоку текста. То же самое происходит и при нажатии кнопки выбора цвета шрифта. Если же выделенного текста нет, выбранный шрифт будет применяться к новым символам, которые будут набираться.

Выполните следующие шаги:

1. Откройте проект WordZilla и перейдите в секцию ResourceView. Откройте папку Toolbar и дважды щелкните на **IDR_MAINFRAME** для открытия Toolbar Editor.
2. В Toolbar Editor перетащите новую кнопку в позицию между кнопками Print и Paste. Отодвиньте ее от остальных кнопок, чтобы редактор панели управления поместил ее в отдельную группу.
3. С помощью инструмента Text из панели Graphics выберите 12-точечный полужирный Times New Roman и поместите в центре кнопки заглавную букву "A".

4. Дважды щелкните на левой панели Toolbar Editor для открытия диалогового окна Toolbar Button Properties. Выберите в выпадающем списке идентификатор ресурса **ID_FORMAT_FONT**. При желании можно добавить подсказку. В конечном итоге экран должен принять вид, как на рис. 17.17.

После добавления кнопки откомпилируйте и запустите программу. Идентификатор ресурса **ID_FORMAT_FONT** был заранее ассоциирован с функцией **OnFormatFont()**, определенной в классе **CRichEditView**, поэтому для ее подключения пользоваться ClassWizard не потребуется. Введите любой текст и выделите его. Как видите, можно легко пользоваться любым шрифтом с любым стилем.

Однако диалоговое окно Font предлагает ограниченный набор цветов для шрифта. Давайте добавим кнопку, открывающую стандартное диалоговое окно Windows Color. Выполните следующие шаги:

1. Откройте в Toolbar Editor панель **IDR_MAINFRAME** и поместите в нее новую кнопку справа от кнопки **ID_FORMAT_FONT**.
2. С помощью инструмента Text из панели Graphics поместите на кнопку красную букву "A".
3. Откройте диалоговое окно Toolbar Button Properties и введите новый идентификатор ресурса "**ID_COLOR_PICK**." Установите строку подсказки в "Изменение цвета текущего шрифта\nЦвет шрифта". Экран должен выглядеть, как показано на рис. 17.18.
4. Откройте ClassWizard и выберите страницу Message Maps. В выпадающем списке Class Name выберите класс **CWZView**. Найдите среди идентификаторов объектов **ID_COLOR_PICK** и выберите его. В списке Message выберите Command, щелкните на Add Function, а затем на Edit Code.
5. Добавьте код, показанный в листинге 17.3, в функцию **OnColorPick()**, созданную ClassWizard.

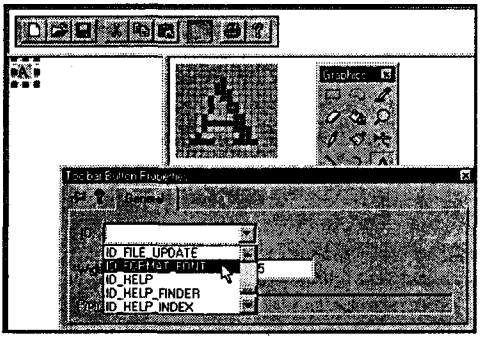


РИСУНОК 17.17. *Добавление кнопки панели управления ID_FORMAT_FONT.*

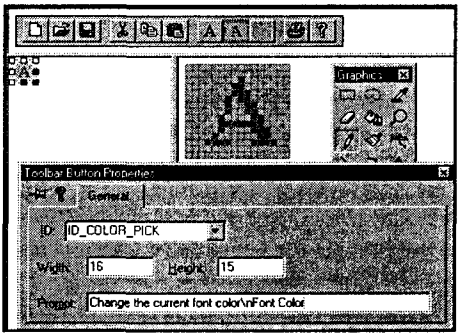


РИСУНОК 17.18. *Добавление в панель управления кнопки ID_COLOR_PICK.*

Листинг 17.3. Функция CWZView::OnColorPick().

```
void CWZView::OnColorPick()
{
   GetCharFormatSelection();
   CColorDialog dlg(m_charformat.crTextColor);
   if (dlg.DoModal() == IDOK)
   {
```

```
    m_charformat.dwMask = CFM_COLOR;
    m_charformat.dwEffects = NULL;
    m_charformat.crTextColor = dlg.GetColor();
    SetCharFormat(m_charformat);
  }
}
```

Класс **CRichEditView** содержит защищенный элемент данных **m_charformat**, содержащий текущие атрибуты текста. Это переменная типа **CHARFORMAT**, похожая на ту, которая применялась при установке стандартного шрифта в **OnInitialUpdate()**. Для использования переменной **m_charformat** сперва необходимо вызвать **GetCharFormatSelection()**, заполняющую каждое ее поле. Цвет текста хранится в поле **crTextColor**, которое можно использовать для инициализации объекта **CColorDialog**, как это делалось в предыдущих главах.

Как и ранее, объект **CColorDialog** отображается при помощи вызова функции **DoModal()**. Если **DoModal()** возвращает **IDOK**, значит пользователь выбрал новый цвет, и текущий атрибут цвета текста обновляется. Для этого сперва **m_charformat.dwMask** устанавливается в **CFM_COLOR**, чтобы функция **SetCharFormat()** приняла во внимание значение **crTextColor**. Затем **dwEffects** устанавливается в **NULL**, поскольку характеристики полужирного и курсива не применяются. Затем выбранный цвет получается из объекта **CColorDialog** и сохраняется в поле **crTextColor**. В заключение структура **m_charformat** передается обратно в **SetCharFormat()**.

Создание кнопок для установки атрибутов символов

Теперь, когда имеется возможность установить любой шрифт, давайте добавим в панель инструментов кнопки для установки атрибутов символов, как это имеет место в большинстве текстовых процессоров. Будут создаваться три кнопки: для полужирного, для курсива и для подчеркнутого. Каждая из этих кнопок будет работать как флажок — после щелчка она будет оставаться в нажатом состоянии до выполнения следующего щелчка.

В последнем разделе рассматривалось использование **GetCharFormatSelection()** и **SetCharFormat()** со структурой **CHARFORMAT** для изменения отдельных атрибутов выделенного текста. Однако к атрибутам в поле **dwEffects** существует даже более простой путь, предполагающий использование функции **OnCharEffects()**. **OnCharEffects()** принимает два параметра: значение для сохранения в **dwMask** и значение для сохранения в **dwEffects**. Функция **OnCharEffects()** самостоятельно позаботится о вызовах **GetCharFormatSelection()** и **SetCharFormat()**.

Один вызов

```
OnCharEffects(CFM_BOLD | CFM_ITALIC, CFE_ITALIC);
```

сделает шрифт в выделенном тексте курсивом и удалит его атрибут полужирного.

Выполните следующие шаги для подключения кнопок атрибутов символов:

1. Откройте в Toolbar Editor панель **IDR_MAINFRAME**. Между кнопками для выбора цвета и печати поместите три новых кнопки. Расположите их чуть дальше от остальных, чтобы сформировать отдельную группу.
2. С помощью инструмента Text выберите 10-точечный полужирный Times New Roman и поместите на первую кнопку букву "B". Повторите этот шаг и сместите вторую "B" так, чтобы из двух букв получился один жирный сим-

вол. Откройте диалоговое окно Toolbar Properties и установите имя идентификатора в **ID_CHAR_BOLD**. В поле подсказки введите "Сделать текущий выделенный текст полужирным\nПолужирный".

3. Измените шрифт инструмента Text на 10-точечный полужирный курсив Times New Roman и пометьте вторую кнопку буквой "I". С помощью инструмента Pencil (Карандаш) увеличьте верхнюю и нижнюю засечки символа на один пиксел в обоих направлениях. Задайте идентификатор **ID_CHAR_ITALIC** и установите поле подсказки в "Сделать текущий выделенный текст курсивом\nКурсив".

4. Аналогичные действия проделайте и с третьей кнопкой. Используйте 10-точечный полужирный Times New Roman, чтобы пометить кнопку символом "U". С помощью инструмента Pencil подчеркните символ одной жирной линией. Установите идентификатор кнопки в ID_CHAR_UNDERLINE, а поле подсказки — в "Сделать текущий выделенный текст подчеркнутым\nПодчеркнутый". На рис. 17.19 показано, как должно выглядеть диалоговое окно и панель инструментов.

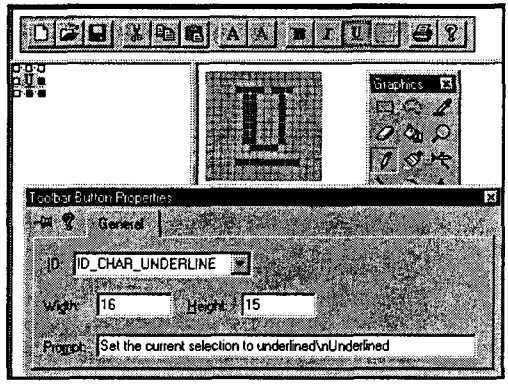


РИСУНОК 17.19. *Добавление кнопки ID_CHAR_UNDERLINE.*

Для кнопок атрибутов символов потребуется записать по две функции на каждую: обработчик команды и обработчик обновления интерфейса. Для ускорения откройте ClassWizard и убедитесь, что в выпадающем списке Class Name выбран класс CWZView. Затем просто добавьте для всех трех кнопок обработчики **COMMAND** и **UPDATE_COMMAND_UI**, безо всякого редактирования.

В каждую из функций добавьте код, показанный в табл. 17.1. Перекомпилируйте программу, после чего кнопки установки атрибутов символов панели инструментов заработают.

Таблица 17.1. Обработчики кнопок установки атрибутов символов панели инструментов.

Имя функции	*Тело функции*
OnCharBold	OnCharEffect(CFM_BOLD, CFE_BOLD);
OnUpdateCharBold	OnUpdateCharEffect(pCmdUI, CFM_BOLD, CFE_BOLD);
OnCharItalic	OnCharEffect(CFM_ITALIC, CFE_ITALIC);
OnUpdateCharItalic	OnUpdateCharEffect(pCmdUI, CFM_ITALIC, CFE_ITALIC);
OnCharUnderline	OnCharEffect(CFM_UNDERLINE, CFE_UNDERLINE);
OnUpdateCharUnderline	OnUpdateCharEffect(pCmdUI, CFM_UNDERLINE, CFE_UNDERLINE);

Создание кнопок для изменения атрибутов абзаца

В элементе форматируемого текста атрибуты шрифта и его цвета применяются к отдельным символам. При желании можно было бы создать документ, в котором каждый символ имел бы другой цвет. Но кроме этих атрибутов уровня символа, элементы форматируемого текста также поддерживают атрибуты уровня абзаца и страницы. Например, поля применяются ко всему документу — их нельзя установить конкретно для каждого абзаца или символа.

В панель инструментов WordZilla будут добавлены четыре кнопки: три для управления выравниванием абзаца — влево, вправо и по центру, и одна для обработки маркеров. Атрибуты выравнивания взаимоисключающи, поскольку абзац не может быть одновременно выровнен влево и вправо. Атрибут маркера можно комбинировать с остальными.

Вот шаги по добавлению кнопок атрибутов абзаца:

1. Откройте ресурс панели инструментов **IDR_MAINFRAME** в Toolbar Editor. Между кнопками подчеркивания и печати установите новую группу из трех кнопок.
2. С помощью инструмента Pencil нарисуйте на каждой кнопке в ряд шесть горизонтальных строк, на расстоянии одного пиксела друг от друга. Линии на каждой кнопке должны быть различной длины. На первой кнопке каждую линию рисуйте так, чтобы правое поле казалось неровным. На второй кнопке линии должны быть симметричны относительно центральной вертикальной оси. На последней кнопке начинайте каждую новую линию на 3 позиции дальше вправо, чтобы левое поле казалось неровным. При создании последней кнопки можно воспользоваться образцом — просто скопируйте и вставьте первую кнопку, а затем выберите опцию меню Image|Flip Horizontal. Все описанное иллюстрирует рис. 17.20.
3. В направлении слева направо задайте идентификаторы ресурса и строки подсказки, приведенные в табл. 17.2.

Таблица 17.2. Свойства кнопок атрибутов абзаца на панели инструментов.

Идентификатор ресурса	*Строка подсказки*
ID_PARA_LEFT	"Выравнивание параграфа влево\nВыровнять влево"
ID_PARA_CENTER	"Выравнивание параграфа по центру\nВыровнять по центру"
ID_PARA_RIGHT	"Выравнивание параграфа вправо\nВыровнять вправо"

4. С помощью ClassWizard (как в случае с атрибутами символов) добавьте для каждой из трех кнопок обработчик **COMMAND** и **UPDATE_COMMAND_UI**. В табл. 17.3 сведены имена обработчиков вместе с их содержимым.

Таблица 17.3. Обработчики кнопок атрибутов абзаца в панели инструментов.

Имя функции	*Тело функции*
OnParaLeft	OnParaAlign(PFA_LEFT);
OnUpdateParaLeft	OnUpdateParaAlign(pCmdUI, PFA_LEFT);
OnParaCenter	OnParaAlign(PFA_CENTER);
OnUpdateParaCenter	OnUpdateParaAlign(pCmdUI, PFA_CENTER);
OnParaRight	OnParaAlign(PFA_RIGHT);
OnUpdateParaRight	OnUpdateParaAlign(pCmdUI, PFA_RIGHT);

Добавление маркированного стиля абзаца

Добавление кнопки и обработчика для применения маркированного стиля абзаца даже проще, поскольку в классе **CRichEditView** этот стиль уже реализован. Вот что потребуется сделать:

1. В Toolbar Editor вставьте еще одну кнопку между кнопками печати и выравнивания вправо. Медленно ее переместите, чтобы она оказалась в отдельной группе, а не присоединенной к кнопкам выравнивания абзацев или кнопке печати.
2. Нарисуйте в ряд черные крестики и темно-синие линии, как показано на рис. 17.21. Они выглядят, как маркированные строки текста, уменьшенные до размера кнопки панели инструментов.
3. Определите идентификатор ресурса как **ID_BULLET** и введите подсказку "Установить маркер для текущего абзаца\nМаркеры".
4. С помощью ClassWizard добавьте обработчики **COMMAND** и **UPDATE_COMMAND_UI** и поместите в них код, выделенный в листинге 17.4.

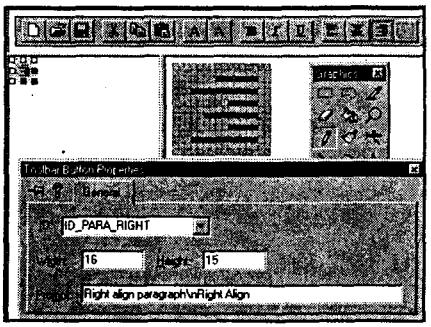


РИСУНОК 17.20. *Добавление в панель инструментов кнопок атрибутов абзаца.*

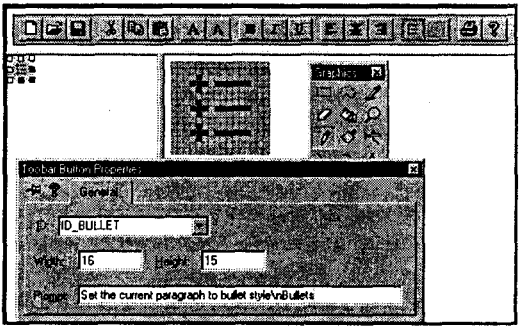


РИСУНОК 17.21. *Рисование кнопки атрибута маркирования в панели инструментов.*

Листинг 17.4. Обработчики кнопки атрибута маркирования в панели инструментов.

```
void CWZView::OnBullet()
{
    CRichEditView::OnBullet();
}

void CWZView::OnUpdateBullet(CCmdUI* pCmdUI)
```

```
{
    CRichEditView::OnUpdateBullet(pCmdUI);
}
```

Откомпилируйте и запустите WordZilla на выполнение. Все кнопки должны работать, поэтому к символам и абзацам можно применять различные шрифты и стили. На рис. 17.22 показано использование в WordZilla нескольких доступных стилей.

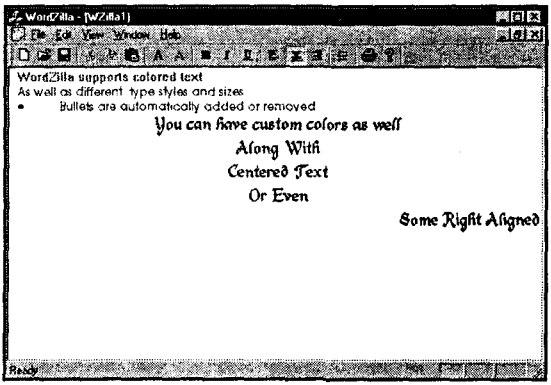


РИСУНОК 17.22.
WordZilla с различными стилями символов и абзацев.

ActiveX в панели диалога

В этой главе вы познакомились с тремя способами повторного использования кода. Во-первых, компоненты исходного кода, хранимые в папке галереи Visual C++ Components, сократили работу по добавлению в программу MiniSketch панелей разбивки. Затем был расширен один из стандартных элементов управления Windows — в данном случае, элемент форматируемого текста, — в результате чего появилась основа для целого приложения. В заключение были показаны заранее написанные компоненты, предназначенные для выполнения определенных задач (такие как **CFontDialog** и **CColorDialog**), которые избавляют от скучной работы, свойственной выполнению подобного рода общих задач.

Однако лучшее все еще впереди. Вспомните тысячи элементов управления ActiveX, которые мы с вами обсуждали в начале главы? ActiveX — это уже не Visual Basic; программисты Visual C++ теперь могут воспользоваться преимуществами той же революции компонентов.

В следующей главе будет разрабатываться панель диалога WordZilla и применяться некоторые элементы управления ActiveX.

Глава 18

ActiveX и приложения, основанные на компонентах

При последней покупке переносного телевизора или кофеварки вы не вызывали электрика, чтобы установить ее. Вместо этого вы просто включили прибор в одну из стандартных электрических розеток, расположенных повсюду в доме.

Стандарты делают возможной промышленность приборов массовой продажи. Если вы хотите производить успешные линии по изготовлению попкорна или фены, неплохо убедиться, что ваши вилки подходят к стандартным розеткам со стандартным напряжением. Большинство потребителей не пожелают устанавливать специальную электросеть, насколько бы революционным ни было ваше приспособление.

Не только электроприборы являются стандартизованными предметами домашнего хозяйства: ваша ванная имеет стандартные размеры и соединения, и вы можете легко заменить текущий водопроводный кран, используя стандартную деталь. Ваш телефон использует стандартное соединение; то же самое относится и к телевизору. Фактически благодаря стандартизации на вашем столе стоит такой мощный и недорогой компьютер. Производители аппаратных компонентов могут массово выпускать высокоэффективные видео- или аудиоподсистемы благодаря интерфейсам — шина PCI, к примеру, стандартизована и хорошо определена. В результате производители компьютеров могут легко собирать окончательный продукт, используя стандартизованные детали.

Индустрия программного обеспечения разрабатывалась в другом направлении. Рынок долго существовал для специализированных библиотек функций в областях коммуникаций, численного анализа и графики. Несмотря на это, большинство программистов до последнего времени писали вручную каждую строку каждой программы. По причине отсутствия стандартного метода соединения компонентов программного обеспечения, программисты мало их использовали. Когда появился Visual Basic (VB), он доказал людям возможность существования "программной шины".

Однако компоненты VB были слишком сильно привязаны к VB. Создав многокомпонентную модель объектов и основанную на ней технологию ActiveX, Microsoft попыталась предложить для компонентов более универсальный и общий способ соединения и общения с другими.

В этой главе мы посмотрим на компоненты с точки зрения требований Visual C++. VB предлагает один вид формы и один вид компонента. В Visual C++ имеется возможность выбирать из нескольких видов компонентов:

- *Оригинальные элементы управления Windows.* Такие элементы включают кнопки, флажки и полосы прокрутки.
- *Общие элементы управления Windows.* Они включают элементы управления, первоначально включенные в Windows 95, а также в Internet Explorer.
- *Компоненты с исходным кодом.* Включают в себя элементы управления, размещенные в папке галереи компонентов
- *Компоненты ActiveX.* Такие элементы управления работают во многих средах.

Вдобавок ко множеству компонентов, Visual C++ включает широкий ряд контейнеров, которые могут хранить эти компоненты. Компоненты можно использовать в диалоговых приложениях, подобных создаваемым в первой части книги. Кроме того, их можно применять в модальных и немодальных диалоговых окнах, а также в окнах свойств и панелях диалога. Можно даже использовать специальный класс **CFormView**, специально предназначенный для хранения компонентов.

Чтобы прочувствовать, как работают эти варианты, добавим в WordZilla команду меню Paste Date, решая одну проблему несколькими способами. Мы добавим

новый элемент управления Windows **CDateTimePicker** и сравним его с ActiveX-версией этого компонента. Также вы узнаете, как использовать эти элементы управления в модальных и немодальных диалоговых окнах. Одна и та же задача, решенная несколькими способами поможет вскрыть все "за и против" каждого подхода.

Вы готовы? Давайте приступим к работе.

WordZilla получает диалоговое окно

Вы уже знакомы с редактором диалоговых окон Visual C++, поскольку широко его использовали при сборке диалоговых приложений в главах с 1-й по 10-ю. При переходе к приложению DocView, вы оставили редактор диалогов безработным, используя его лишь для создания кнопок панели инструментов.

Впрочем, большинство приложений "документ/представление" используют диалоговые окна для широкого спектра задач, от выбора опций документа в окне свойств до поиска и замены текста. Простейшем видом, с которого мы и начнем, является *модальное* диалоговое окно.

Модальное диалоговое окно появляется по команде меню или панели инструментов, позволяя пользователю выполнить какое-то действие либо ввести информацию. Прежде чем программа сможет продолжать выполнение, пользователь должен закрыть это диалоговое окно. Стандартные диалоговые окна Windows Print и Color являются хорошими примерами модальных диалоговых окон — необходимо выбрать принтер или цвет, прежде чем шоу сможет продолжаться.

Первая версия команды Paste Date в приложении WordZilla использует модальное диалоговое окно, позволяющее выбрать дату с помощью компонента **CDataTimePicker**. Если вы закроете диалоговое окно, щелкнув на OK, программа вставит отмеченную дату в документ. (Формально дата *заменит* текущее выделение. Если ничего не выделено, WordZilla просто вставит дату в текущую позицию курсора.) Если закрыть диалоговое окно щелчком на Cancel, содержимое документа не изменится.

Как обычно, мы сначала создадим программу, а затем рассмотрим, как она работает.

Рисование диалогового окна Select a Date

Для добавления нового компонента к WordZilla необходимо пройти следующие шаги:

- Сперва при помощи редактора диалогов конструируются и размещаются компоненты пользовательского интерфейса.
- Затем в ClassWizard создается новый класс C++, привязанный к ресурсу диалога, созданного в Dialog Editor. Кроме того, в ClassWizard создаются переменные для передачи данных, предназначенные для получения и внесения информации в диалоговое окно.
- В заключение, с использованием редактора меню добавляется новая команда меню, которая затем связывается с функцией-обработчиком сообщения **COMMAND** в ClassWizard. Функция обработки отобразит диалоговое окно; затем, если окно будет закрыто щелчком на OK, программа получит от диалогового окна новую дату и вставит ее в текущий документ.

Начнем с рисования нового диалогового окна

1. Откройте проект WordZilla и добавьте новый шаблон диалога, выбрав в главном меню Insert|Resource. Выберите Dialog в списке Resource Type, как показано на рис. 18.1, и щелкните на New.
2. Найдите в панели инструментов Controls элемент управления Date Time Picker (см. рис. 18.2). Перетащите и установите его вблизи верхней части диалогового окна.
3. Откройте диалоговое окно Data Time Picker Properties, отметив компонент, щелкнув на нем правой кнопкой мыши и выбрав из контекстного меню команду Properties. (В качестве альтернативного способа, можно отметить компонент и нажать Enter.) Перейдите на вкладку Styles и из выпадающего списка Format выберите Long Date.
4. Поскольку длинный формат даты требует больше места, расширьте диалоговое окно и сам элемент управления. Затем укоротите его так, чтобы не терять место внизу. Окончательное расположение показано на рис. 18.4.
5. В диалоговом окне Dialog Properties введите идентификатор "**IDD_PICK_DATE**" и заголовок "Select A Date", как показано на рис. 18.5.

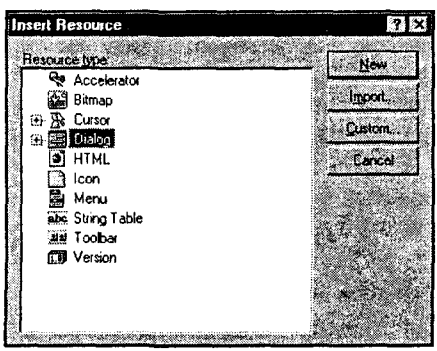


РИСУНОК 18.1. *Диалоговое окно New Resource.*

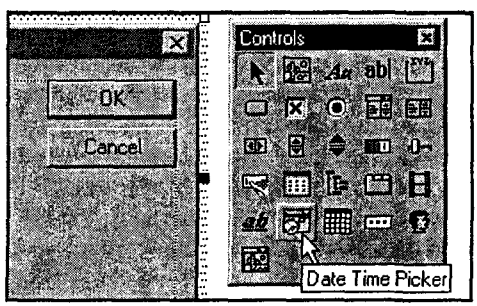


РИСУНОК 18.2. *Элемент управления Data Time Picker.*

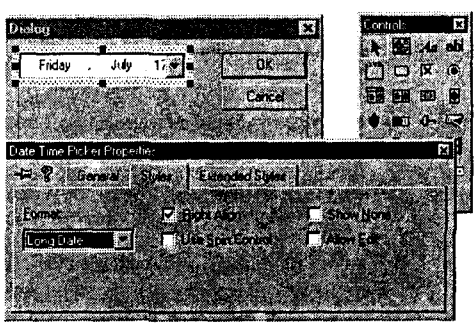


РИСУНОК 18.3. *Изменение свойства Format элемента управления Date Time Picker.*

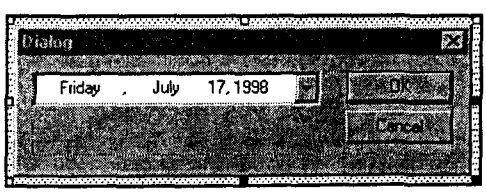


РИСУНОК 18.4. *Изменение размеров панели диалога.*

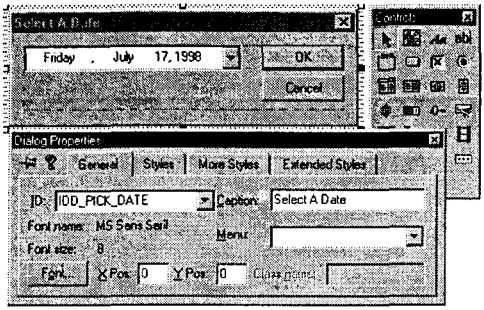


РИСУНОК 18.5. *Изменение идентификатора и заголовка ресурса.*

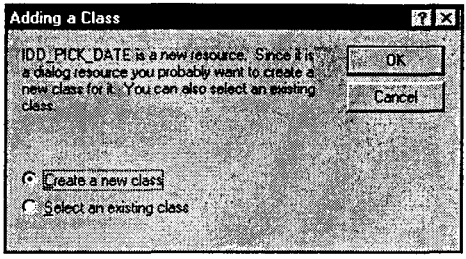


РИСУНОК 18.6. *Диалоговое окно Adding a Class для IDD_PICK_DATE.*

Создание класса диалогового окна

Для работы с диалоговым окном обычно создается новый класс, содержащий общедоступные элементы данных, которые используются для получения и внесения информации в диалоговое окно. Новый класс наследуется от **CDialog**.

Если элементы данных, объявленные общедоступными, стесняют вас (как это и должно происходить), их можно сделать приватными и реализовать необходимые функции доступа и изменения. Однако последнее придется сделать самостоятельно, поскольку элементы данных, создаваемые ClassWizard являются общедоступными.

Создайте класс диалога, выполнив следующие шаги:

1. Вызовите ClassWizard, нажав Ctrl+W или выбрав из главного меню View | ClassWizard. ClassWizard заметит, что новое диалоговое окно создано, но класс для него еще не объявлен. В результате ClassWizard отобразит диалоговое окно Adding a Class (Добавление класса), показанное на рис. 18.6.
2. Выберите переключатель Create a New Class (Создать новый класс) и щелкните на OK. ClassWizard выведет новое окно New Class. Введите в поле Name имя **"CPickDateDlg"**, а для всех остальных полей примите значения по умолчанию. Щелкните на OK для возврата в ClassWizard.

На рис. 18.7 показана вкладка Message Maps из ClassWizard для класса **CPickDateDlg**. Заметьте, что класс уже имеет одну виртуальную функцию: **DoDataExchange()**. Эта функция автоматически передает информацию от компонентов наподобие элемента управления Date Time Picker в переменные, поэтому программа может получить доступ к этим компонентам.

Сейчас мы создадим несколько переменных и вы увидите, как работает механизм передачи данных диалогового окна. Выполните следующие шаги:

1. В открытом ClassWizard перейдите на вкладку Member Variables. Выберите идентификатор ресурса **IDC_DATETIMEPICKER1** (см. рис. 18.8). Щелкните на Add Variable.
2. Когда откроется диалоговое окно Add Member Variable, введите в поле Member Variable Name имя **"m_DateValue"**. Из выпадающего списка Category выберите Value, а из выпадающего списка Variable type — **COleDateTime**. Когда окно будет выглядеть как показано на рис. 18.9, щелкните на OK.

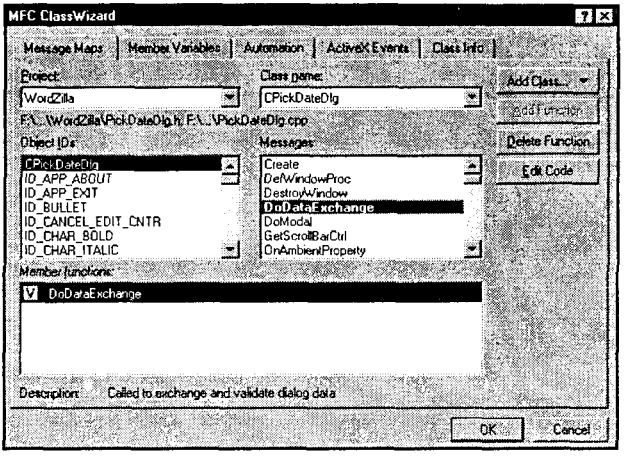


РИСУНОК 18.7.
Класс CPickDateDlg в ClassWizard.

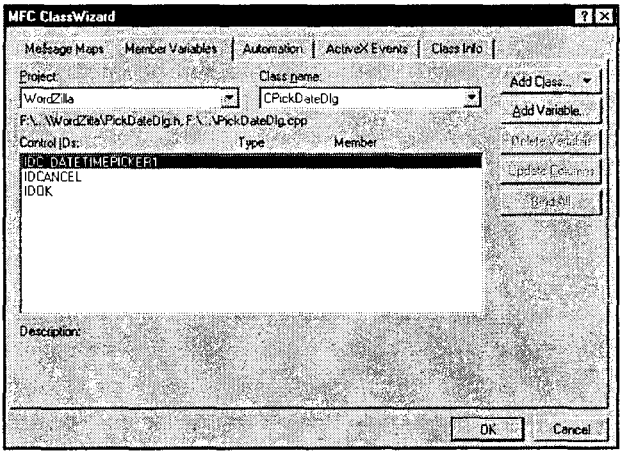


РИСУНОК 18.8.
Переменные ClassWizard для класса CPickDateDlg.

Подключение диалогового окна

Сделано все необходимое для использования нового диалогового окна Select A Date. Осталось только добавить команду меню и в ее обработчике вывести это диалоговое окно. Выполните следующие шаги:

1. Выберите в окне Workspace вкладку ResourceView и разверните папку Menu. Дважды щелкните на меню **IDR_WZLLATYPE**. MFC будет отображать это меню, когда активен документ WordZilla. Поскольку вставлять дату необходимо только тогда, когда документ активен, добавлять команду в меню **IDR_MAINFRAME** нельзя.
2. Откройте меню Edit и перетащите новый элемент меню ниже элемента Paste Special. Установите заголовок "Paste &Date" и идентификатор ресурса "**ID_EDIT_PASTE_DATE**". Поле подсказки должно содержать текст "Выбрать и вставить дату в документ\nPaste Date". Когда ваш экран будет выглядеть, как показано на рис. 18.10, можно продолжать.

3. При выбранной опции меню Paste Date вызовите ClassWizard, нажав Ctrl+W. Перейдите на вкладку Message Maps. В списке Object IDs уже должна быть выбрана ID-EDIT-PASTE-DATE; если же нет — выберите ее. Убедитесь, что в списке Name выбран класс **CWZView** (класс, в котором будет обрабатываться команда меню). Выберите **COMMAND** из списка Messages и щелкните на Add Function. В открывшемся диалоговом окне Add Member Function примите имя, предлагаемое ClassWizard — **OnEditPaste** (см. рис. 18.11). Щелкните на Edit Code.

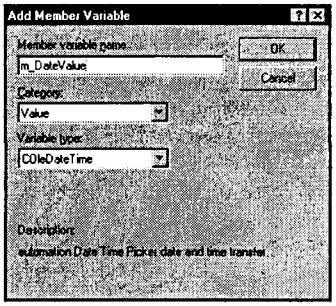


РИСУНОК 18.9. *Добавление переменной m_DateValue в класс CPickDateDlg.*

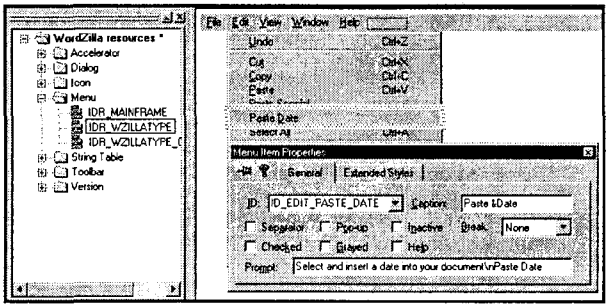


РИСУНОК 18.10. *Свойства опции меню Paste Date.*

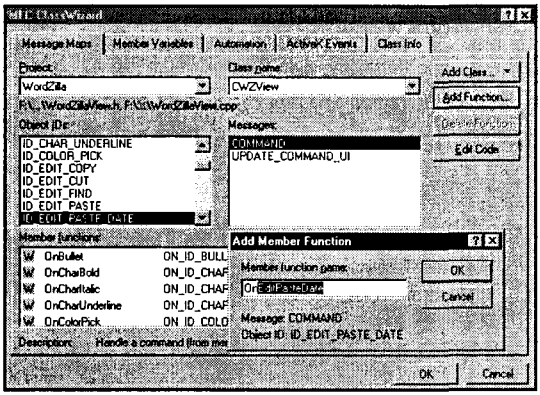


РИСУНОК 18.11. *Добавление функции OnEditPasteDate() в класс CWZView.*

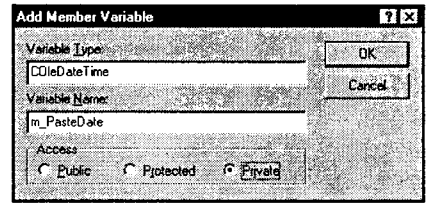


РИСУНОК 18.12. *Добавление переменной m_PasteDate.*

4. Код функции **OnEditPasteDate()** находится в листинге 18.1. Несколько позже мы кратко обсудим, как работает этот код.

Листинг 18.1. Обработчик команды меню CWZView::OnEditPasteDate().

```
void CWZView::OnEditPasteDate()
{
    CPickDateDlg dlg;
    dlg.m_DateValue = m_PasteDate;
    if (dlg.DoModal() == IDOK)
    {
        m_PasteDate = dlg.m_DateValue;
```

```
            CString sDate = m_PasteDate.Format("%B %d, %Y");
            GetRichEditCtrl().ReplaceSel(sDate);
        }
}
```

5. Кроме переменной даты, хранимой в классе **CPickDateDlg**, потребуется переменная для хранения даты, когда диалоговое окно неактивно. Перейдите во вкладку ClassView и отыщите класс **CWZView**. Щелкните правой кнопкой мыши для открытия контекстного меню и выберите в нем Add Member Variable. После открытия диалогового окна заполните его поля, как показано на рис. 18.12. Именем переменной должно быть **m_PasteDate**, типом — **COleDateTime**, а типом доступа — **private**.
6. В ClassView дважды щелкните на функции **CWZView::OnInitialUpdate()**. В конце функции допишите следующий код:

```
// Инициализируем с использованием текущей даты
m_PasteDate = COleDateTime::GetCurrentTime();
```

7. В начале открытого файла WordZilla.cpp после серии операторов **#include** добавьте следующую строку:

```
#include "PickDateDlg.h"
```

Откомпилируйте и запустите WordZilla. При выборе опции меню Edit | Paste Date открывается диалоговое окно Select a Date. Выберите компонент Date Time Picker, чтобы вывести календарь, как показано на рис. 18.13. Для движения по календарю можно использовать клавиши со стрелками, а стрелки по обеим сторонам панели позволяют изменить текущий месяц. Щелчок на дате приводит к закрытию календаря и к изменению даты в элементе Date Time Picker.

Можно сколько угодно раз открывать и закрывать Date Time Picker, но диалоговое окно остается открытым, пока не будет выполнен щелчок на OK или Cancel. По нажатию OK WordZilla вставит новую дату в документ; при следующем выборе в этом же документе опции Edit | Paste Date, элемент Date Time Picker будет первоначально содержать дату, выбранную вами в предыдущий раз.

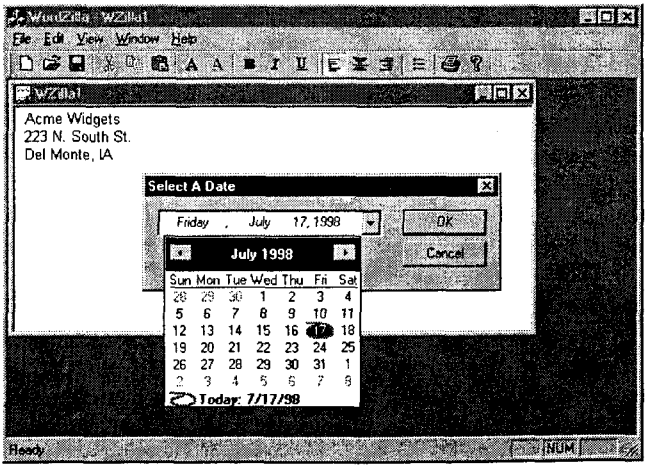


РИСУНОК 18.13.
Использование диалогового окна Select a Date.

Как работают модальные диалоговые окна

На первый взгляд код **CWZView::OnEditPasteDate()** кажется простым и очевидным. Благодаря своей простоте, он предоставляет хорошую возможность для понимания того, как работают диалоговые окна в целом, и модальные диалоговые окна — в частности. Позже вы увидите, что внешняя простота обманчива: многое происходит за кулисами. На рис. 18.14 схематически изображен жизненный цикл модального диалогового окна с использованием в качестве примера кода функции **OnEditPasteDate()**. Мы пронумеровали каждый шаг, чтобы в следующем обсуждении вы могли использовать эту иллюстрацию как дорожную карту.

Конструирование диалогового окна

В MFC при конструировании обычного окна создается переменная одного из классов **CWnd** и затем вызывается его функция **Create()**. Диалоговые окна не работают таким же образом. Диалоговые окна являются окнами, унаследованными от **CWnd**, подобно окнам **CFrameWnd**, но создаются они "на лету" с помощью данных, содержащихся в скомпилированном сценарии ресурса. Итак, вы просто создаете переменную диалогового окна, как показано в шаге 1 на рис. 18.14.

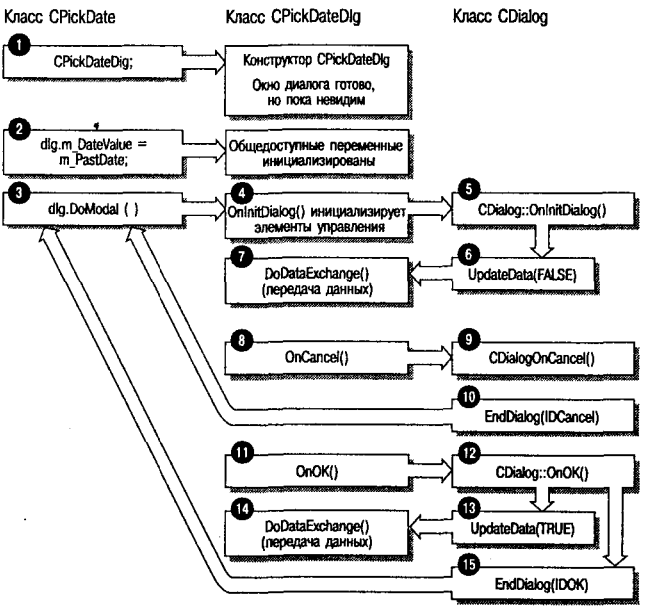


РИСУНОК 18.14.
Жизненный цикл модального диалогового окна.

При конструировании экземпляра нового класса диалога класс вызывает функцию **CDialog::OnCreate()** для чтения из сценария ресурса спецификаций диалогового окна, а затем создает и размещает каждый из его компонентов. Следовательно, переопределить метод **OnCreate()**, как это делалось в классе **CMainFrame**, нельзя — вам необходим другой метод инициализации элементов управления окна диалога. Во время выполнения метода **OnCreate()** элементы управления пока не существуют.

Инициализировать элементы данных можно после того, как конструктор завершит выполнение, как показано в шаге 2 на рис. 18.14. К этому времени в памяти объект диалогового окна будет существовать, но пока будет невидим. В функции **OnEditPaste()**, например, переменная **m_DataValue** класса **CPickDateDlg** инициализируется прямо за счет присвоения значения, хранимого в **m_PasteDate** (переменная класса **CWZView**).

Вывод диалогового окна

Для вывода диалогового окна вызывается его функция **DoModal()**, которая генерирует цепочку событий, показанную в шаге 3 на рис. 18.14. Сперва **DoModal()** вызывает виртуальную функцию **CDialog::OnInitDialog()**. Поскольку **CDialog::OnInitDialog()** — виртуальная, создавать для нее обработчик сообщений не потребуется — можно просто перекрыть ее.

Зачем может потребоваться перекрыть **OnInitDialog()**? Например, может возникнуть необходимость вручную инициализировать элементы управления — скажем, заполнить окно списка. Впрочем, **OnInitDialog()** — не место для инициализации значений переменных, ассоциированных с элементами управления. Позже будет показано, как их инициализировать.

Кроме того, можно перекрыть **OnInitDailog()**, чтобы установить начальный фокус клавиатуры на элементе, отличном от первого в порядке обхода элементов управления, основанный на каком-то значении времени выполнения. В этом случае вызывается **SetFocus()** для установки фокуса клавиатуры на выбранный элемент управления, и возвращается значение **FALSE** из **OnInitDialog()**. В противном случае **OnInitDialog()** должна вернуть **TRUE**.

В случае перекрывания **OnInitDialog()** вызовите в последней строке функции метод **CDialog::OnInitDialog()**, иначе он будет вызван автоматически, как показано в шаге 5 на рис. 18.14. **CDialog::OnInitDialog()** сперва вызывает функцию **CWnd::UpdateData()**, передавая **FALSE** в качестве аргумента (шаг 6 на рис. 18.14). Функция **UpdateData()** создает указатель на объект **CDataExchange**, а затем возвращается к функции **DoDataExchange()** класса **CPickDateDlg** (шаг 7 на рис. 18.14). В заключение **CDialog::OnInitDialog()** выведет окно диалога, чтобы пользователь мог в нем работать.

Как работает механизм передачи данных диалога

На первый взгляд серия функций, порождаемая вызовом **dlg.DoModal()**, может показаться запутанной. Впрочем, если вы поймете цель каждого из вызовов, эта операция приобретет намного больше смысла. Рассмотрим в двух словах предназначение функций **UpdateData()** и **DoDataExchange()**:

- **UpdateData()** передает информацию между элементами управления и переменными класса диалогового окна. Если **UpdateData()** вызывается с параметром **FALSE**, элементы управления инициализируются значениями переменных. Если **UpdateData()** вызывается с параметром **TRUE**, передача данных производится в другом направлении — из элементов управления Windows в переменные.
- При передаче данных между элементами управления и переменными **UpdateData()** использует *карту данных* для связывания каждого элемента управления с конкретной переменной. Карта данных также указывает тип преобразования, который должен производиться при передаче информации. Эта карта данных содержится в функции **DoDataExchange()**.

Понимание работы кода окажет помощь при разрешении проблем. В листинге 18.2 приведена функция **CPickDateDlg::DoDataExchange()**.

Листинг 18.2. Функция CPickDateDlg::DoDataExchange().

```
void CPickDateDlg::DoDataExchange(CDataExchange* pDX)
{
      CDialog::DoDataExchange(pDX);
      //{{AFX_DATA_MAP(CPickDateDlg)
      DDX_DateTimeCtrl(pDX, IDC_DATETIMEPICKER1, m_DateValue);
      //}}AFX_DATA_MAP
}
```

Функция **DoDataExchange()** является двунаправленной. Функция принимает указатель на объект **CDataExchange()**, созданный функцией **UpdateData()**, который указывает направление передачи информации. **DoDataExchange()** передает этот указатель (**pDX** в листинге 18.2) истинной функции обмена данными.

Функции обмена данными, называемые DDX-функциями, принимают три параметра:

- Указатель на объект **CDataExchange**, определяющий направление передачи данных.
- Идентификатор ресурса элемента управления.
- Переменная, выступающая в качестве источника или цели передаваемого значения.

Для каждого элемента управления и каждого типа данных его значения существуют специфические DDX-функции.

Для некоторых элементов управления можно задать *код проверки корректности значения* (validation code). Например, если диалоговое окно содержит элемент редактирования, ClassWizard предоставляет возможность ассоциировать с ним несколько типов переменных. Если элемент управления ассоциируется с числовым значением, ClassWizard позволяет определять минимальное и максимальное значения этого элемента. Он записывает функцию проверки, называемую DDV-функцией, и помещает ее вместо с DDX-функцией в **DoDataExchange()**. DDV-функция предотвращает закрытие диалогового окна в то время, когда ассоциированный с нею элемент управления содержит некорректное значение.

Закрытие диалогового окна

После вызова **CDialog::OnInitDialog()** можно передвигаться по элементам управления диалогового окна. Если какой-либо из элементов управления имеет обработчик сообщения, он будет выполняться при использовании этого элемента. В связи с тем что это модальное окно диалога, элемент управления не может вернуть значения вызывающей программе до тех пор, пока окно не закроется.

Для закрытия модального диалогового окна вызывается функция **EndDialog()** с одним из значений: **IDOK** или **IDCANCEL**. **DoModal()** возвращает это значение в вызывающую программу. Функция, выполняющая функцию **DoModal()**, проверяет это значение и действует соответствующим образом.

Вернувшись к рис. 18.14, можно увидеть, что **OnCancel()** и **OnOK()** являются виртуальными функциями **CDialog**. Они не перекрываются в классе **CPickDateDlg**. При желании имеется возможность реализовать более тщательную проверку правильности данных, нежели та, что обеспечивается DDV-функциями ClassWizard. Если вы решите перекрыть эти функции, обратитесь к **CDialog::OnOK()** или

CDialog::OnCancel() вместо того, чтобы напрямую вызывать **EndDialog()** — это особенно важно в случае виртуальной функции **OnOK()**.

Когда работа с диалоговым окном завершается щелчком на OK, функция **CDialog::OnOK()** вновь вызовет **UpdateData()**, передав ей на этот раз **TRUE**, тем самым указывая, что данные будут передаваться из элементов управления в переменные. Как и ранее, **UpdateData()** вызывает функцию **DoDataExchange()**. В заключение **CDialog::OnOK()** обращается к **EndDialog()**, передавая в нее **IDOK**, чтобы **DoModal()** знала, чем все завершилось. Еще раз взгляните на шаги 11–14 на рис. 18.14.

Обработка щелчка на OK

Когда **DoModal()** завершает выполнение, функция **EndDialog()** уже скрыла диалоговое окно. Его больше нет на экране, но объект окна диалога все еще остается в памяти. Он не будет уничтожен до тех пор, пока переменная диалогового окна не покинет пределы области видимости.

Если **DoModal()** возвращает **IDOK**, следует немедленно передать информацию переменных диалога в класс, который вызвал это окно диалога. Несмотря на то что функции обмена данными диалога обрабатывают передачу между элементами управления окна диалога и его переменными, они не отвечают за обмен информацией с внешним миром. Как только объект окна диалога выходит за пределы области видимости, эта информация теряется.

Если функция **CWZView::OnEditPasteDate()** получает **IDOK** от сконструированного ею объекта диалога, она принимается за выполнение следующих трех задач:

- Получает значение **COleTimeDate**, содержащееся в переменной диалога **m_DateValue**, и сохраняет его в переменной **m_PasteData** класса **CWZView**.
- Создает временную переменную **CString** с именем **sDate** с помощью функции **Format()** класса **COleTimeDate**. **Format()** работает подобно **printf()***, но имеет несколько опций форматирования, специально предназначенных для форматирования дат. В **OnEditPasteDate()** используются **%B** для вывода полного названия месяца, **%d** для вывода дня в виде десятичного числа, и **%Y** для вывода четырехзначного десятичного значения года. (Другие спецификаторы форматирования описаны в оперативной справке Visual C++ по функции **strftime()**.)
- Вставляет отформатированную дату в текущий вид, вызвав функцию **CRichEditView::GetRichEditCtrl()** для получения лежащего в основе элемента форматируемого текста, а затем обращается к функции **RichEditCtrl::ReplaceSel()** для вставки вновь отформатированной даты.

Немодальные диалоговые окна

Если вы пишете в WordZilla письма своим друзьям, то, вероятно, использовать возможность вставки даты будете нечасто. Для редко повторяемых операций модальное диалоговое окно будет хорошим решением.

Однако предположим, что вместо писем друзьям, WordZilla применяется для создания еженедельных отчетов для юго-западной ассоциации игры в кегли, и каждый отчет содержит 20 или 30 дат. В этом случае модальное диалоговое окно кажется менее подходящим. В данном случае потребуется окно диалога, остающееся открытым до тех пор, пока его не закрыть принудительно — *немодальное* диалоговое окно.

** Более близким аналогом Format() является wsprintf(), а не printf()) — прим. перев.*

Как работают немодальные диалоговые окна

Немодальные диалоговые окна используют тот же шаблон диалога, что и модальные. MFC создает модальные окна диалога в стеке, тогда как немодальные создаются в куче. Для модального окна диалога MFC автоматически вызывает деструктор, а для удаления немодального необходимо вызывать **delete**.

Модальное окно диалога выводится с помощью **DoModal()**, тогда как немодальное — с помощью **Create()** и **ShowWindow()**. Для закрытия модального окна диалога так или иначе вызывается **EndDialog()**. Немодальное диалоговое окно можно скрыть, вызвав **ShowWindow(SW_HIDE)** либо уничтожить, вызвав **DestroyWindow()**.

В немодальном диалоговом окне следует перекрыть виртуальные функции **OnOK()** и **OnCancel()**, чтобы предотвратить вызов стандартных версий этих функций класса **CDialog**. Поскольку метод **OnOK()** в этом случае не вызывается, для передачи данных необходимо самостоятельно вызвать **UpdateData(TRUE)**.

Давайте посмотрим, как это работает, переключившись к немодальной версии диалогового окна Select a Date.

Немодальное диалоговое окно для WordZilla

Прежде чем перейти к конкретным действиям, давайте решим, чего мы добиваемся. Когда диалоговое окно было модальным, оно создалось в функции **OnEditPasteDate()** класса представления. Все работало прекрасно, поскольку модальное диалоговое окно создавалось и уничтожалось в одной функции.

Эта схема не годится для немодального диалогового окна, поскольку ничто не может запретить пользователю переключиться на другой документ или закрыть текущий при открытом окне диалога. Взамен диалоговое окно будет создаваться в конструкторе **CMainWnd**, а разрушаться — в деструкторе этого класса. Таким образом вы будете уверены, что не произойдет утечки памяти.

По умолчанию диалоговые окна после создания невидимы. Следовательно, для вывода окна диалога в момент, когда пользователь выберет в меню опцию Edit | Paste Date потребуется вызвать **ShowWindow()**. В перекрытой функции **OnCancel()** окно диалога уничтожаться не будет, оно просто будет делаться невидимым при помощи вызова **ShowWindow(SW_HIDE)**. В заключение потребуется перенести код, вставляющий дату, из **CWZView::OnEditPasteDate()** в функцию **CPickDateDlg::OnOK()**.

Изменения в классе CWZView

В классе **CWZView** необходимо сделать только одно изменение — удалить функцию **OnEditPasteDate()**. Этот процесс состоит из двух шагов:

1. Откройте ClassWizard, выбрав из главного меню View | ClassWizard или нажав Ctrl+W и щелкнув на вкладке Message Maps. Выберите в выпадающем списке Class Name класс **CWZView**, а затем выберите обработчик **COMMAND** для **ID_EDIT_PASTE_DATE**. Когда ваше окно будет выглядеть, как показано на рис. 18.15, щелкните на Delete Function.
2. ClassWizard выдаст предупреждение, приведенное на рис. 18.16; в этом случае щелкните на OK. Откройте панель ClassView и разверните класс **CWZView**. Дважды щелкните на функции **OnEditPasteDate()** для открытия редактора исходного кода, а затем удалите или закомментируйте функцию **OnEditPasteDate()**.

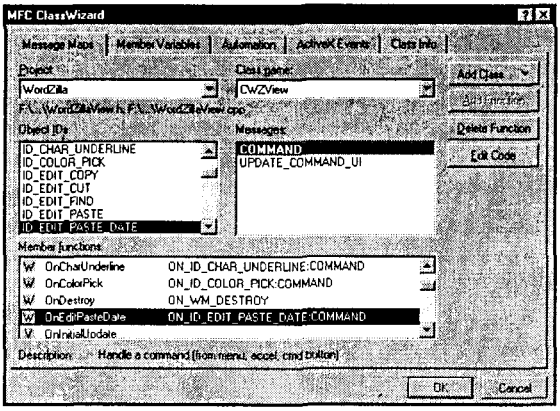


РИСУНОК 18.15. *Удаление функции CWZView:: OnEditPasteDate().*

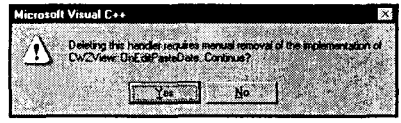


РИСУНОК 18.16. *Предупреждение ClassWizard об удалении класса вручную.*

Изменения в классе CMainFrame

В класс **CMainFrame** необходимо внести несколько больше изменений. Выполните следующие шаги:

1. Откройте проект WordZilla и отыщите в панели ClassView класс **CMainFrame**. Щелкните на нем правой кнопкой и выберите из контекстного меню Add Member Variable. Добавьте переменную **private CPickDateDlg *** с именем **m_pPickDateDlg**. Когда окно диалога приобретет вид, как на рис. 18.17, щелкните на OK.
2. Дважды щелкните на классе **CMainFrame** в панели ClassView, чтобы открыть заголовочный файл MainFrm.h. За объявлением класса **CMainFrame** поместите следующую строку:

   ```
   class CPickDateDlg;
   ```

   Такая строка называется *неполным объявлением класса* или *опережающей ссылкой* (forward reference). Она предоставляет компилятору достаточно информации для разбора объявления класса **CMainFrame**.
3. Разверните класс **CMainFrame** в панели ClassView и дважды щелкните на конструкторе **CMainFrame()**. Добавьте к конструктору и деструктору выделенный код из листинга 18.3.

Листинг 18.3. Конструктор и деструктор класса CMainFrame.

```
CMainFrame::CMainFrame()
{
    m_pPickDateDlg = new CPickDateDlg;
}

CMainFrame::~CMainFrame()
{
    m_pPickDateDlg->DestroyWindow();
    delete m_pPickDateDlg;
}
```

4. Найдите функцию **OnCreate()** и поместите следующую строку непосредственно перед оператором **return** в конце файла:

```
m_pPickDateDlg->Create(IDD_PICK_DATE, this);
```

(Если ошибочно поместить этот оператор в конструктор, диалоговое окно не будет оставаться сверху при потере фокуса.)

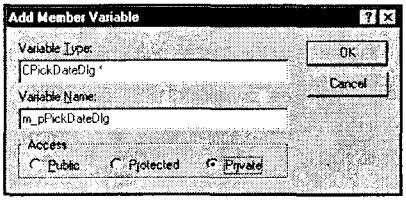


РИСУНОК 18.17. *Добавление в класс CMainFrame переменной m_pPickDateDlg.*

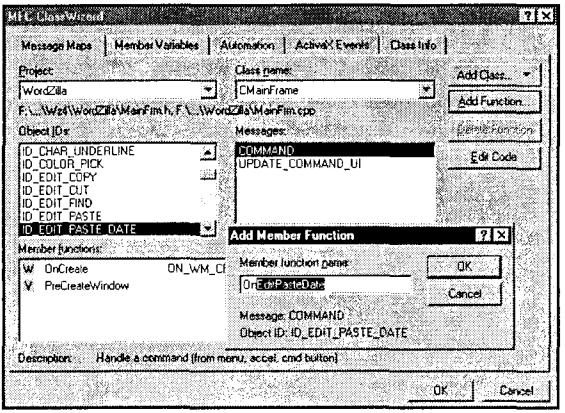


РИСУНОК 18.18. *Добавление обработчика меню OnEditPasteDate() в класс CMainFrame.*

5. Перейдите в начало файла CMainFrm.cpp и добавьте следующую строку после операторов **#include**:

```
#include "PickDateDlg.h"
```

6. Откройте ClassWizard и выберите класс **CMainFrame** из выпадающего списка Class Name. Отметьте в списке Object IDs **ID_EDIT_PASTE_DATE**, а в списке Messages — **COMMAND**. Щелкните на Add Function, а затем на OK после открытия диалогового окна Add Member Function (см. рис. 18.18). Добавьте в функцию **CMainFrame::OnEditPasteDate()** код, показанный в листинге 18.4.

Листинг 18.4. Обработчик меню CMainFrame::OnEditPasteDate().

```
void CMainFrame::OnEditPasteDate()
{
    if (m_pPickDateDlg->IsWindowVisible())
    {
        m_pPickDateDlg->SetFocus();
    }
    else
    {
        m_pPickDateDlg->ShowWindow(SW_SHOW);
    }
}
```

Изменения в классе CPickDateDlg

Для изменения класса **CPickDateDlg** используется Dialog Editor; кроме того, изменяется часть исходного кода. Выполните приведенные ниже шаги:

1. Откройте панель ResourceView в окне Workspace и разверните папку Dialogs. Дважды щелкните на ресурсе диалога **IDD_PICK_DATE**, чтобы открыть Dialog Editor. Откройте диалоговое окно Properties для кнопки Cancel и введите заголовок "&Close". Идентификатор оставьте установленным в **IDCANCEL**. Откройте диалоговое окно Properties для кнопки OK и измените ее заголовок на "&Paste". Идентификатор оставьте установленным в **IDOK**. Окончательный вид экрана показан на рис. 18.19.

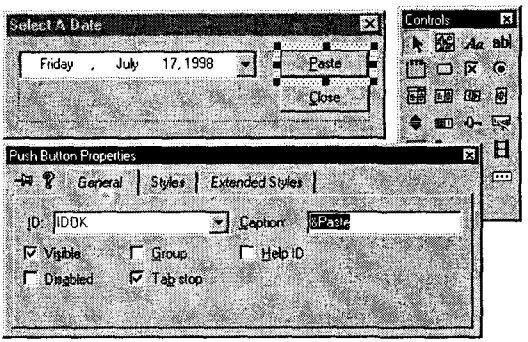


РИСУНОК 18.19. *Изменение заголовков кнопок диалогового окна Select a Date.*

2. Дважды щелкните на кнопке Close для создания нового обработчика. Здесь важно принять предлагаемое имя **OnCancel()** — это позволит перекрыть виртуальную функцию **OnCancel()**. Замените сгенерированный код кодом, приведенным в листинге 18.5.

Листинг 18.5. Функция CPickDateDlg::OnCancel().

```
void CPickDateDlg::OnCancel()
{
    ShowWindow(SW_HIDE);
}
```

3. Вернитесь к редактору диалога и дважды щелкните на кнопке Paste для создания нового обработчика. Как и в случае с кнопкой Close, примите предлагаемое имя **OnOK()** — не изменяйте его. Это позволит перекрыть функцию **CDialog::OnOK()**, что необходимо для немодального диалогового окна. Замените ее содержимое кодом из листинга 18.6. Как видите, этот код несколько сложнее предыдущего, однако изучив комментарии, его можно легко понять.

Листинг 18.6. Функция CPickDateDlg::OnOK().

```
void CPickDateDlg::OnOK()
{
  // 1. Сохранить данные элемента управления в переменной
  UpdateData(TRUE);

  // 2. Получить окно MainFrame
  CMainFrame * pMain = (CMainFrame*)AfxGetMainWnd();

  // 3. Проверить, есть ли активные CChildFrame
  CChildFrame * pChild = (CChildFrame*)pMain->GetActiveFrame();
  if (pChild == NULL)  // Активные дочерние элементы отсутствуют
      return;
```

```
    // 4. Получить текущее представление
    CWZView * pView = (CWZView*)pChild->GetActiveView();
    if (pView == NULL)    // Активные представления отсутствуют
        return;

    // 5. Отформатировать объект даты
    CString sDate = m_DateValue.Format("%B %d, %Y");

    // 6. Получить элемент форматируемого текста и вставить дату
    pView->GetRichEditCtrl().ReplaceSel(sDate);

    // 7. Установить фокус на элемент форматируемого текста
    pView->SetFocus();
}
```

4. Отыщите конструктор класса **CPickDateDlg** и закомментируйте строку, в которой вызывается конструктор **CDialog**. Задача, выполняемая ранее конструктором **CDialog**, теперь реализуется в его методе **Create()**. Окончательный конструктор должен выглядеть, как показано в листинге 18.7, причем изменения выделены.

Листинг 18.7. Изменения в конструкторе CPickDateDlg.

```
CPickDateDlg::CPickDateDlg(CWnd* pParent /*=NULL*/)
//    : CDialog(CPickDateDlg::IDD, pParent)
{
      //{{AFX_DATA_INIT(CPickDateDlg)
      m_DateValue = COleDateTime::GetCurrentTime();
      //}}AFX_DATA_INIT
}
```

5. За счет добавления функции **OnOK()** класс **PickDateDlg** теперь ссылается на несколько классов, о которых ранее ему было неизвестно. Сообщите ему о них, добавив выделенные операторы **#include** из листинга 18.8 в начало файла PickDateDlg.cpp.

Листинг 18.8. Добавление заголовочных файлов в PickDateDlg.cpp.

```
#include "stdafx.h"
#include "WordZilla.h"
#include "PickDateDlg.h"
#include "MainFrm.h"
#include "ChildFrm.h"
#include "CntrItem.h"
#include "WordZillaDoc.h"
#include "WordZillaView.h"
```

Вот и все, что требовалось. Откомпилируйте и запустите WordZilla после чего откройте диалоговое окно Select a Date. Теперь при нажатии на Paste диалоговое окно не закрывается — оно остается открытым, располагаясь поверх окна документа. Можно переключаться между несколькими документами, а кнопка Paste будет вставлять выбранную вами дату в текущий документ.

ActiveX-версия DatePicker

Теперь, когда вы узнаете о том, как создавать модальные и немодальные диалоговые окна, давайте рассмотрим, как добавить новый тип элемента управления.

Элементы управления ActiveX схожи с обычными элементами управления Windows по нескольким характеристикам. Код элементов управления Windows находится не в программе, но в отдельной библиотеке динамической компоновки (DLL). Например, общие элементы управления Windows 95 содержатся в файле Comctl32.dll, расположенном в каталоге Windows\System (для Windows 95 или Windows 98) или в каталоге WinNT\System32 (для Windows NT).

Элементы управления ActiveX, подобно обычным элементам управления Windows, также являются динамически компонуемыми. Однако библиотеки динамической компоновки ActiveX обычно используют расширение .ocx. Как правило, каждый OCX-файл содержит только один элемент.

По внутренней структуре элементы управления ActiveX довольно сильно отличаются от обычных элементов управления Windows. С другой стороны, при их использовании в MFC-программах, с ними можно обращаться почти как со встроенными элементами управления — но только "почти." Давайте заменим встроенный элемент управления DateTiemPicker его ActiveX-версией, чтобы посмотреть на существующие отличия.

Использование элементов ActiveX в MFC-программе — это 3-шаговый процесс:

- Во-первых, следует добавить элемент управления ActiveX в панель инструментов Controls редактора диалога. Тогда Visual C++ создаст прокси-класс для нового элемента управления ActiveX и добавит его в ваш проект.
- Во-вторых, при помощи Dialog Editor необходимо добавить экземпляры элемента управления ActiveX в диалоговое окно.
- В-третьих, потребуется вызвать ClassWizard для генерации кода реакции на *события* элемента управления. Кроме того, при помощи ClassWizard следует ассоциировать элемент управления с прокси-классом, созданным при первом добавлении элемента управления в проект.

Давайте пройдемся по этим шагам, добавляя ActiveX-версию элемента выбора даты и времени.

Добавление элементов управления ActiveX

Все начинается с помещения элемента управления ActiveX в свой проект и создания прокси-класса. Затем элемент управления появляется в панели инструментов Controls редактора диалога, подобно встроенному элементу управления.

ПРИМЕЧАНИЕ

Элементы управления ActiveX размещаются в палитре элементов управления на попроектной, а не постоянной основе.

Вот шаги для добавления элемента управления Microsoft Date And Time Picker в проект WordZilla:

1. Откройте проект WordZilla. Выберите Project | Add To Project | Components And Controls, чтобы открыть диалоговое окно галереи компонентов и элементов управления ActiveX. Откройте папку Registered ActiveX Controls, затем выберите элемент Microsoft Date And Time Picker Control, Version 6.0. Когда ваше окно будет выглядеть, как показано на рис. 18.20, щелкните на Insert.

СОВЕТ

Справка не всегда полезна

Если элемент управления ActiveX имеет связанный с ним справочный файл, MFC активизирует кнопку More Info, что видно на рис. 18.20. Однако документация в целом подразумевает, что компонент используется в VB (вот почему так важно понимать различия между использованием элементов управления ActiveX в Visual C++ и в VB). Заметьте, что этот элемент управления находится в файле MSCOMCT2.OCX. Если программа использует этот элемент управления, файл MSCOMCT2.OCX должен поставляться вместе с программой, иначе она работать не будет.

2. В своей программе на Visual C++ можно напрямую работать с элементами управления ActiveX, но *определенно* это делаться не будет. (Вскоре будет показано почему.) Вместо этого для каждого добавляемого элемента управления ActiveX Visual C++ генерирует классы оболочки, или прокси-классы, позволяющие рассматривать элемент управления как объект C++. На рис. 18.21 показано диалоговое окно Confirm Classes, позволяющее определить генерируемые классы. Поскольку добавляется только один элемент управления, просто щелкните на OK.

Если в проекте задействовано несколько элементов управления ActiveX, все они могут использовать класс **COleFont**, и его можно включить только один раз. В таком случае необходимо снять отметку с флажка класса **COleFont**. Также можно изменить имя основного класса, генерируемого Visual C++ (хотя по вполне понятным причинам нельзя менять имена вспомогательных классов наподобие **CPicture** и **COleFont**). Если, например, в программе уже есть класс **CDTPicker**, то предлагаемое имя класса можно изменить.

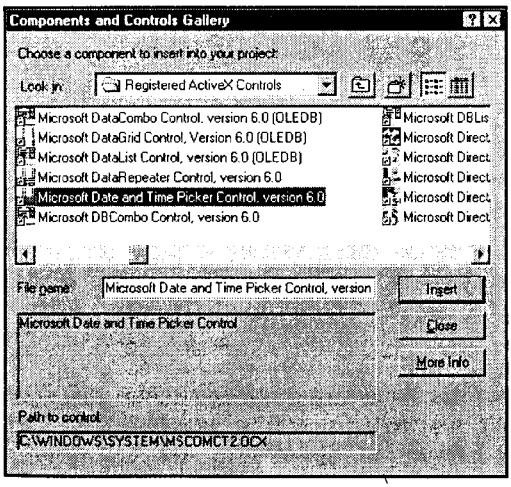


РИСУНОК 18.20. *Добавление в проект элемента управления ActiveX.*

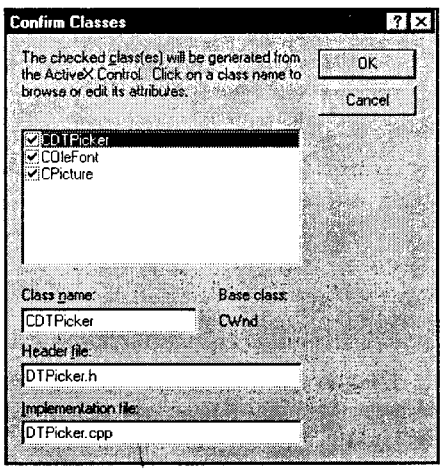


РИСУНОК 18.21. *Диалоговое окно Confirm Classes при добавлении элемента управления ActiveX.*

Это первый шаг по внедрению элемента управления ActiveX в MFC. Закройте диалоговое окно Components And Controls Gallery и приготовьтесь к следующему шагу: использованию элементов управления ActiveX в Dialog Editor.

Элементы управления ActiveX в редакторе диалоговых окон

По завершению работы с Gallery воспользуйтесь панелью ResourceView в окне Workspace для открытия диалогового окна **IDD_PICK_DATE** в Dialog Editor. Отметьте элемент управления Date Time Picker (**IDC_DATETIMEPICKER1**) и нажмите клавишу Delete для его удаления.

Теперь вы готовы заменить элемент управления Date And Time Picker его ActiveX-версией. Для этого выполните следующие шаги:

1. Отыщите в панели инструментов новый элемент управления Date And Time Picker, показанный на рис. 18.22.
2. Перетащите элемент управления из панели инструментов Controls в диалоговое окно Select a Date. На рис. 18.23 показано, что при перетаскивании компонента к курсору мыши добавляется "X". Это изменение указывает на то, что добавляется элемент управления ActiveX, а не обычный элемент управления Windows.

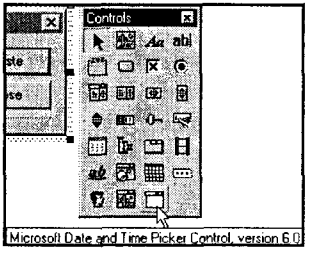


РИСУНОК 18.22. *Элемент управления Date And Time Picker в панели инструментов Controls.*

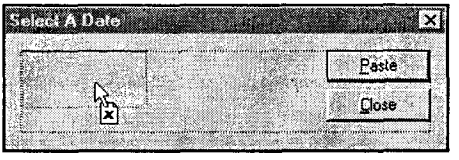


РИСУНОК 18.23. *Перетаскивание элемента управления ActiveX.*

3. Разместите и установите размеры компонента, как это делалось со встроенным элементом управления Data And Time Picker. Затем щелкните на нем правой кнопкой для открытия контекстного меню компонента. Следует отметить, что помимо обычной опции меню Properties появляется отдельная опция Properties DTPicker Object. Выберите одну из них, чтобы открыть окно свойств элемента управления ActiveX.

Свойства Date And Time Picker

При открытии диалогового окна Date And Time Picker Properties вы сразу заметите, что здесь намного больше опций, чем во встроенном элементе управления Date And Time Picker. Поскольку элементы управления ActiveX обычно используются в средах "drag-and-drop" типа VB, они обычно обладают множеством опций. Давайте исследуем некоторые из свойств, которые можно изменить.

1. Перейдите на вкладку Control, затем при помощи выпадающего списка измените свойство Format на 3-dtpCustom. Во встроенном элементе управле-

ния Date And Time Picker можно выбирать между длинным и коротким форматом, однако нельзя точно задать требуемый формат. Зато это допускается в аналогичном элементе управления ActiveX. После изменения свойства Format введите в поле редактирования Custom Format строку "MMMM d, yyy". Этот пользовательский формат подразумевает, что элемент управления будет использовать тот же формат для вставки в элемент форматируемого текста. (Другие возможные форматы описаны в документации по элементу управления.) Экран должен выглядеть приблизительно так, как показано на рис. 18.24.

СОВЕТ

Как насчет проблемы 2000-го года?

Элемент управления ActiveX не кажется совместимым с 2000 годом. Если вы попытаетесь изменить свойство MaxDate, элемент управления использует свой собственный календарь для установки даты. Поскольку максимальная дата установлена в 31 декабря 1999 года, установить дату после текущего тысячелетия нельзя.

Не волнуйтесь, доступ ко всем свойствам всех страниц можно получить в окне свойств All, где просто вводится новое значение. После изменения свойства MaxDate на странице All появляется возможность ввести любую дату вплоть до 31 декабря 9999 года. Если же вам нужен продукт, совместимый с 10 000 годом — поищите еще что-нибудь.

2. Страница Color (рис. 18.25) позволяет изменить цвета, используемые в разных частях выводимого календаря. Однако, изменить цвет выпадающего списка в элементе управления ActiveX нельзя. Выбор можно производить между системными цветами Windows и стандартными цветовыми наборами, а также определять свои собственные цвета. Измените цвета нескольких элементов календаря: попробуйте придумать цветовую гамму, которая легка для чтения, но в то же время заставляет пользователя обратить внимание на календарь.

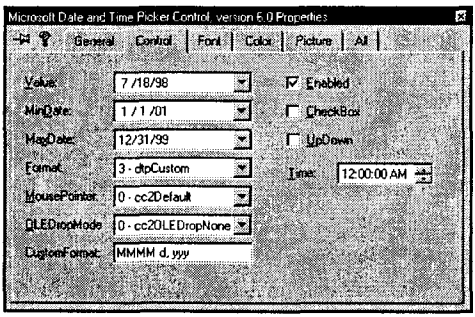


РИСУНОК 18.24. *Изменение свойств элемента управления Date And Time Picker.*

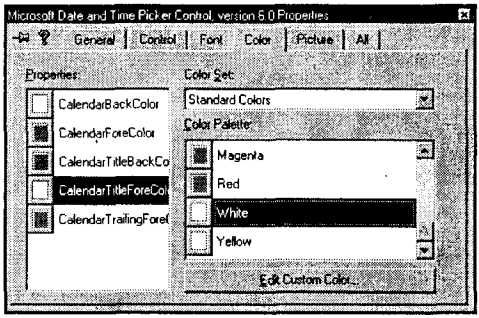


РИСУНОК 18.25. *Установка свойств элемента управления Date And Time Picker.*

3. В заключение измените свойство **Font**, используемое во всем элементе управления. Как и для страницы свойств цвета, попытайтесь поговорить с кем-нибудь осведомленным в дизайне, прежде чем изменять это свойство. С

другой стороны, маленький 12-точечный шрифт Blippo никогда никому не помешает (см. рис. 18.26).

Элементы управления ActiveX, код и ClassWizard

Прежде чем подключать новый элемент управления Date And Time Picker, необходимо отключить старый. Вы удалили элемент управления в редакторе диалога, но класс **CPickDateDlg** все еще имеет переменную **m_DateValue**, связанную со старым элементом управления.

Класс **CPickDateDlg** все еще нуждается в хранении данных. Поэтому после удаления переменной с помощью ClassWizard, ее необходимо вернуть как несвязанную переменную.

В конечном итоге, вместо автоматического обмена данными **DDX** (который не работает с элементами управления ActiveX), потребуется перехватить событие ActiveX, а затем вручную выполнить необходимый перенос данных.

Вы готовы? Последуйте приведенным ниже шагам:

1. Откройте ClassWizard и выберите класс **CPickDateDlg** из выпадающего списка Class Name. Перейдите на вкладку Member Variables и удалите переменную **m_DateValue**, ассоциированную с идентификатором **IDC_DATETIMEPICKER1**. Затем отметьте идентификатор нового элемента управления **IDC_DTPICKER1** и добавьте новую переменную **m_DTPickerCtrl**, как показано на рис. 18.27. Обратите внимание, что создаваемая переменная является экземпляром ранее созданного прокси-класса. Объект прокси будет использоваться для вызова функций, которые он, в свою очередь, будет передавать самому объекту ActiveX.

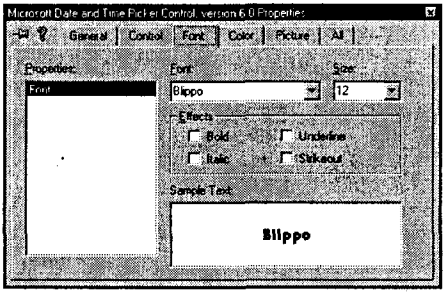


РИСУНОК 18.26. *Изменение свойства Font элемента управления Date And Time Picker.*

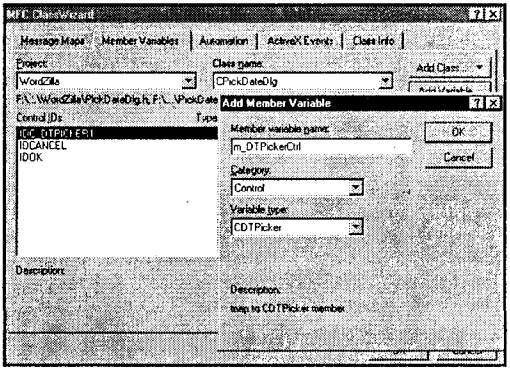


РИСУНОК 18.27. *Добавление переменной m_DTPicker в класс CPickDateDlg.*

2. Перейдите на вкладку Message Maps, отыщите **IDC_DTPICKER1** и выберите Change в списке Messages. Каждый из элементов этого списка является событием, генерируемым элементом управления ActiveX. Элемент управления Date And Time Picker посылает сообщение Change всякий раз при изменении даты в элементе управления. Щелкните на Add и примите предлагаемое имя функции. Щелкните на Edit Code и замените сгенерированный код приведенным в листинге 18.9.

Листинг 18.9. Функция CPickDateDlg::OnChangeDtpicker1().

```
void CPickDateDlg::OnChangeDtpicker1()
{
      m_DateValue = m_DTPickerCtrl.GetValue();
}
```

При каждом изменении даты в элементе управления ActiveX Date And Time Picker, последний генерирует событие, приводящее к вызову вашей функции. Внутри функции с помощью объекта прокси вы вызываете функцию **GetValue()**, и опять-таки сохраняете значение в переменной **m_DateValue**.

3. Отыщите класс **CPickDateDlg** в панели ClassView окна Workspace. Щелкните на нем правой кнопкой мыши для вызова контекстного меню, а затем добавьте новую приватную переменную **COleDateTime m_DateValue**, как показано на рис. 18.28. Она заменит переменную, которая была связана с оригинальным элементом управления Date And Time Picker.

По завершении откомпилируйте и запустите программу. Как можно увидеть на рис. 18.29, новый элемент управления работает практически так же, как и в предыдущей версии — только более красочно!

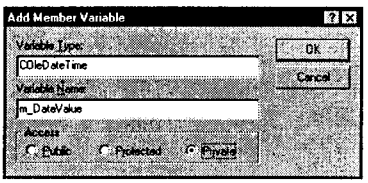


РИСУНОК 18.28. *Добавление переменной m_DateValue.*

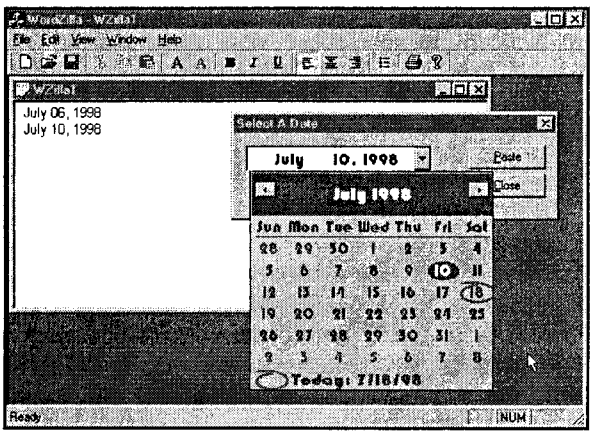


РИСУНОК 18.29. *WordZilla с элементом управления ActiveX Date And Time Picker.*

Свойства, события и методы

Из документации по элементу управления ActiveX Date And Time Picker можно сделать вывод, что с точки зрения Visual Basic каждый элемент управления может иметь свойства, события и методы.

В контексте VB свойства подобны элементам данных (переменным), и доступ к ним производится напрямую. Скажем, для свойства **Value** можно было бы записать такой код:

```
m_DateValue = mDTPickerCtrl.Value;
```

Если бы **Value** была общедоступной переменной класса **CDTPicker**, подобный код оказался бы корректным. Однако в Visual C++ свойства не имеют прямой доступ к общедоступным переменным. Взамен каждое свойство имеет пару фун-

кций (для случая **Value** они называются **GetValue()** и **SetValue()**), которые позволяют получать или устанавливать значение свойства.

В элементе управления ActiveX метод подобен процедуре — это что-то, что можно попросить сделать элемент управления. Элемент управления Date And Time Picker не имеет методов. Если бы они у него были, к ним можно было бы обращаться через прокси-класс точно так же, как и получать доступ к свойствам объекта. С точки зрения Visual C++ как методы, так и свойства являются функциями, вызываемыми с целью получения доступа к элементу управления ActiveX.

События, наоборот, позволяют элементу управления ActiveX общаться с вашей программой. Можно считать их подобными сообщениям **WM_COMMAND** или **WM_NOTIFY**. События отображаются на функции вашего класса при помощи механизма, очень похожего на карты сообщений. Механизм ActiveX вместо **MESSAGE_MAP** использует карту сообщений **EVENTSINK_MAP**, но поскольку практически всегда поддерживать карту будет ClassWizard, это различие несущественно. Ради удовлетворения вашего любопытства приведем карту **EVENTSINK_MAP**, направляющую событие Change в функцию **OnChangeDtpicker1()**:

```
BEGIN_EVENTSINK_MAP(CPickDateDlg, CDialog)
      //{{AFX_EVENTSINK_MAP(CPickDateDlg)
      ON_EVENT(CPickDateDlg, IDC_DTPICKER1, 2,
         OnChangeDtpicker1, VTS_NONE)
      //}}AFX_EVENTSINK_MAP
END_EVENTSINK_MAP()
```

И вновь ActiveX

В этой главе было показано, как создавать диалоговые окна, которые могут содержать обычные элементы управления Windows, а также элементы управления ActiveX. В следующей главе рассматривается использование класса представлений, основанного на шаблоне диалога FormView в качестве основы для сборки приложений баз данных. Ваш опыт работы с ActiveX окажется неоценимым, когда вы перейдете к *элементам управления, ориентированными на данные (data-aware controls)*.

Однако прежде чем двигаться дальше, давайте применим новые навыки работы с элементами управления ActiveX для реализации последней версии WordZilla. Мы уверены, что при работе над этим проектом в глаза бросался один недостаток. Наступило время исправить его. Да, сейчас будет создано *музыкальное* диалоговое окно About с использованием мультимедийного элемента управления ActiveX.

Мы не будем вести вас через каждый шаг, а просто осветим главные детали. Вот, что потребуется сделать:

1. В диалоговом окне Components And Controls Gallery выбрать элемент управления Microsoft Multimedia Control (MCI) и добавить его в проект. (Необходимый класс **CPicture** уже имеется, так что нет необходимости создавать его еще раз.)
2. Откройте диалоговое окно About, затем перетащите на него элемент управления MCI и обычную кнопку (см. рис. 18.30). Определите идентификатор кнопки как **IDC_NEW_FILE**, а для элемента управления MCI оставьте значение, принятое по умолчанию. Заголовок кнопки установите в "New File".

3. С помощью ClassWizard создайте новую переменную, связанную с элементом управления MCI, как показано на рис. 18.31. Назовите переменную **m_MCICtrl**.

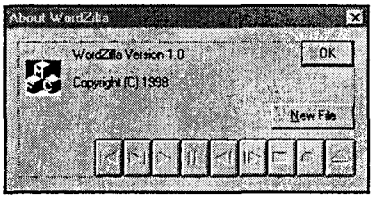


РИСУНОК 18.30. *Компоновка диалогового окна About в WordZilla.*

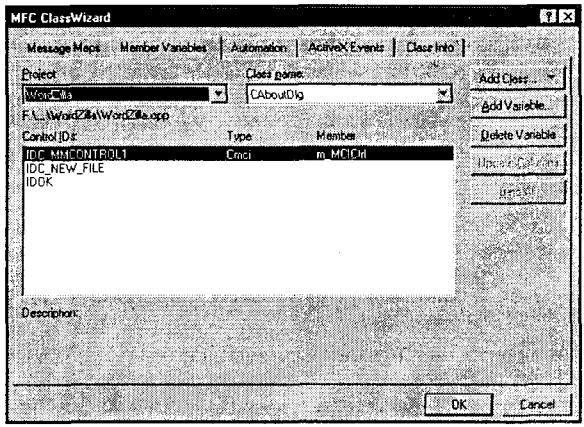


РИСУНОК 18.31. *Добавление переменной для элемента управления MCI.*

4. Вручную добавьте в класс **CAboutDlg** переменную **CString m_DefaultSong**. Инициализируйте ее в конструкторе значением "Yeeehaaa.mid".
5. Дважды щелкните на кнопках New File и OK для добавления новых обработчиков событий. Замените сгенерированный код показанным в листинге 18.10.

Листинг 18.10. Функции OnOK() и OnNewFile().

```
void CAboutDlg::OnOK()
{
      // ЧТО СДЕЛАТЬ: Добавить код дополнительной проверки
      // допустимости
      m_MCICtrl.SetCommand("Close");
      CDialog::OnOK();
}

void CAboutDlg::OnNewFile()
{
    // ЧТО СДЕЛАТЬ: Поместить здесь код обработчика уведомлений
    // элемента управления
      CFileDialog dlg(TRUE, "*.mid", m_DefaultSong);
      if (dlg.DoModal() == IDOK)
      {
            m_DefaultSong = dlg.GetPathName();
            m_MCICtrl.SetCommand("Close");
            m_MCICtrl.SetFileName(m_DefaultSong);
            m_MCICtrl.SetCommand("Open");
            m_MCICtrl.SetCommand("Play");
      }
}
```

6. С помощью ClassWizard создайте обработчик сообщения **WM_INITDIALOG**. Добавьте в функцию **OnInitDilaog()** код из листинга 18.11.

Листинг 18.11. Функция CAboutDlg::OnInitDialog().

```
BOOL CAboutDlg::OnInitDialog()
{
      CDialog::OnInitDialog();

      // ЧТО СДЕЛАТЬ: Добавить код дополнительной инициализации
      m_MCICtrl.SetFileName(m_DefaultSong);
      m_MCICtrl.SetCommand("Open");
      m_MCICtrl.SetCommand("Play");
      return TRUE;          // Вернуть TRUE, если фокус не установлен
                            // на какой-то элемент управления
                            // ИСКЛЮЧЕНИЕ: Страницы свойств OCX будут
                            // возвращать FALSE
}
```

Вот и все! Скопируйте MIDI-файлы, находящиеся на сопровождающем CD-ROM в одном каталоге вместе с выполняемым файлом WordZilla, и вам начнет завидовать весь цивилизованный мир. (Если программа запускается из IDE, то MIDI-файлы должны находиться в каталоге с исходными файлами проекта, в не с выполняемой программой.)

Путешествие к источнику данных

Данные — это необходимое сырье информационных систем. Большинство программного обеспечения компьютеров создано для транспортировки данных и преобразования их в информацию. В связи с такой важностью данных большинство современных приложений хранят их в базах данных, что делает возможным их безопасное совместное использование. Однако защита и удобство иногда ведут себя несколько странно: доступ к данным, защищенным в базе данных, может оказаться слишком громоздким. В следующей главе будет показано, как использовать несколько возможностей MFC, позволяющих заполнять источники данных и выкачивать из них данные при необходимости. Обещаем, что вы получите массу удовольствия.

Глава 19

Программное обеспечение в работе: создание запроса в базу данных и обновление приложений

SQL, ODBC, DAO, ISAM, RDO, ADO, OLE DB.... Если вы поклонник телесериала "Секретные материалы", аббревиатуры баз данных могут напомнить мрачные правительственные агентства, показанные в фильме, где за каждым скрывается загадочный набор материалов, равно как и часть истины. Реальность намного более прозаична; правду о технологии баз данных нельзя найти в телевизоре, хотя стоит попробовать поискать ее в холодильнике.

"Таинственная запеканка" — это самый быстрый способ понять последние технологии баз данных. Сделать таинственную запеканку можно, не глядя смешав содержимое всех отсеков вашего холодильника и выпекая полчаса при температуре 350°. Иногда может получиться прекрасно, но иногда все закончится кисло-сладкими сырными блинчиками с сыроватой свининой. Подобным образом каждая новая технология баз данных исправляет некоторые недостатки предыдущей, но часто новая технология сохраняет привкус своих предшественников.

Visual C++ — чрезвычайно гибкий инструмент; если приложение вообще может быть написано, значит, оно может быть написано на Visual C++. То же самое сказать, например, о Visual Basic или о Microsoft Access, нельзя. Но вы не должны писать приложение с использованием определенного продукта только из-за того, что он это позволяет. Если вы, например, хотите разрабатывать коммерческие приложения баз данных, Visual C++, вероятно, не подойдет для такой работы — вместо этого вы должны рассмотреть варианты использования Visual FoxPro, Powersoft PowerBuilder или Borland Visual dBASE. Средства поддержки баз данных позволяют получать доступ к базам данных из программ C++, но это еще не делает Visual C++ средой разработки баз данных.

Несмотря на то что у вас установлена новейшая версия Visual C++, обновлены не все составляющие ее средства и технологии. Приведем пример. При использовании ODBC (Open Database Connectivity — Открытое соединение с базами данных) для подключения к источнику данных необходимо задать местонахождение источников данных и драйвер, который хотите использовать. Когда ODBC только появился, это производилось при помощи средства ODBC Data Source Administrator, в котором создавалось имя источника данных (Data Source Name, DSN) на пользовательской машине.

С течением времени и появлением новых версий ODBC изъяны этого метода ставали все более очевидными. В ответ Microsoft разработала файловые DSN, позволяющие совместно использовать DSN в сети, и таким образом исключить необходимость установки DSN на каждый компьютер. Если для подключения к базе данных применяется средство Microsoft типа Visual InterDev, оно по умолчанию использует файловые DSN. Однако при использовании MFC для подключения к базе данных использовать файловый DSN возможности нет. Дело не в том, что поддержка ODBC в MFC устарела — просто она не реализована должным образом.

В связи с таким большим количеством вариантов (и мест, где можно потеряться), давайте потратим несколько минут на составление плана. Мы умышленно упростили большинство примеров программ этой главы, чтобы вы явно видели то, что происходит. Вместо подробного исследования одного из средств доступа к базам данных, такого как ODBC или Microsoft Access, мы хотим, чтобы вы увидели всю территорию целиком. Поэтому мы рассмотрим ODBC, Data Access Object (DAO) и OLE DB. В следующей главе мы покажем, как с помощью элементов управления ActiveX создавать более изощренные приложения баз данных, причем с намного меньшими затратами по времени.

В этой ориентированной на проекты главе мы сконцентрируемся на *практике*. Мы покажем, как писать приложения баз данных типа "запросить-и-обновить" ("query-and-update"), но не будем сильно распространяться о языке структурированных запросов (SQL) или о дизайне баз данных. С ними вы познакомитесь в следующей главе.

Сперва получите некоторые данные

Visual C++ поставляется со множеством примеров программ. Две из них, DAOView и DAOTable, позволяют создавать и просматривать структуру баз данных Microsoft Access без использования самого Access. Давайте воспользуемся этими программами для изучения баз данных. Если вы предпочитаете использовать Access — прекрасно, только вам придется придумать свой личный способ решения задачи.

Для программ этой главы будет использоваться пример базы данных Sampdata.mdb, поставляемый в рамках приложения DAOView. Вот как получить эти программы и данные:

1. Находясь в Visual C++, нажмите F1 для вызова оперативной справки MSDN, или выберите Microsoft Developer Network из меню Windows Start.
2. Щелкните на кнопке Home в панели инструментов, чтобы перейти к начальной странице MSDN, а затем щелкните на гиперссылке Visual C++. Таким образом выполнится переход на начальную страницу Visual C++.
3. Нажмите Explore The Samples в правой части экрана. Скорее всего, появится приглашение вставить MSDN Disk 1 CD (оригинальный CD-ROM, используемый для установки MSDN, а не CD-ROM, с которого устанавливались Visual C++ или Visual Studio). Все примеры MSDN находятся на первом диске MSDN, а документация — на втором. После смены диска щелкните на OK.
4. При появлении страницы Visual C++ Samples (Примеры Visual C++) щелкните на гиперссылке MFC, а затем выберите Categorical List Of MFC Samples (Список примеров MFC по категориям). Выберите из появившегося списка гиперссылку Databases, затем — DAOVIEW.
5. При появлении диалогового окна DAOVIEW нажмите Click To Open Or Copy The DAOVIEW Project Files (Щелкните для открытия или копирования файлов проектов DAOVIEW). В диалоговом окне Visual C++ Samples нажмите на Copy All, выберите подходящий каталог и нажмите на OK.
6. С помощью той же процедуры скопируйте на жесткий диск приложение DAOTable.
7. После копирования всех файлов запустите Visual C++ и откройте проект DAOView.
8. Выберите из главного меню Build | Set Active Configuration. В диалоговом окне Set Active Configuration смените конфигурацию Win32 Unicode Debug на Daoview — Win32 Release. (В случае работы в Windows NT можно использовать конфигурацию Win32 Unicode Release, однако в Windows 95/98 этого делать нельзя.)
9. Выполните сборку приложения DAOView. Создайте каталог для данных и перетащите в него копию базы данных Sampdata.mdb из каталога проекта DAOView. (Убедитесь, что вы копируете, а не перемещаете файл. Таким образом в случае повреждения рабочей копии не придется устанавливать файлы примера заново.) Перетащите ярлык DAOView на рабочий стол, чтобы ускорить к нему доступ. Запустите его, а затем попытайтесь открыть в программе свою копию базы Sampdata. На рис. 19.1 показано, как исполь-

зовать программу для просмотра структуры таблиц и запросов базы данных Sampdata.

Доступ к базе данных через ODBC

В нашем следующем уроке ловкости вы узнаете, как подключиться к базе данных с помощью ODBC и как создать форму, позволяющую перемещаться по таблице базы данных и редактировать ее содержимое. На самом деле здесь мало ловкости, поскольку большую часть работы выполняет AppWizard. Вы же сделаете несколько указаний и щелчков, но не запишете ни одной строки кода.

Сперва базу данных Sampdata следует сделать источником данных ODBC. Для этого вам необходим установленный на компьютере 32-разрядный администратор источников данных ODBC (ODBC Data Source Administrator). Убедиться в его присутствии можно, посмотрев на панель управления, где должна находиться пиктограмма, показанная на рис. 19.2. Если приложение ODBC Data Source Administrator не установлено, установите его с оригинального CD-ROM Windows 95, Windows NT или Windows 98.

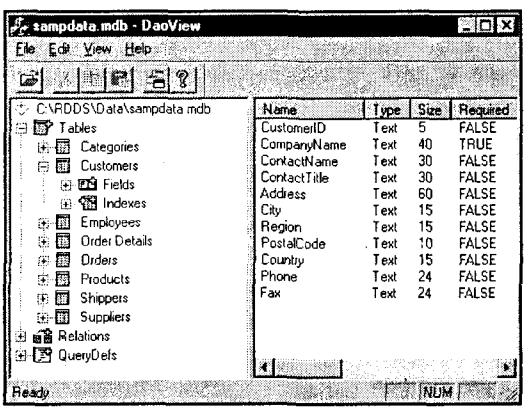


РИСУНОК 19.1. *Использование приложения DAOView для просмотра структуры Sampdata.*

РИСУНОК 19.2. *Пиктограмма администратора источников данных ODBC в панели управления Windows 95.*

Шаг 1: Создание имени источника данных (Data Source Name)

ODBC позволяет создавать программы, работающие одинаковым образом независимо от того, где сохранены реальные данные — в базе данных Oracle, Microsoft Access или в файлах индексированного последовательного метода доступа (ISAM), поддерживаемого dBASE и Paradox.

Для использования ODBC вначале потребуется создать DSN с помощью администратора источников данных ODBC. DSN определяет тип базы данных, местонахождение файлов данных и драйвер, используемый для доступа к файлам. В своих

MFC-программах вы работаете так, как будто DSN и является вашей базой данных, хотя на самом деле вы будете общаться с драйвером баз данных, направляющим ваши запросы самой системе баз данных.

В этой главе создается DSN с именем BucketOStuff. Важно помнить, что BucketOStuff не является базой данных — он только ссылается на нее. Подобного рода косвенность дает некоторое преимущество: вы можете изолировать все приложение от специфических характеристик рабочей базы данных. Сначала следует соединить BucketOStuff с базой данных SampData.mdb, используя драйвер Microsoft Access 7.0. Позже можно переопределить ссылку BucketOStuff на SQL-сервер или на базу данных Oracle — пока все имена и типы полей будут сохраняться, программа будет работать.

Для создания DSN BucketOStuff выполните следующие шаги:

1. Запустите 32-разрядный администратор источников данных ODBC, дважды щелкнув на его пиктограмме в панели управления. Когда откроется окно программы, перейдите на вкладку User DSN (см. рис. 19.3). (Было бы лучше использовать File DSN или System DSN, но AppWizard, к сожалению, поддерживает только источники User DSN.) Нажмите на Add.
2. Администратор источников данных ODBC должен знать, какой из драйверов будет использоваться. На рис. 19.4 показано диалоговое окно Create New Data Source. Выберите из списка драйвер Microsoft Access Driver (*.mdb) и нажмите на Finish.

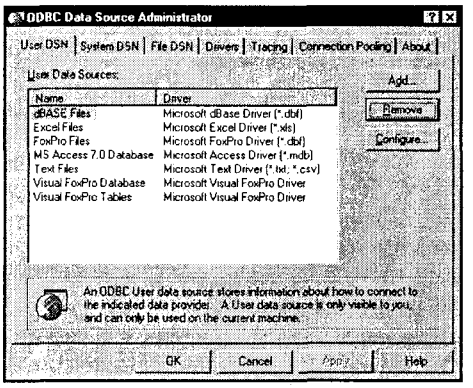


РИСУНОК 19.3. *Добавление нового источника данных ODBC.*

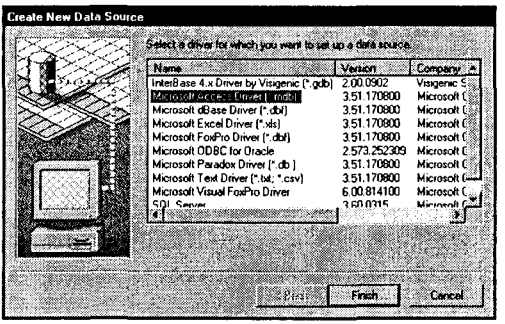


РИСУНОК 19.4. *Выбор драйвера ODBC.*

3. Каждый из доступных драйверов слегка отличается набором опции, но все они позволяют вводить имя источника данных и добавлять краткое его описание. В появившемся диалоговом окне ODBC Microsoft Access 97 Setup, показанном на рис. 19.5, введите имя источника данных "BucketOStuff". При желании можно добавить какое-то описание. Если вы хотите защитить базу данных паролем, добавьте имя пользователя и пароль, нажав на Advanced.
4. После именования источник данных необходимо подключить к базе данных. Нажмите на Select; диалоговое окно Select Database (рис. 19.6) позволяет отыскать на диске требуемый файл. Найдите копию Sampdata.mdb, сохраненную в своем каталоге данных, выберите ее и нажмите на OK.

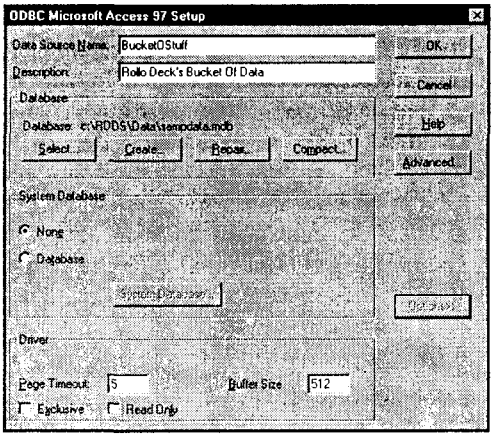


РИСУНОК 19.5. *Именование и описание нового источника данных.*

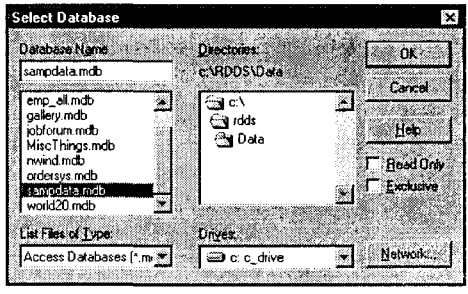


РИСУНОК 19.6. *Сообщение ODBC о местонахождении файлов.*

5. Теперь в администраторе источников данных ODBC появился новый пользовательский источник данных BucketOStuff. Нажмите на OK для закрытия приложения.

Теперь перейдем к AppWizard и создадим приложение баз данных.

Шаг 2: Создание ODBC-приложения с помощью AppWizard

Первое приложение будет называться OBos (Эта интригующая аббревиатура означает, естественно, **O**DBC **B**ucket**OS**tuff). Затем последуют приложения DBos (**D**AO **B**ucket**OS**tuff) и EBos (**OLE** DB **B**ucket**OS**tuff). В конечном счете вы перейдете к ABos (**A**DO **B**ucket**OS**tuff) и **R**Bos (**R**DO **B**ucket**OS**tuff). Ближе к завершению, поверьте, слово *Stuff* будет вызывать зевоту.

Начните с запуска AppWizard для создания нового проекта MFC App Wizard (exe). Назовите проект OBos и выполните следующие шаги:

1. В диалоговом окне MFC AppWizard — Step 1 выберите приложение SDI с поддержкой архитектуры "документ-представление".
2. В диалоговом окне MFC AppWizard — Step 2 выберите Database View Without File Support. (Опция Database View With File Support нужна только тогда, когда приложение читает и записывает обыкновенные файлы и работает с данными, хранящимися в базе данных.) Когда окно будет выглядеть, как показано на рис. 19.7, нажмите на Data Source.
3. В группе Recordset Type выберите Dynaset. В то время как Snapshot (снимок) предоставляет доступ только к копии данных из базы данных, Dynaset позволяет напрямую обновлять базу данных. В группе Datasource выберите переключатель ODBC и выберите в выпадающем списке имя BucketOStuff, как показано на рис. 19.8. Нажмите на OK.

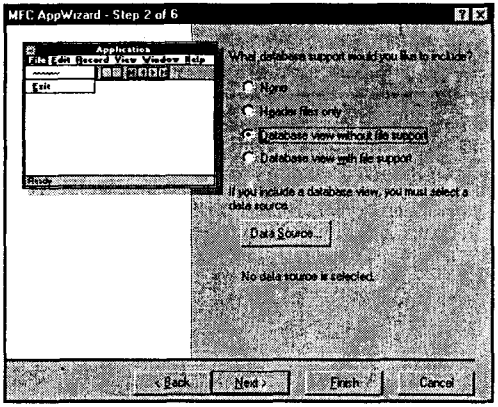


РИСУНОК 19.7. *Выбор опций базы данных.*

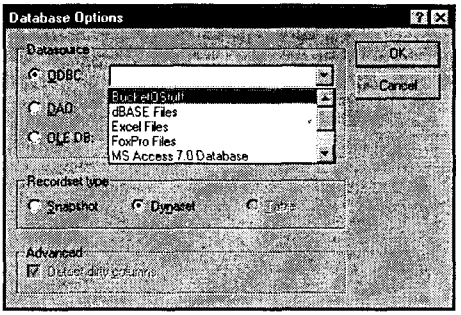


РИСУНОК 19.8. *Выбор источника данных ODBC.*

4. AppWizard откроет базу данных для определения ее структуры. Полученная информация выводится в виде списка в диалоговом окне Select Database Tables. Несмотря на свой заголовок, список помимо таблиц включает и другие записи, такие как запросы и представления базы данных. Выберите в списке элемент Customers (см. рис. 19.9) и нажмите на OK. (Если еще раз взглянуть на рис. 19.1, где база данных Sampdata просматривается с помощью приложения DAOView, можно увидеть структуру таблицы Customers — таблицы, с которой вам предстоит работать.) При появлении окна MFC AppWizard — Step 2 нажмите на Next.

СОВЕТ

Что вы означаете, таблицы?

Если вы внимательны, то заметите "s" (признак множественного числа) в конце заголовка диалогового окна Select Database Tables. Если вы предприимчивы, то можете даже попытаться выбрать несколько таблиц и увидите, что это сработает. Но позвольте дать вам совет: не выбирайте несколько таблиц.

Если в диалоговом окне Select Database Tables выбрать несколько таблиц, AppWizard создаст приложение, использующее SQL для получения декартова произведения всех выбранных таблиц. Результат может запросто содержать поистине огромное количество записей.

Если вы все же хотите работать с несколькими таблицами, задайте в диалоговом окне Select Database Tables первичную таблицу. После того как AppWizard создаст приложение, добавьте дополнительные классы, порожденные от CRecordSet.

5. В диалоговых окнах MFC AppWizard — Step 3, Step 4 и Step 5 примите установки по умолчанию. В окне Step 6 (см. рис. 19.10) отметьте для себя две особенности. Во-первых, базовым классом класса представления выступает **CRecordView**. **CRecordView** является подклассом **CFormView**, который, в свою очередь, позволяет создавать интерфейс с помощью Dialog Editor, подобно диалоговым приложениям, с которых в книге все начиналось. Во-вторых, четыре класса, обычно включаемые в приложение SDI, дополняются пятым — **COBosSet**. **COBosSet** является подклассом **CRecordSet**; он будет использоваться в качестве прокси для чтения и записи информации в базу данных.

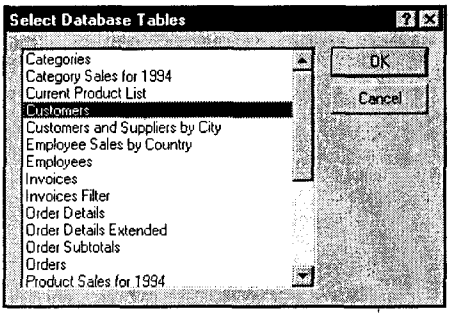


РИСУНОК 19.9. *Выбор таблицы базы данных.*

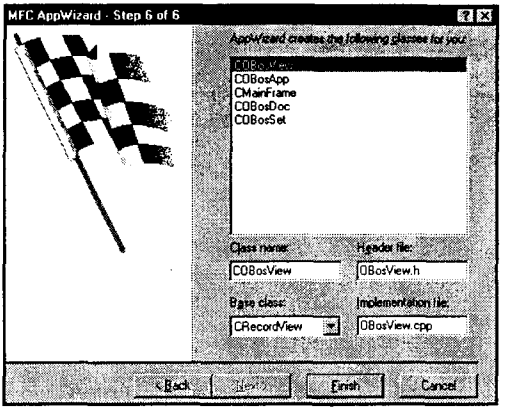


РИСУНОК 19.10. *Классы OBos.*

Шаг 3: Создание формы

Нажмите на Finish, а затем на OK, чтобы AppWizard сгенерировал приложение. Завершив создание файлов, AppWizard запустит Dialog Editor, предоставив вам возможность сконструировать новую форму. Приложения **CRecordView** строятся на основе диалогового окна, выступающего в качестве главного окна всего приложения. Главное диалоговое окно будет использоваться в качестве формы для вывода значений полей одной записи базы данных.

Для упрощения работы приложение AppWizard, основанное на классе **CRecordView**, также включает кнопки панели инструментов и пункты меню, позволяющие передвигаться по базе данных, перемещаясь к следующей, предыдущей, первой и последней записям. При перемещении от записи к записи поля формы обновляются значениями, хранимыми в таблице базы данных. Если в один из элементов управления базой данных ввести новое значение, оно запишется в базу при перемещении к другой записи.

Опять-таки, с целью упрощения каждое приложение **CRecordSet** имеет хотя бы один класс, порожденный от **CRecordSet**. Этот класс (генерируемый AppWizard) имеет по одной переменной для каждого поля (или колонки, если хотите) вашей таблицы. Эти прокси-переменные можно увидеть, нажав Ctrl+W для запуска ClassWizard, перейдя на вкладку Member Variables и выбрав из выпадающего списка Class name класс **COBosSet**.

На рис. 19.11 показаны переменные класса **COBosSet**. В этом случае все переменные являются объектами **CString**, но так происходит далеко не всегда. Каждая переменная, создаваемая ClassWizard, обычно имеет тип, совпадающий с типом соответствующего поля базы данных. Помимо упомянутых переменных, AppWizard создает набор функций обмена данными, подобных DDX-функциям, используемым в диалоговых окнах, которые служат для передачи информации между элементами управления, применяемыми для вывода информации, и полями базы данных, определенными в классе **CRecordSet**.

Для создания главной формы **OBos** выполните следующие шаги:

1. Убедитесь, что проект OBos загружен. Откройте в окне Workspace панель ResourceView и разверните папку Dialogs. Отыщите ресурс **IDD_OBOS_FORM** и дважды щелкните на нем, чтобы открыть редактор диалога. Удалите элемент статического текста To Do.

2. Вам не надо использовать все поля набора записей. Вместо этого создайте диалоговое окно, выводящее информацию, которая обычно присутствует на почтовой этикетке. Перетащите и поместите на форму элемент группы (groupbox), растяните его по периметру и установите заголовок в "Mailing Labels".
3. Перетащите на форму четыре элемента статического текста и расположите их в колонку с левой стороны формы. При помощи инструментов Alignment выровняйте элементы и промежутки между ними. Идентификатор каждого из элементов оставьте в **ID_STATIC**, выравнивание текста установите по правому краю и задайте следующие заголовки: "Name", "Address", "City, State, Zip" и "Country".
4. Поместите на форму шесть элементов редактирования и расположите их, как показано на рис. 19.12. Измените идентификаторы полей в соответствии с именами используемых полей базы данных. Вашими идентификаторами должны быть: **IDC_COMPANYNAME**, **IDC_ADDRESS**, **IDC_CITY**, **IDC_REGION**, **IDC_POSTALCODE** и **IDC_COUNTRY**. Когда форма примет вид, как на рис. 19.12, вы будете готовы к следующему шагу — подключению кода.

Шаг 4: Связывание отдельных частей

Для того чтобы заработал механизм передачи данных, потребуется связать каждый элемент редактирования с переменной **COBosSet**, представляющей поле базы данных. Выполняется это связывание за счет *внешнего подключения (foreign connection)* — косвенного отображения элемента управления Windows на переменную класса **COBosSet**.

Вы уже знаете, как с помощью ClassWizard можно связать элемент управления, например, **IDC_ADDRESS** с переменной **CString m_Address**. В таком случае любые изменения **m_Address** копируются в элемент управления **IDC_ADDRESS** или из него во время выполнения функции **DoDataExchange()**.

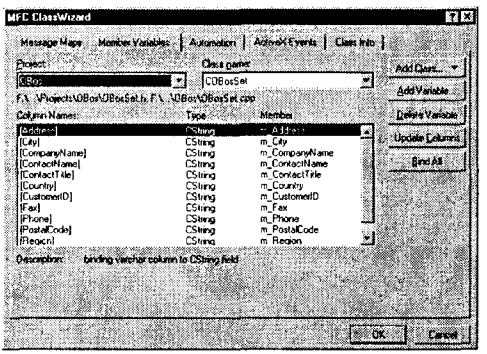


РИСУНОК 19.11. *Переменные COBosSet в ClassWizard.*

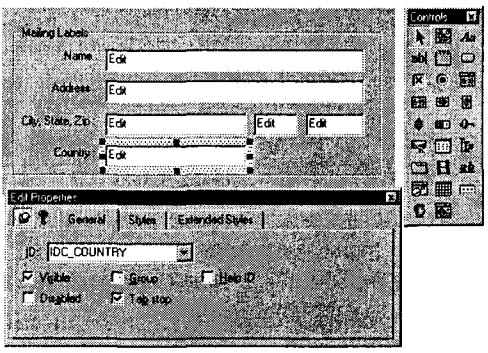


РИСУНОК 19.12. *Добавление полей редактирования.*

При использовании **CRecordView** ClassWizard позволяет связать элемент управления с одной из переменных класса **COBosSet**. Делается это через переменную

m_pSet класса **COBosView**, указывающую на объект **COBosSet**. Выполните следующие шаги:

1. Откройте ClassWizard, нажав Ctrl+W или выбрав из главного меню View | ClassWizard. В выпадающем списке Class Name выберите класс **COBosView**. Перейдите на вкладку Member Variables, на которой отображаются идентификаторы только что добавленных элементов редактирования.
2. Последовательно выбирая каждый из элементов, нажимайте Add Variable, как будто работаете с обычным диалоговым окном. Обратите внимание, что когда появляется диалоговое окно Add Member Variable, поле Member Variable Name уже не является обычным элементом редактирования — теперь оно представляет собой выпадающий список, содержащий косвенные ссылки на переменные типа **m_pSet->m_Address**. Это показано на рис. 19.13. Свяжите каждый элемент управления с корректным полем **CRecordSet**, используя переменную типа **CString**, которая и будет содержать значение поля.

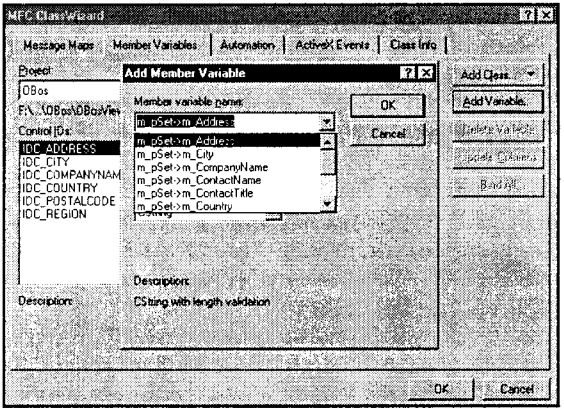


РИСУНОК 19.13.
Создание в ClassWizard внешних подключений.

3. Каждая внешняя переменная, добавленная ClassWizard, позволяет включить проверку длины вводимых данных. Это очень важно при создании приложения на основе **CRecordView**. При записи полей в базу данных следует проявить внимательность, чтобы не отослать символов больше, чем может разместиться в поле. Если еще раз взглянуть на рис. 19.1 (или запустить приложение DAOView и открыть Sampdata.mbb), можно заметить, что каждое поле таблицы Customers обладает максимальной длиной. Введите эти длины в поле Maximum Characters, чтобы установить проверку длины строки каждой внешней переменной, включенной в класс **COBosView** (см. рис. 19.14). Теперь ваше приложение перед отправкой данных в базу будет выполнять проверку их корректности.

Теперь откомпилируйте и запустите программу. Из рис. 19.15 видно, что по базе данных можно передвигаться при помощи кнопок панели управления прямо как в видеомагнитофоне. Кроме того, перемещаться можно и с помощью команд меню Record. В случае изменения данных приложение запишет их в базу при перемещении к другой записи. Неслабо; при этом не написано ни одной строки кода.

Для того чтобы сделать OBos полнофункциональной утилитой поддержки таблицы, придется добавить некоторый код. В следующем разделе будет показано, как фильтровать данные, таким образом отображая требуемые записи. Затем мы рассмотрим, как создать код для добавления и удаления записей.

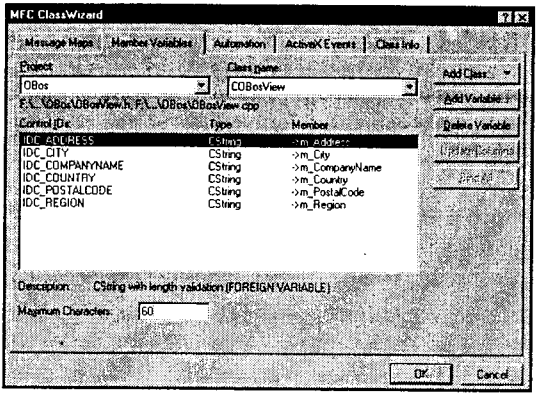


РИСУНОК 19.14. *Установка в ClassWizard проверки корректности полей ввода.*

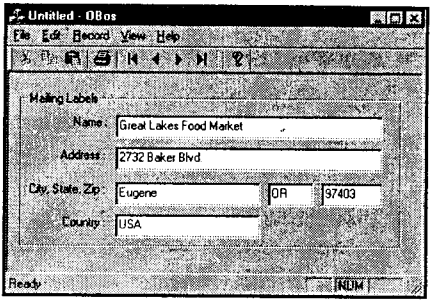


РИСУНОК 19.15. *Приложение OBos.*

Добавление кода в OBos

В приложении **CRecordView** класс документа практически ничего не делает, поскольку хранит и получает информацию сама база данных. Поэтому класс **COBosDoc**, например, не имеет функции сериализации. Класс документа просто содержит объект **CRecordSet** как общедоступный элемент.

При запуске приложения функция класса представления **OnInitialUpdate()** сохраняет адрес объекта документа **CRecordSet** в своей переменной **m_pSet** следующим образом:

```
m_pSet = &GetDocument()->m_oBosSet;
```

Затем осуществляется вызов функции **OnInitialUpdate()** базового класса, которая заполняет набор записей данными из базы, создает на форме элементы управления и передает им данные из первой записи набора.

После того как класс представления получит объект **CRecordSet**, но прежде чем он вызовет **CRecordView::OnInitialUpdate()**, есть возможность определить, какие записи следует получить за счет установки значений полей **m_strFilter** и **m_strSort** объекта **CRecordSet**. **m_strFilter** определяет *какие записи* необходимо получить, а **m_strSort** — в *каком порядке*.

Вот как это работает. Предположим, требуется вывести только записи заказчиков, расположенных в США. Добавьте следующую строку непосредственно перед вызовом **CRecordView::OnInitialUpdate()**:

```
m_pSet->m_strFilter = "Country = 'USA'";
```

Аналогично, если необходимо отсортировать записи по значению PostalCode, следует добавить такую строку:

```
m_pSet->m_strSort = "PostalCode";
```

Если **strFilter** и **m_strSort** пусты (по умолчанию это так), набор записей будет содержать все записи таблицы без какого-либо порядка — это как раз и есть BucketOStuff (пучок вещей).

Добавление фильтра записей

Вместо жесткой привязки значений к переменным **m_strFilter** или **m_strSort**, давайте добавим элемент редактирования, позволяющий определить, какие записи необходимо отобразить, во время выполнения. Рассмотрим соответствующие шаги.

1. Откройте проект OBos и загрузите в Dialog Editor форму **IDD_OBOS_FORM**. Расширьте диалоговое окно и переместите существующие элементы управления вниз, освободив сверху место под новые элементы управления.
2. Перетащите и установите на форму поле статического текста, элемент редактирования и кнопку. Разместите их, как показано на рис. 19.16.
3. Установите заголовок элемента статического текста в "Country". Идентификатор ресурса элемента редактирования установите в **IDC_COUNTRY_FILTER**. Заголовком кнопки должен быть "&Filter", а ее идентификатором ресурса — **IDC_FILTER**.
4. С помощью ClassWizard добавьте в элемент редактирования переменную. Назовите ее **m_CountryFilter**, категорию установите в Control, а тип — в **CEdit**. Когда окно будет выглядеть, как на рис. 19.17, нажмите на OK. Затем закройте ClassWizard.

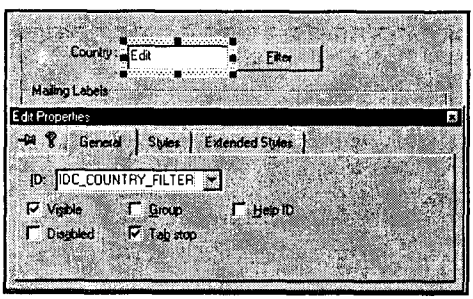


РИСУНОК 19.16. *Добавление полей фильтрации записей.*

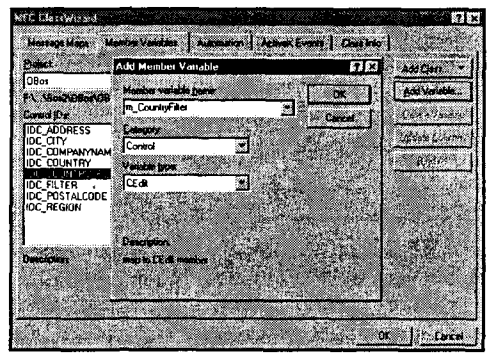


РИСУНОК 19.17. *Подключение элемента редактирования фильтра.*

5. Дважды щелкните на кнопке **IDC_FILTER**, чтобы создать в классе **COBosView** новый обработчик **OnFilter()**. Введите код, показанный в листинге 19.1.

Листинг 19.1. Функция COBosView::OnFilter().

```
void COBosView::OnFilter()
{
      CString filter = "";
      m_CountryFilter.GetWindowText(filter);
      if (filter == "")
      {
            m_pSet->m_strFilter = "";
      }
      else
      {
```

```
            m_pSet->m_strFilter = "Country = '"+filter+"'";
        }
        m_pSet->Requery();
        UpdateData(FALSE);
}
```

Как работает OnFilter()

Сперва обработчик **OnFilter()** извлекает текст из элемента редактирования **IDC_COUNTRY_FILTER** с помощью функции **GetWindowText()**. Когда элемент редактирования пуст, код отбирает все записи, устанавливая **m_pSet->m_strFilter** в "". Если элемент редактирования содержит значение, код объединяет его со строкой **"Country="**, тем самым ограничивая диапазон значений, получаемых из базы данных.

После создания набора записей изменение его значений **m_strFilter** или **m_strSort** не приведет к изменению диапазона значений или порядка чередования значений в наборе записей. MFC читает значения **m_strFilter** и **m_strSort** при получении данных. Чтобы заставить объект **COBosSet** создать новый набор записей, следует вызвать его метод **Requery()**.

После всего этого остается нерешенной одна небольшая проблема. Набор записей изменен, но программа все еще выводит запись из старого набора. Для обновления экрана потребуется перенести значения переменных набора записей в элементы управления диалогового окна, вызвав **UpdateData(FALSE)**.

Двигайтесь дальше и запустите новую версию OBos. При первом запуске приложение получает все записи из таблицы Customers. Если в элементе редактирования Country ввести значение "USA" и нажать на Filter, приложение выведет только тех заказчиков, у которых в поле Country установлено значение USA.

Добавление и удаление записей

Форму OBos все еще нельзя использовать для добавления данных в таблицу Customers, поскольку форме недостает для этого нескольких необходимых полей, таких как CustomerID. Давайте создадим специальную форму только для ввода данных. В процессе вы узнаете, как использовать обычное диалоговое окно вместе с формой **CRecordView**.

Для создания диалогового окна Add A Record выполните следующие инструкции:

1. Убедитесь, что проект OBos открыт, и выберите из главного меню Insert | Resource. При появлении нового диалогового окна Insert Resource, из списка Resource Type выберите Dialog, а затем нажмите на New.
2. Установите заголовок диалогового окна в "Add New Record", а идентификатор ресурса — в **IDD_ADD_NEW**.
3. Перетащите и установите на форме 11 элементов статического текста и 11 элементов редактирования. Расположите элементы в порядке, показанном на рис. 19.18. Идентификаторы элементов статического текста оставьте установленными в **IDC_STATIC**. Заголовки элементов статического текста и идентификаторы ресурса каждого из элементов редактирования установите в соответствии с именами полей таблицы Customer. Например, первое поле редактирования должно получить идентификатор **IDC_CUSTOMER_ID**.

4. Удерживая клавишу Ctrl, дважды щелкните на первом элементе редактирования **IDC_CUSTOMER_ID**. ClassWizard предупредит об отсутствии класса, объявленного для ресурса диалога **IDD_ADD_NEW**. Выберите New Class и нажмите на OK. Когда откроется диалоговое окно New Class, задайте имя **CAddNew** и породите класс от **CDialog**, как показано на рис. 19.19.

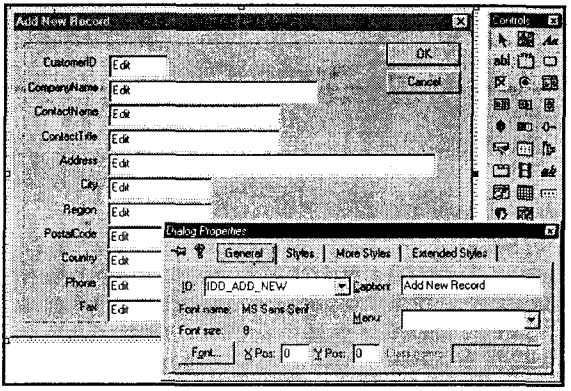


РИСУНОК 19.18. *Создание диалогового окна Add A Record.*

5. Добавьте переменные для каждого элемента управления диалогового окна Add New Record. Имя каждой из переменных задайте в соответствии с именем таких же переменных класса преставления, если таковые существуют. Тип переменных установите в **CString** и добавьте проверку на корректность вводимых значений для гарантии того, что длина каждого поля соответствует длине связанной записи таблицы Customer. (Для выяснения актуальных длин полей взгляните на рис. 19.1 либо воспользуйтесь программой DAOView.)
6. Откройте меню **IDR_MAINFRAME** в редакторе меню и добавьте разделяющую полосу после пункта меню Last Record. Добавьте два новых пункта меню. Первый пункт назовите "&Add New" и установите его идентификатор ресурса в **ID_RECORD_ADD_NEW**, как показано на рис. 19.20. Заголовок второго пункта меню установите в "&Delete" и определите для него идентификатор **ID_RECORD_DELETE**.
7. С помощью ClassWizard подключите к классу **COBosView** обработчики **COMMAND** для **ID_RECORD_ADD_NEW** и **ID_RECORD_DELETE**; назовите их **OnRecordAddNew()** и **OnRecordDelete()**. Поместите в них выделенный код из листинга 19.2.

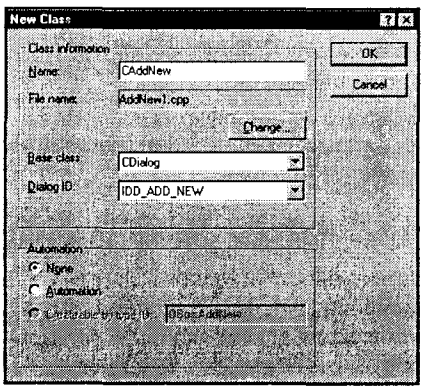


РИСУНОК 19.19. *Создание класса диалога CAddNew.*

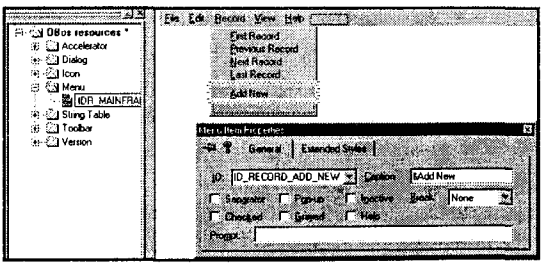


РИСУНОК 19.20. *Создание пунктов меню Add New и Delete.*

Листинг 19.2. Функции OnRecordAddNew() и OnRecordDelete().

```
void COBosView::OnRecordAddNew()
{
    //  1. Создание нового диалога Add New
    CAddNew dlg;
    if (dlg.DoModal() == IDOK)
    {
        // 2. Убедиться, что обязательное поле не пусто
        if (dlg.m_CompanyName.IsEmpty())
        {
            MessageBox(
                "Cannot add without company name");
            return;
        }
        // 3. Добавить новую запись в набор
        m_pSet->AddNew();
        // 4. Передать значения полей из диалога в набор
        m_pSet->m_Address      = dlg.m_Address;
        m_pSet->m_City         = dlg.m_City;
        m_pSet->m_CompanyName  = dlg.m_CompanyName;
        m_pSet->m_ContactName  = dlg.m_ContactName;
        m_pSet->m_ContactTitle = dlg.m_ContactTitle;
        m_pSet->m_Country      = dlg.m_Country;
        m_pSet->m_CustomerID       = dlg.m_CustomerID;
        m_pSet->m_Fax          = dlg.m_Fax;
        m_pSet->m_Phone        = dlg.m_Phone;
        m_pSet->m_PostalCode       = dlg.m_PostalCode;
        m_pSet->m_Region       = dlg.m_Region;

        // 5. Записать изменения в базу данных
        m_pSet->Update();

        // 6. Переместиться на новую запись
        m_pSet->MoveLast();

        // 7. Обновить экран
        UpdateData(FALSE);
    }
}

void COBosView::OnRecordDelete()
{
    // Попытка удаления записи
    try
    {
        m_pSet->Delete();
    }
    catch (CDBException* error)
    {
        AfxMessageBox(error->m_strError);
        error->Delete();
        m_pSet->MoveFirst();
        UpdateData(FALSE);
        return;
    }
    // Переместиться на новую запись
    m_pSet->MoveNext();
    if (m_pSet->IsEOF())
```

```
        m_pSet->MoveLast();
    UpdateData(FALSE);
}
```

8. Установите следующую строку в начале OBosView.cpp, непосредственно после других операторов **#include**:

```
#include "AddNew.h"
```

9. Откомпилируйте и запустите приложение.

Как работает добавление записей

Для добавления в базу данных новой записи код создает новый объект диалогового окна **CAddNew**, а затем вызывает его функцию **DoModal()**. Если **DoModal()** возвращает **IDOK**, код пытается добавить новую запись, но сперва убеждается, что пользователь ввел обязательное имя компании. В противном случае выдается вежливое сообщение и выполняется выход. Здесь можно поместить и любые дополнительные проверки.

Для добавления новой записи в **CRecordSet** код обращается к функции **AddNew()**, создающей новую пустую запись. После создания новой записи функция **OnRecordAddNew()** передает каждое поле из диалогового окна в эту новую запись.

После добавления новой записи в набор она на самом деле не записывается в базу данных до тех пор, пока не будет вызвана функция **Update()**. Прежде чем пытаться обновить базу данных, необходимо вызвать **IsUpdated()**, чтобы проверить допустимость обновления. Обновление может потерпеть неудачу по многим причинам. Например, в случае использования для двух разных записей одинаковое значение CustomerID, появляется сообщение об ошибке, показанное на рис. 19.21.

После того как **Update()** сохранит данные в базе, можно перейти на новую запись, вызвав **MoveLast()**. Даже если набор записей отсортирован или фильтруется, новые записи всегда добавляются в конец набора. Если конкретная запись должна занять "свое место", после вызова **Update()** обратитесь к **Requery()**. Однако в таком случае не будет способа переместиться в позицию только что добавленной записи, поэтому ее придется искать по всей таблице.

Как работает удаление записей

Удаление записей намного проще добавления. Для этого достаточно вызвать функцию **Delete()**, удаляющую текущую запись из набора данных и затем выполняющую перемещение на следующую запись. В функции **OnRecordDelete()** перемещение на новую запись осуществляется при помощи **MoveNext()** с проверкой признака конца файла с использованием **IsEOF()**. Если **IsEOF()** возвращает **TRUE**, значит была удалена последняя запись и вместо **MoveNext()** следует вызвать **MoveLast()**. Может возникнуть соблазн переписать код следующим образом:

```
if (m_pSet->IsEof())
    m_pSet->MoveLast();
else
    m_pSet->MoveNext();
```

Никогда так не делайте! Несмотря на рациональность приведенного кода, он работать не будет: признак конца файла не включается до тех пор, пока не будет предпринята попытка переместиться *за* последнюю запись.

Может показаться, что удаление записи — довольно безопасная операция, но это на самом деле это не так. Если вы удалите запись заказчика тогда, когда он все еще должен вам деньги, ваш бизнес окажется на грани краха — записи счетов покажут, что кто-то закупил партию компьютеров, а вы не будете иметь ни малейшего представления о том, кто бы это мог сделать.

Во избежание подобных проблем, в системы управления базами данных были введены *правила ссылочной целостности* (*referential integrity rules*), которые предотвращают удаление записей, от которых зависят другие записи. При попытке такого удаления генерируется исключение **CDBException**. В **OnRecordDelete()** функция **Delete()** помещена в блок C++ **try-catch** — при появлении ошибки программа выдает предупреждение и перемещается на первую запись. На рис. 19.22 показано сообщение, выведенное после неудачного удаления записи.

Использование DAO

В приложении баз данных, использующем ODBC, собственно приложение общается с драйвером ODBC, а драйвер ODBC общается с базой данных или другим источником данных, например, с текстовым файлом. Подобного рода косвенность обещает большую долю гибкости — просто переконфигурируйте ваш источник данных ODBC, и программа сможет работать с любой другой базой, начиная от базы данных ПК и завершая клиент-серверной базой данных на мейнфрейме.

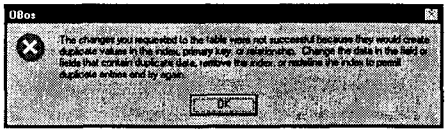


РИСУНОК 19.21. *Попытка добавления дублированной записи в таблицу заказчиков.*

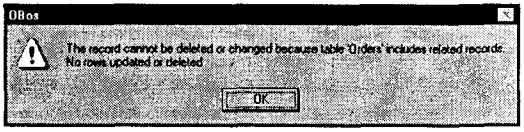


РИСУНОК 19.22. *Попытка удалить запись, имеющую зависящие от нее записи.*

С другой стороны, упомянутая гибкость требует жертв. Если используемая база данных обладает специальными возможностями, то получить к ним доступ через драйвер ODBC нельзя. И даже если это удастся, то, прежде всего, потеряется мобильность, присущая применению ODBC. Кроме того, доступ через ODBC к базам данных типа FoxPro и Access оказывается не настолько быстрым, как прямое подключение к собственно механизму упомянутых баз данных. Такие задачи позволят решать объекты доступа к данным — Microsoft Data Access Objects (DAO).

В DAO можно напрямую использовать механизм базы данных Access (Jet) для чтения и записи. Кроме того, механизм Jet позволяет напрямую читать и записывать большое множество файлов данных настольных ПК, таких как FoxPro и dBASE. Этот механизм можно использовать даже для подключения к удаленным источникам данных ODBC, хотя последнее несколько неэффективно.

Помимо своей эффективности, DAO обладает еще рядом преимуществ по сравнению с ODBC. Классы DAO в целом более мощны. Вы можете без труда отыскать в наборе записей конкретное значение, либо запомнить свое текущее местонахождение, разместив закладку. Кроме этого, можно воспользоваться коман-

дами Data Definition Language (DDL — язык объявления данных) для добавления таблиц, а также изменения структуры базы данных. В конце концов, DAO предоставляет объектно-ориентированный интерфейс, который особенно хорошо подходит для C++.

Создание приложения CDaoRecordView

Структура приложения MFC с архитектурой "документ-представление" подобна той, которая использовалась для классов ODBC. Изменилось лишь несколько имен классов. Например, приложение вместо **CRecordView** использует **CDaoRecordView**. Вместо указателя на объект **CRecordSet** работа осуществляется с указателем на объект **CDaoRecordSet**. Кроме отличающихся имен классов внесено мало изменений — имена переменных класса и способ, в соответствие с которым ClassWizard общается с полями данных, остались прежними. Если не обратить внимание на различия в именах классов, то можно вообще не заметить никакой разницы — по меньшей мере, до тех пор, пока не придется использовать **CDaoRecordSet**. Затем следует обратить внимание на присутствие дополнительных функций, подобных **FindFirst()** and **Seek()**.

Чтобы ощутить особенности работы с классом **CDaoRecordView**, давайте построим другое простое приложение для просмотра таблиц. В нем будет использована таблица Snippers той же базы данных Sampdata.mdb, что и в предыдущем примере. Таблица Snippers имеет только три поля, как показано на рис. 19.23.

Выполните следующие шаги:

1. Создайте новый проект AppWizard MFC и назовите его DBos. Создайте SDI-приложение с поддержкой архитектуры "документ-представление". В диалоговом окне MFC AppWizard — Step 2 выберите Database Without File Support. Нажмите на Data Source, чтобы вывести диалоговое окно Database Options, показанное на рис. 19.24. Выберите переключатель DAO, а в секции Recordset Type выберите Table. *DAO позволяет напрямую работать с таблицами*, чего не было в ODBC. Нажмите на кнопку с троеточием, выберите требуемую базу данных и щелкните на OK.
2. Откроется список таблиц базы данных. (Если в качестве источника записей выбран Dynaset или Snapshot, в списке будут присутствовать также и запросы к базе данных.) Выберите таблицу Snippers, как показано на рис. 19.25, и нажмите на OK.

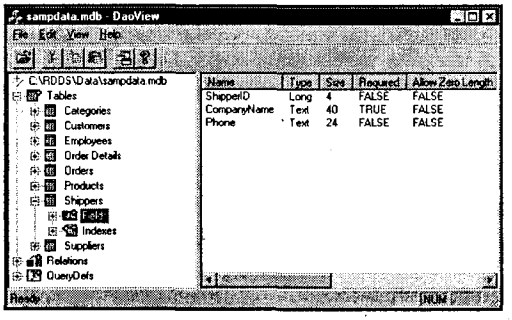


РИСУНОК 19.23. *Таблица Snippers в приложении DAOView.*

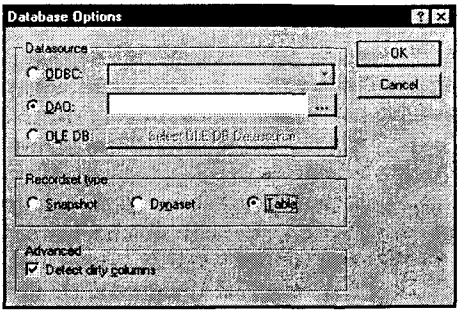


РИСУНОК 19.24. *Заполнение диалогового окна Database Options для приложения DBos.*

3. На остальных шагах AppWizard примите значения, предлагаемые по умолчанию. Заметьте, что вместо класса **CRecordView** приложение основывается на **CDaoRecordView**. Когда AppWizard завершит создание файлов для приложения, откройте ClassWizard и посмотрите на поля, сгенерированные для класса **CDBosSet**. Как видно из рис. 19.26, AppWizard создает поле для каждой колонки базы данных. На этот раз не все переменные имеют тип **CString** — переменная **m_ShipperID** — имеет тип **long**.

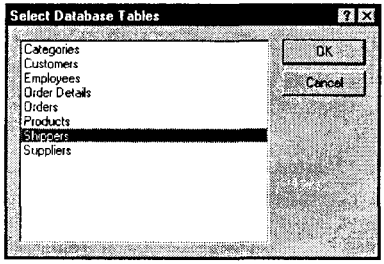


РИСУНОК 19.25. *Выбор таблицы базы данных для приложения DBos.*

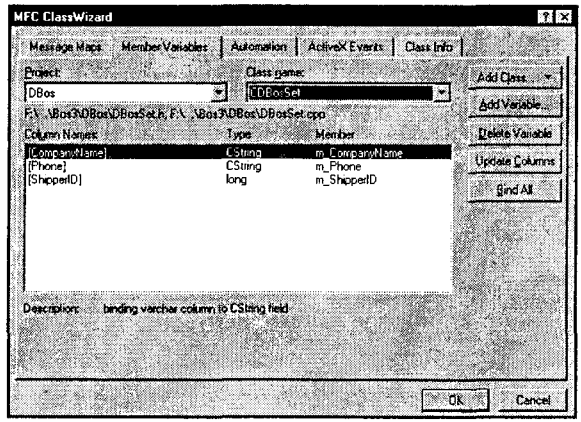


РИСУНОК 19.26. *Переменные CDBosSet.*

4. Откройте диалоговое окно главной формы в Dialog Editor, удалите элемент статического текста To Do и разместите на форме три панели статического текста и три элемента редактирования. Назовите их **IDC_SHIPPER_ID**, **IDC_COMPANY_NAME** и **IDC_PHONE**. Заголовки элементов статического текста установите, соответственно, в "ShipperID", "CompanyName" и "Phone". Разместите эти компоненты и воспользуйтесь командой меню Layout|Tab Order, чтобы установить их порядок обхода в соответствие с рис. 19.27.
5. С помощью ClassWizard свяжите каждый элемент управления с одной из внешних переменных класса **CDBosSet**, как показано на рис. 19.28. В диалоговом окне Add Member Variable используйте выпадающий список для выбора имени каждой внешней переменной. Длину поля Company Name установите в 40, а длину Phone — в 24. Не задавайте предельных значений для поля SnipperID.

Откомпилируйте и запустите программу. Можно, как и ранее, передвигаться по базе данных и изменять элементы данных — и все это без написания какого-либо кода. Как и прежде, можно внести некоторые улучшения. На этот раз мы оставим в покое фильтрование и сортировку, а поработаем над Add и Delete.

Улучшение DBos

Улучшение DBos начинается с изменения идентификатора поставщика с "чтения-записи" на "только-для-чтения". Для этого не нужно добавлять код — просто откройте форму в Dialog Editor и отметьте флажок свойства Read-Only для поля **IDC_SHIPPER_ID** (см. рис. 19.29). Поскольку поле предназначено только для чтения, нет необходимости выбирать стиль Number, хотя это и не повредит.

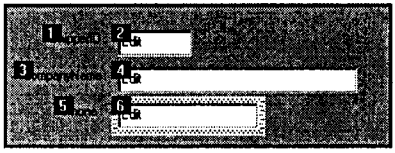


РИСУНОК 19.27. *Раскладка элементов управления для приложения DBos.*

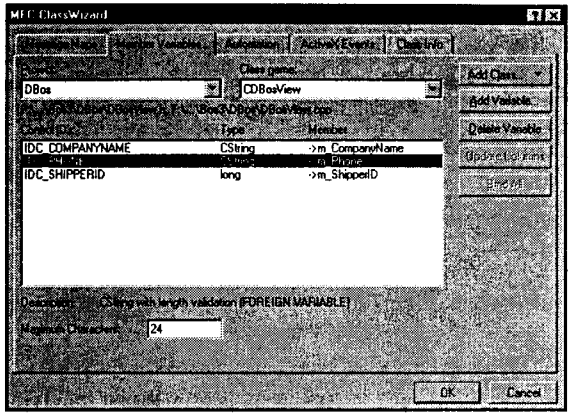


РИСУНОК 19.28. *Добавление переменных для полей базы данных.*

Вместо использования меню давайте на этот раз соединим команды Add и Delete с кнопками панели инструментов. Выполните следующие шаги:

1. Откройте ресурс панели инструментов **IDR_MAINFRAME** в редакторе панели инструментов (Toolbar Editor).
2. Удалите с панели инструментов кнопки Cut, Copy, Paste и Print. Вспомните, что удаление кнопок с панели инструментов осуществляется простым перетаскиванием их за пределы панели инструментов.
3. Добавьте в панель инструментов две новых кнопки, расположив их слева от кнопок со стрелками для движения по таблице. На первой кнопке нарисуйте контур листа бумаги. На второй нарисуйте небольшую мусорную урну. Вид кнопок показан на рис. 19.30. Установите идентификаторы ресурсов, соответственно, в **ID_ADD_NEW** и **ID_DELETE**. Снабдите каждую кнопку соответствующей подсказкой.

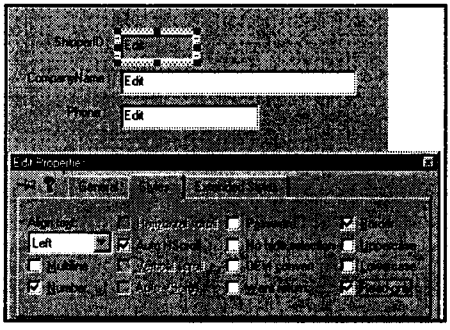


РИСУНОК 19.29. *Защита ShipperID от случайного повреждения.*

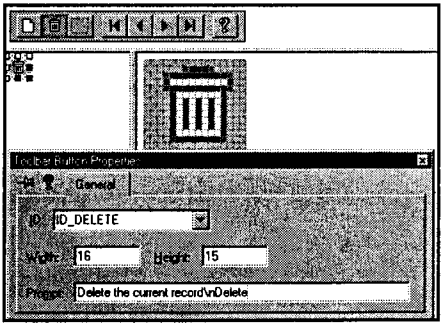


РИСУНОК 19.30. *Создание в панели инструментов кнопок Add и Delete.*

4. Добавьте в класс **CDBosView** новую переменную **bool m_IsAdding**. Используйте ClassWizard для создания обработчика **COMMAND** для **ID_ADD_NEW** и **ID_DELETE**, как показано на рис. 19.31. Поместите выделенный код из листинга 19.3 в функцию, сгенерированную AppWizard.

Листинг 19.3. Функции OnAddNew() and OnDelete().

```
void CDBosView::OnAddNew()
{
    m_pSet->MoveLast();
    long newID = m_pSet->m_ShipperID + 1;

    m_pSet->AddNew();
    m_pSet->SetFieldNull(&(m_pSet->m_ShipperID), FALSE);
    m_pSet->m_ShipperID = newID;
    UpdateData(FALSE);
    m_IsAdding = true;
}

void CDBosView::OnDelete()
{
    m_pSet->Delete();
    m_pSet->MoveNext();
    if (m_pSet->IsEOF())
        m_pSet->MoveLast();
    UpdateData(FALSE);
}
```

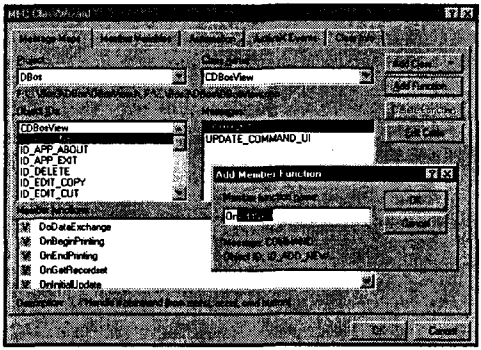


РИСУНОК 19.31. *Добавление обработчиков команд.*

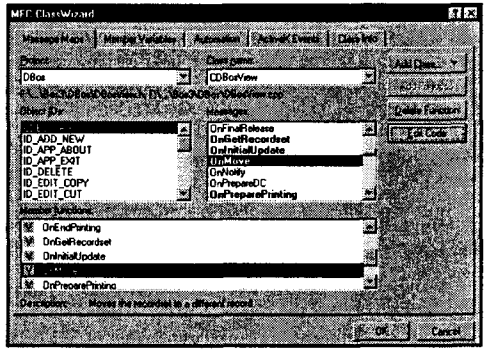


РИСУНОК 19.32. *Перекрытие виртуальной функции OnMove().*

Добавление и удаление в DBos

Как и в приложении OBos, для добавления новых записей в базу данных используется модальное диалоговое окно. Когда пользователь нажимает Add New, программа переходит на последнюю запись в файле, извлекает значение идентификатора SnipperID в качестве ключа для следующей записи, после чего вызывает **AddNew()**. Чтобы пользователь не вводил новый идентификатор SnipperID, вы вычисляете новое значение и сохраняете его в поле **m_SnipperID**. Обратите внимание на то, как функция **SetFieldNull()** предотвращает установку пустого значения в поле SnipperID.

После установки ключевого поля **OnAddNew()** установит поле **m_IsAdding** в **true** и завершит выполнение. Функция OnMove(), которая вызывает функцию **CDaoRecordSet::Update()**, сохраняет значения в базе данных.

В отличие от OBos-версии Delete, функция **DBosView::OnDelete()** предоставляет системе возможность распоряжаться исключениями; она просто удаляет текущую запись, а затем вызывает **MoveNext()**.

Перемещение на другую запись

При помощи модального диалогового окна, такого как использованное в процедуре добавления записи в OBos, базу данных можно обновить только после того, как это окно закроется. Однако в случае применения обычной, немодальной формы, типа класса представления DBos, следует поискать другую возможность для записи данных. Как происходит в большинстве случаев, данные можно сохранить только тогда, когда пользователь перейдет на другую запись. Для записи данных потребуется переопределить функцию **OnMove()**. Для этого выполните такие шаги:

1. Откройте ClassWizard и выберите класс **CDBosView**. В списке Messages выберите виртуальную функцию **OnMove()**. Нажмите на Add Function (см. рис. 19.32).
2. Как правило, класс **CDaoView** удовлетворительно обновляет записи. Вы хотите действовать только после того, как добавили новую запись и пользователь переместился с нее. Добавьте выделенный код из листинга 19.4, который и реализует упомянутое поведение.

Листинг 19.4. Перекрытая виртуальная функция CDBosView::OnMove().

```
BOOL CDBosView::OnMove(UINT nIDMoveCommand)
{
    if (m_IsAdding)
    {
        UpdateData();
        try
        {
            m_pSet->Update();
        }
        catch (CDaoException* err)
        {
            AfxMessageBox(err->m_pErrorInfo
                ->m_strDescription);
            m_pSet->MoveLast();
            err->Delete();
        }
        UpdateData(FALSE);
        m_IsAdding = false;
        return TRUE;
    }
    return CDaoRecordView::OnMove(nIDMoveCommand);
}
```

Откомпилируйте и запустите программу. Окно программы показано на рис. 19.33. Добавьте и удалите несколько записей, и посмотрите как все это работает.

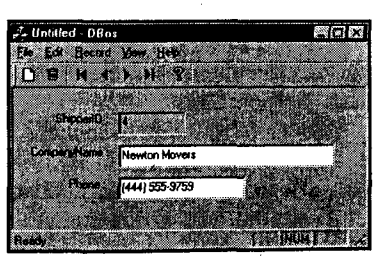


РИСУНОК 19.33. *Приложение DBos.*

Использование OLE DB

При построении приложений OBos и DBos вы, вероятно, заметили, что при выборе источника данных доступен еще один вариант выбора — OLE DB. OLE DB — это последняя технология Microsoft, которая предоставляет один интер-

фейс для всех ваших данных, где бы они не находились — в Web, в унаследованной системе либо в реляционной базе данных.

Настройка приложения на использование источника данных OLE DB даже проще настроек ODBC или DAO. После выбора OLE DB в AppWizard потребуется определить поставщика (provider) OLE DB и ввести информацию, необходимую для создания соединения. Затем, как и ранее, выбирается таблица для работы, и AppWizard создаст каркас приложения.

В случае использования классов OLE DB AppWizard создает класс, основанный на **COleDBRecordView**. В свою очередь, этот класс унаследован от **CFormView** и работает практически так же, как и **CRecordView** и **CDaoRecordView**. На форме диалогового окна размещаются элементы управления, а класс **COleDBRecordView** будет осуществлять получение данных и управление всей базой.

COleDBRecordView отличается от других главным образом в том, что ClassWizard не обеспечивает поддержку отображения внешних полей на ваши элементы управления — для этого потребуется вручную ввести код. Процесс хотя и не сложен, но достаточно утомителен.

На сопровождающем CD-ROM находится проект EBos, подобный тем, которые создавались ранее. Он использует классы OLE DB для перемещения по таблице Products базы данных Sampdata.mdb. Если вы заинтересованы в написании приложений, использующих шаблоны OLE DB, — взгляните на код, обратив особое внимание на функцию **DoDataExchange()**. Запустите программу на выполнение, пощупав ее в работе.

На подходе: ActiveX и "зеленая волна"

Ближе к концу главы вы понимаете, что обработка баз данных в Visual C++ — это не все, что могло бы быть. Написание программ несомненно сложнее, чем работа в средах Visual FoxPro, Visual Basic или Access.

Хотя ваши программы Visual C++ эффективнее программ, написанных на других языках, они все же существенно упрощены. Пока еще нет средств для соединения нескольких таблиц, прокрутки таблиц и форм "главный-детальный". Фактически, взглянув на программы AppWizard, неясно даже, где вы *начали* их писать.

В следующей главе мы покинем жестко-закодированные наборы записей, генерируемые AppWizard, и скажем "Привет!" разработке баз данных с использованием ActiveX.

Вы будете приятно удивлены.

Глава 20

Основы реляционных баз данных

"Когда я говорю слово," — презрительно сказал Шалтай-Болтай, — "это значит только то, что я подразумеваю, и ничего более."

"Вопрос в том," — сказала Алиса, — "можешь ли ты заставить слова обозначать много разных вещей?"

"Вопрос в том," — ответил Шалтай-Болтай, — "какое из слов будет главным — вот и все."

Если вы недавно перешли из другой области вселенной программирования в мир реляционных баз данных, то вам простительно ощущать себя подобно "Алисе в Зазеркалье" Льюиса Кэррола. Здесь вы найдете изобилие терминов, и некоторые из них, например, *база данных (database)*, имеют несколько значений.

База данных — это просто упорядоченное множество информации. С течением времени этот термин стал также означать программное обеспечение, управляющее информацией. Однако, фактически, программное обеспечение является не базой данных, а *системой управления базами данных* (СУБД).

Информация в базе данных может быть упорядочена несколькими способами. Ранние СУБД-продукты размещали данные либо в иерархической, либо в сетевой структуре. Затем, в 1960 г., доктор Е. Ф. Кодд (E. F. Codd), работающий на IBM, изобрел реляционную базу данных, которая организует информацию с использованием таблиц, колонок и строк.

В этой главе мы сделаем небольшой обзор технологий и концепций реляционных баз данных, включая язык структурированных запросов (SQL), предназначенный для получения информации из базы данных. На обратном пути из Страны баз данных мы посетим астероид ActiveX, чтобы узнать о странных существах: элементах управления, связывающих данные.

Что *представляет собой* реляционная база данных?

В реляционной базе данных вся информация сохраняется в *таблицах* (технически называемыми *отношениями (relations)*, или *совокупностью кортежей*). Как видно из рис. 20.1, эти таблицы очень похожи на электронные таблицы — они содержат ячейки, организованные в строки и колонки. Каждая из таблиц содержит информацию одного вида. Одна таблица может существовать для заказчиков, другая — для служащих, а третья — для счетов.

Employee ID	Name	Address	Job Class	Salary
1235	Dot Matrex	1234 South Street	Programmer	50,000
1305	Kay Sera	733 East 5th Street	Planner	150,000
1427	Barb Dwyer	12809 Mar Vista	Security	15,000
1811	Neil Doughn	623 Breckenridge Dr	Chaplain	15,000
1912	Paul Murkey	126 Oxford	Pollster	100,000
1011	Les Moody	119 Foothill Blvd	Psychologist	150,000
1112	Marge Novera	118 East Slauson Ave	Statistician	75,000

РИСУНОК 20.1.
Таблица базы данных.

Каждая *строка (row)* в таблице (технически называемая *кортежем (tupple)* описывает один экземпляр объекта, содержащегося в таблице. (Иногда строки называются *записями (records)*.) Например, каждая строка в таблице Employees может содержать информацию об одном служащем. Кроме того, таблицы будут разрабатываться так, чтобы каждая строка содержала некоторое отдельное значение. Ваша таблица Employees для каждого из служащих будет содержать по одной записи.

Каждая строка в таблице может включать несколько *колонок (columns)*, причем каждая колонка хранит значение некоторой характеристики записи. Например, колонки таблицы Employees, вероятно, будут включать адреса служащих, классификацию их работы и зарплату. (Колонки часто называются *полями (fields)* или *атрибутами (attributies)*.)

Эти "схематические" схемы

Пользу вы получите при использовании СУБД, а не набора случайных таблиц, поскольку именно программное обеспечение СУБД содержит важные подробности о базе данных, которой оно управляет. Например, СУБД знает имена содержащихся в ней таблиц и колонок этих таблиц.

Подобный вид информации называется *метаданными (meta-data)*, т.е. данными о данных. Как и все остальное в реляционных базах данных, эти данные сохраняются в таблицах — в этом случае в таблицах, используемых СУБД. Таблицы, описывающие структуру базы данных, называются *схемой (schema)* базы данных. Они позволяют программам определить, какие таблицы и поля содержатся в конкретной базе данных. Очень большие базы данных могут иметь несколько схем, объединенных в *каталог (catalog)*.

В базе данных все значения в каждой строке определенной колонки должны иметь одинаковый тип. В таблице Employees, например, колонка **Salary** может содержать только числа, но не текст. Таким образом, таблицы баз данных отличаются от электронных таблиц, поскольку значения, содержащиеся в колонке электронной таблицы, могут быть любого типа.

Во многих программах СУБД можно пойти дальше простого ограничения колонки определенным типом значения, такого как числовое или текстовое. Имеется возможность задавать *домен* (*domain*) — набор приемлемых значений в колонке. Для описания домена колонки объявляется специальный набор правил, называемых *ограничениями (constraints)*, которые впоследствии применяет СУБД. Например, можно ограничить колонку Salary, определив, что каждое значение должно быть более $10000 и менее $1000000. Доменные ограничения помогают избежать сохранения в базе некорректных данных.

Понятие ключей

Большинство таблиц спроектированы так, чтобы включать одну определенную колонку, гарантирующую наличие уникального значения. Ваша компания может иметь много служащих с именем, скажем, Джейн Смит, однако включив уникальную колонку **EmployeeID**, можно избежать путаницы.

Такая уникальная колонка называется *первичным ключом (primary key)* таблицы. Две колонки таблицы Employees не могут иметь одинаковые значения **EmployeeID**. Иногда одной колонки таблицы не хватает для достижения условия уникальности значений, тем самым делая невозможным использование одной колонки в качестве первичного ключа. В таком случае можно задать *составной первичный ключ (composite primary key)*, включающий значения нескольких колонок.

Первичные ключи гарантируют, что каждая строка таблицы содержит уникальное значение. Поэтому первичный ключ одной таблицы можно использовать в качестве колонки другой таблицы. Например, если имеется таблица продуктов Products, то, скажем, таблице счетов Invoice не нужны колонки имени продукта

ProductName и стоимости продукта **ProductPrice**. Вместо этого имеет смысл просто вставить первичный ключ таблицы Products в поле таблицы Invoice, и таким образом сохранять информацию только в одном месте.

При использовании первичного ключа в качестве связующего поля в другой таблице, такое поле называется *внешним ключом (foreign key)*. Рисунок 20.2 демонстрирует связь первичных и внешних ключей. Например, несколько счетов могут содержать записи изделий.

В этом разделе было введено несколько новых терминов, важнейшие из которых перечислены в табл. 20.1. Прежде чем двигаться дальше, убедитесь, что вы четко понимаете значение каждого из этих терминов. Если термин неясен, перечитайте его описание выше.

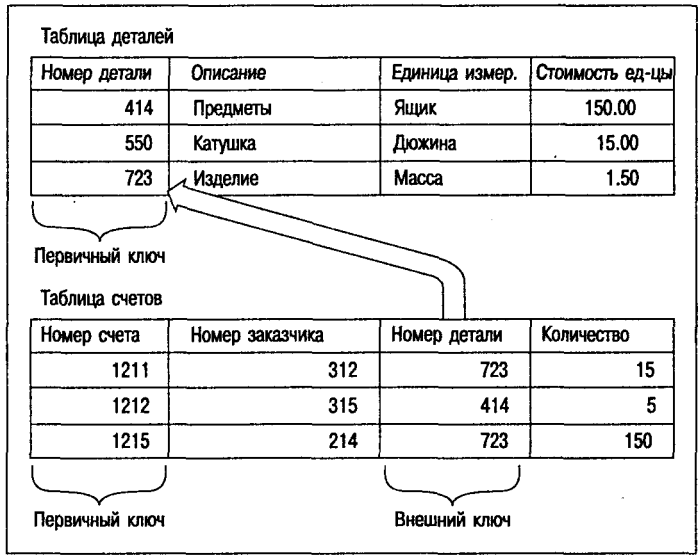


РИСУНОК 20.2. *Первичные и внешние ключи.*

Таблица 20.1. Термины реляционных баз данных.

Термин	*Значение*
Таблица (Table)	Один из видов сущности, состоящий из строк и колонок.
Строка (Row)	Часть таблицы, представляющая некоторый физический или воображаемый объект.
Колонка (Column)	Часть таблицы, содержащая один атрибут таблицы; атрибуты описывают свойства объектов
Метаданные (Meta-data)	Данные о данных. Позволяют базе данных хранить информацию о себе самой.
Схема (Schema)	Метаданные, описывающие содержимое базы данных, включая ее таблицы и структуры.
Каталог (Catalog)	Набор схем.

Таблица 20.1 (окончание)

Термин	*Значение*
Домен (Domain)	Диапазон приемлемых значений в колонке.
Ограничение (Constraint)	Правило, ограничивающее значения данных; управляется СУБД.
Первичный ключ (Primary key)	Колонка (или комбинация колонок), содержащая значение, уникальное для каждой строки таблицы.
Внешний ключ (Foreign key)	Колонка (или комбинация колонок), ссылающаяся на значение первичного ключа таблицы.

Взгляд на базу данных через призму DAO

Несмотря на то что метод получения информации из реляционной базы данных до некоторой степени стандартизован, не существует стандартного способа узнать о ее содержимом. Однако, чтобы заглянуть внутрь баз данных Microsoft Access, можно использовать классы DAO в MFC. Давайте посмотрим, как это работает, рассмотрев пример базы данных.

Каждый год американское отделение администрации информации по энергетике (U.S. Department of Energy's Energy Information Administration)(EIA) собирает статистику по производству и потреблению энергии во всем мире. Затем EIA публикует эту информацию в ежегоднике International Energy Annual. Исходные данные, используемые для создания этого ежегодного отчета, доступны в виде базы данных Microsoft Access; ее можно загрузить с Web-сайта EIA по адресу http://www.eia.doe.gov/emeu/world/main1.html. Чтобы предохранить вас от копирования относительно больших файлов, текущая редакция базы данных World Energy Database включена в состав сопровождающего книгу CD-ROM. На нем вы найдете две версии файла: World20.mdb (в старом формате Microsoft Access 2.0) и World97.mdb (в новом формате Access 97). База данных World Energy Database используется во всех примерах этой главы.

Пример DBExplore

Использовать классы DAO в MFC достаточно просто. Вначале потребуется создать объект **CDaoDatabase** и вызвать его метод **Open()**, передав ему имя базы данных Access, с которой необходимо работать. Объект **CDaoDatabase** содержит набор таблиц, запросов и связей. С помощью методов **CDaoDatabase** можно получать эти объекты либо информацию о них. По завершении вызывается метод **Close()** для освобождения подключения к базе данных.

До сих пор все написанные вами программы баз данных имели встроенные имена баз данных. При помощи **CDaoDatabase** можно использовать класс **CFileDialog**, чтобы разрешить пользователю открывать любую базу данных Access и исследовать ее содержимое. Давайте назовем программу DBExplore. Ниже приведены шаги по ее построению.

1. С помощью AppWizard создайте приложение однодокументного интерфейса с поддержкой архитектуры "документ/представление" и назовите его DBExplore. В диалоговом окне Step 2 выберите Header Files Only, как показано на рис. 20.3. Нажмите на Next.
2. В диалоговом окне Step 3 примите значения по умолчанию, а в окне Step 4 снимите отметку с флажка Printing. В окне Step 5 примите значения по умолчанию. В диалоговом окне Step 6 замените базовый класс представления на **CFormView**, а имя каждого из классов установите в **CDBEXXX** (за исключением класса **CMainFrame**), как показано на рис. 20.4.

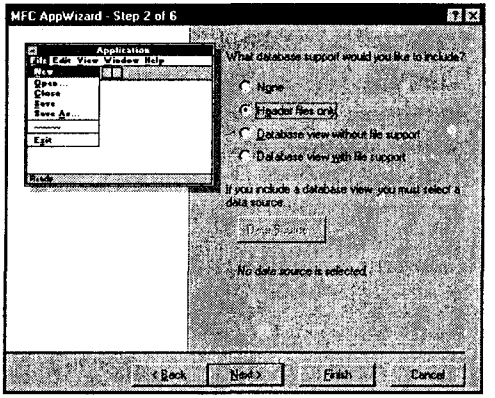


РИСУНОК 20.3. *Опции базы данных для проекта DBExplore.*

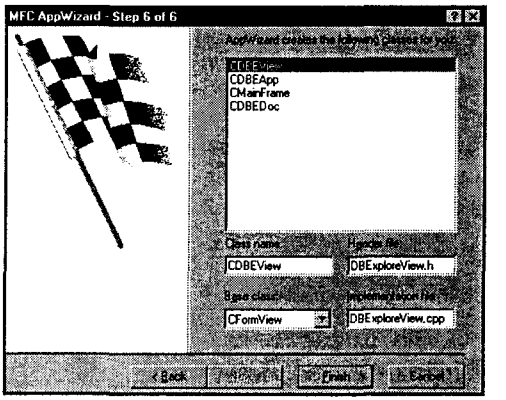


РИСУНОК 20.4. *Классы DBExplore.*

3. Откройте главную форму в Dialog Editor и добавьте в нее два элемента статического текста, два поля редактирования и окно списка. Установите размеры и размещение компонент в соответствии с рис. 20.5. Заголовки для элементов статического текста определите как "Database" и "Tables". Элемент редактирования назовите **IDC_DBNAME**, а список — **IDC_TABLE_LIST**.
4. С помощью ClassWizard добавьте в класс **CDBEView** переменные для каждого элемента управления, как показано на рис. 20.6. Переменную, связанную с элементом редактирования, назовите **m_DBName**; ее категорию установите в Control, а тип — в **CEdit**. Переменную списка назовите **m_TableList**; ее категорию установите в Control, а тип — в **CListBox**.
5. Откройте редактор меню и удалите меню Edit, а также пункты File | New, File | Save и File | Save As. Окончательное меню должно выглядеть, как показано на рис. 20.7.
6. Те же элементы удалите также из панели инструментов, оставив только кнопки File Open и Help. В конечном итоге панель инструментов должна выглядеть как на рис. 20.8.
7. В ClassWizard перейдите на вкладку Message Maps, а затем выберите из списка Class Name класс **CDBEView**. В списке Object IDs выберите **ID_FILE_OPEN**, в списке Messages — **COMMAND** и нажмите на Add Function. Примите имя **OnFileOpen**, как показано на рис. 20.9.

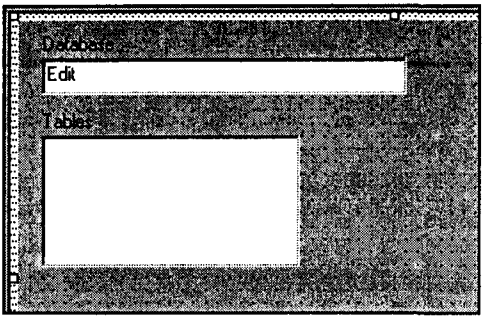


РИСУНОК 20.5. *Диалоговое окно DBExplore.*

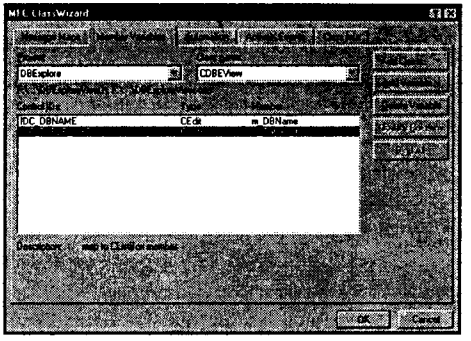


РИСУНОК 20.6. *Управляющие переменные для DBExplore.*

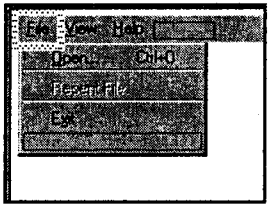


РИСУНОК 20.7. *Меню DBExplore.*

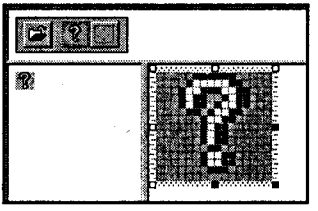


РИСУНОК 20.8. *Панель инструментов DBExplore.*

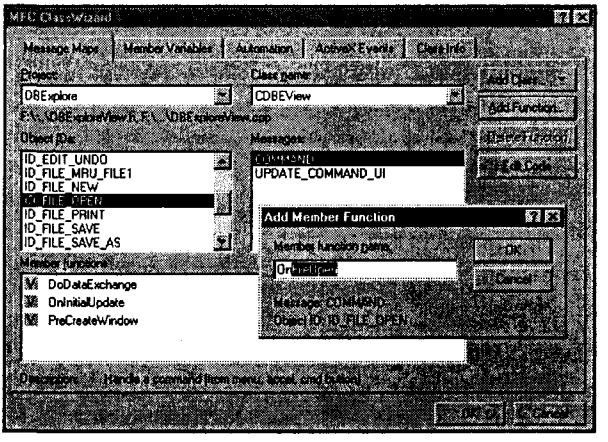


РИСУНОК 20.9.
Добавление обработчика меню OnFileOpen().

Использование классов DAO в OnFileOpen()

Когда пользователь выбирает File | Open или нажимает кнопку панели инструментов File Open, вначале создается объект **CFileDialog**, причем в конструктор передается ".mbd" как третий параметр. Это обеспечит вывод в диалоговом окне только файлов баз данных Access. Код выглядит следующим образом:

```
CFileDialog dlg(TRUE, "Open a Database File", "*.mdb");
```

Затем выполняется вызов функции **DoModal()** и проверка возвращенного значения на предмет равенства — ID_OK, а расширения файлов — на предмет ра-

венства ".mdb". (Для этого используется функция **GetFileExt()**.) Если любое из перечисленных условий не соблюдается, остаток шагов пропускается. Вот код:

```
if (dlg.DoModal() == IDOK && dlg.GetFileExt() == "mdb")
{
      // Здесь следует остаток кода
}
```

Имя базы данных можно получить из диалогового окна, вызвав **GetPathName()**, и сохранить его в поле **CEdit m_DBNames** с помощью **SetWindowText()**. Не забудьте удалить все имена из списка **CListBox m_TableList**. Вот код для выполнения этих задач:

```
m_TableList.ResetContent();
m_DBName.SetWindowText(dlg.GetPathName());
```

Для получения доступа к базе данных потребуется создать объект **CDaoDatabase**, а затем вызвать его метод **Open()**. По завершении вызывается **Close()**. Код должен выглядеть следующим образом:

```
CDaoDatabase db;
db.Open(dlg.GetPathName());
// Здесь производятся действия с базой данных
db.Close();
```

Список **m_TableNames** необходимо заполнить именами всех таблиц базы данных. Все объявления таблиц содержатся в коллекции **TableDef**. Для получения объявления каждой таблицы вызовите **GetTableDef()**. Вам незачем получать полное объявление таблицы, включающее всю информацию о колонках таблицы — необходимы только имена. Для этого создается объект **CDaoTableDefInfo** и вызывается функция **GetTableDefInfo()**. Затем пройдите по каждому объявлению таблицы, использовав функцию **GetTableDefCount()** для инициализации счетчика. В завершение можно использовать переменную **m_strName** для заполнения списка. Вот код, соответствующий перечисленным требованиям:

```
CDaoTableDefInfo info;
int nTables = db.GetTableDefCount();
for (int i = 0; i < nTables; i++)
{
      db.GetTableDefInfo(i, info);
      m_TableList.AddString(info.m_strName);
}
m_TableList.SetCurSel(0);
```

Добавьте этот код в функцию **OnFileOpen()** так, чтобы все действия выполнялись до вызова **close()**. Затем откомпилируйте программу и откройте в ней базу данных World Energy Database. Список таблиц можно прокручивать, как показано на рис. 20.10.

Функция **GetTableDefInfo()** возвращает объявление запрошенной таблицы базы данных в структуре **CDaoTableDefInfo**. (Несмотря на префикс "C" в имени, **CDaoTableDefInfo** — это обычная структура, а не класс.) Для заполнения списка используется поле **m_strName** этой структуры.

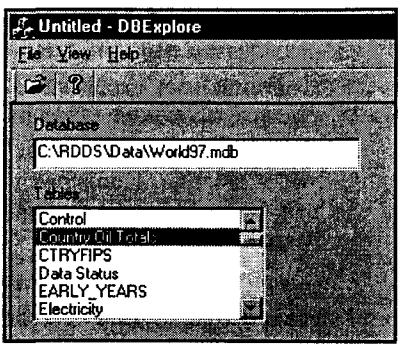


РИСУНОК 20.10. *Программа DBExplore.*

Кроме того, может заинтересовать поле **m_lAttributes**, описывающее несколько характеристик таблицы, на которую ссылается структура. С помощью логического оператора **AND (&)** с перечисленными ниже константами имеется возможность проверить специфические характеристики таблицы:

- ***dbAttachExclusive***. Внешне присоединяемая таблица, открытая для эксклюзивного доступа.
- ***dbAttachSavePWD***. Пользовательские идентификатор и пароль сохранены вместе с информацией о соединении.
- ***dbSystemObject***. Системная таблица Access (только для чтения).
- ***dbHiddenObject***. Скрытая таблица Access для временного использования (только-для-чтения).
- ***dbAttachedTable***. Таблица в присоединенной базе данных не ODBC-типа.
- ***dbAttachedODBC***. Таблица в присоединенной базе данных ODBC.

Поместив показанный ниже код добавления имен таблиц в список **m_TableList** внутри оператора **if**

```
if ((info.m_lAttributes & dbSystemObject) == 0)
{
      m_TableList.AddString(info.m_strName);
}
```

можно быть уверенным, что в списке таблиц будут перечислены только несистемные таблицы.

Исследование других объектов

Помимо таблиц объект **CDaoDatabase** содержит набор запросов и связей. Их можно модифицировать, добавлять новые и удалять существующие. В настоящий момент вы больше не будете работать с **CDaoDatabase**; давайте приступим к исследованию класса **CDaoTableDef**.

Объект **CDaoTableDef** можно использовать для получения поля, индекса и информации о корректности каждой таблицы базы данных. Объект **CDaoTableDef** также может изменять структуру существующей таблицы либо получать информацию из таблицы.

Давайте воспользуемся этим классом для получения информации о таблице, когда пользователь выбирает ее из списка **m_TableNames**. Вот, что необходимо сделать при изменении текущего элемента списка:

- Создать новый объект **CDaoDatabase**, используя имя, сохраненное при перечислении всех таблиц.
- Получить имя открываемой таблицы и передать его в объект **CDaoDatabase** с целью создания объекта **CDaoTableDef**.
- Вызвать функцию **GetFieldCount()** для определения общего количества полей в таблице. Пройтись по каждому полю списка, извлекая имя поля и его тип. Затем сохранить информацию в объекте **CListBox m_FieldList** (который вскоре будет создан.)

Это несложно. Вот как это сделать:

1. Откройте вашу главную форму в Dialog Editor. Добавьте в диалоговое окно элемент статического текста с заголовком "Fields" и список с именем

IDC_FIELD_LIST. Список Fields немного шире списка Tables. Ваше окно должно выглядеть, как на рис. 20.11. Отметьте флажок Use Tabstops на странице Styles диалогового окна List Box Properties.

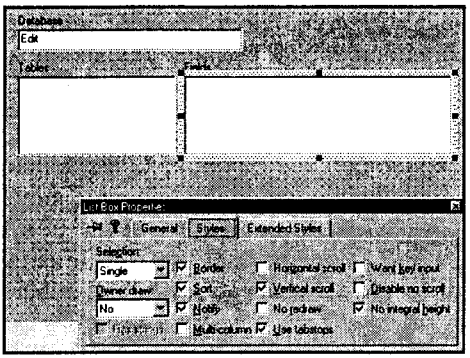


РИСУНОК 20.11. *Добавление списка Fields в диалоговое окно.*

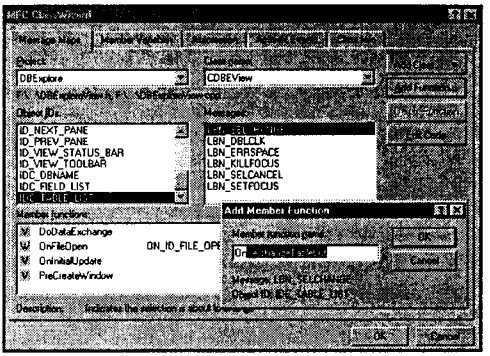


РИСУНОК 20.12. *Добавление обработчика сообщения OnSelchangeTableList().*

2. С помощью ClassWizard создайте в классе **CDBEView** переменную **CListBox**, связанную со списком **m_FieldList**.
3. С помощью ClassWizard добавьте обработчик сообщений **LBN_SELCHANGE** для окна списка **IDC_TABLE_LIST**. Функция обработчика должна находиться в классе **CDBEView** (см. рис. 20.12).
4. Поместите код из листинга 20.1 в функцию **OnSelchangeTableList()**. Несмотря на размеры, этот код очевиден — просто обратите внимание на пронумерованные комментарии и свяжите их с соответствующими пояснениями в главе.

Листинг 20.1. Обработчик CDBEView::OnSelChangeTableList().

```
void CDBEView::OnSelchangeTableList()
{
    // 1. Построить и открыть объект SDaoDatabase
    CDaoDatabase db;
    CString name;
    m_DBName.GetWindowText(name);
    db.Open(name);

    // 2. Построить и открыть объект SDaoTableDef
    CDaoTableDef table(&db);
    m_TableList.GetText(m_TableList.GetCurSel(), name);
    table.Open(name);

    // 3. Очистить список Field и установить в нем табулостопы
    m_FieldList.ResetContent();
    m_FieldList.SetTabStops(110);

    // 4. Определить количество полей в таблице
    int nFields = table.GetFieldCount();
    CDaoFieldInfo info;        // Сохранить информацию о поле
    CString sType;             // Сохранить описание типа

    // 5. Цикл по базе данных
    for (int i = 0; i < nFields; i++)
    {
```

```
            // 6. Извлечь информацию о поле из самого поля
            table.GetFieldInfo(i, info);
            // 7. Установить описание типа поля
            switch (info.m_nType)
            {
            case dbBoolean: sType = "Boolean";
               break;
            case dbByte:     sType = "Byte";
               break;
            case dbInteger: sType = "Integer (2 bytes)";
               break;
            case dbLong:     sType = "Long (4 bytes)";
               break;
            case dbCurrency:sType = "Currency";
               break;
            case dbSingle:   sType = "Single (4 bytes)";
               break;
            case dbDouble:   sType = "Double (8 bytes)";
               break;
            case dbDate:     sType = "Date/Time";
               break;
            case dbText:
                  sType.Format("Text %ld", info.m_lSize);
                     break;
            case dbLongBinary:   sType = "Long Binary";
               break;
            case dbMemo:     sType = "Memo";
               break;
            case dbGUID:     sType = "GUID";
               break;
            }
            // 8. Добавить в список имя и тип
            m_FieldList.AddString(info.m_strName + "\t" +
               sType);
      }
      // 9. Закрыть все, что было открыто
      m_FieldList.SetCurSel(0);
      table.Close();
      db.Close();
}
```

Реляционные базы данных: SQL

Некоторые СУБД для ПК (подобные dBASE или Paradox) включают как модель хранения данных, так и язык программирования. Раньше программисты считали это плохим решением — если компания решит перейти на другие языки программирования, то, скорее всего, не захочет оставить свои данные.

Большинство СУБД поддерживают SQL, позволяющий получать доступ к базам данных из языков COBOL, C++ или даже Snobol. (SQL произносится как "Эс-Кью-Эл" или "сиквел".) SQL предоставляет два уровня команд: те, которые позволяют манипулировать данными (Язык манипулирования данными, или DML), и те, которые позволяют манипулировать структурой базы данных (Язык объявления данных, или DDL).

Ввиду недостатка места, мы не сможем обсудить SQL полностью. Будучи программистом MFC и C++, вам часто потребуется работать с командами SQL, чтобы получать данные из базы. Первоначально необходимо понять команду SQL **SELECT**.

Команда SQL SELECT

Основной командой SQL является **SELECT**. Для получения всех строк и колонок таблицы Oil следует записать такой код:

```
SELECT * FROM Oil
```

Ключевые слова SQL **SELECT** и **FROM** в этом случае можно писать как в верхнем, так и в нижнем регистрах, однако часто верхний регистр используется с целью отделения ключевых слов от других частей синтаксиса SQL. Неключевые слова SQL-запроса вводятся таким же образом, как это было при создании таблицы. Некоторые СУБД более снисходительны в этом плане, но все же стоит выработать привычку писать переносимые запросы, распознаваемые широким диапазоном СУБД.

Команда **SELECT** уведомляет интерпретатор SQL о следующем:

- К какой таблице обращаться
- Какую колонку получать
- Какие строки включать в результат

Если посмотреть внимательно, можно заметить, что ни один из операторов в команде **SELECT** не сообщает интерпретатору SQL о том, какая *база данных* содержит таблицу Oil. База данных определяется в коде, который устанавливает соединение с базой данных и выполняет команду SQL.

Последней частью этой простой команды **SELECT** является звездочка (*). Звездочка в операторе **SELECT** просто сообщает: "получить все." В результате приведенный запрос вернет все колонки и строки. Часто, а на самом деле практически всегда — это не то, что вам необходимо. Вместо этого будет в точности задаваться, какие *именно* строки и колонки требуется получить. Вот как это делается.

Выбор полей

Чтобы задать возвращаемые поля, необходимо включить имя колонки таблицы или список имен колонок, разделенных запятыми. Если вы, например, хотите получить данные по добыче неочищенной нефти во всех странах, упоминаемых в таблице Oil, запишите такой оператор:

```
SELECT Country, Year, [Crude Production] FROM Oil
```

Если имя поля содержит пробелы (как поле **Crude Production**), заключите его в квадратные скобки. Интерпретатор SQL возвращает колонки в порядке, определенном оператором **SELECT**. Если год требуется разместить в левой колонке, запишите следующим образом:

```
SELECT Year, Country, [Crude Production] FROM Oil
```

Эта форма **SELECT** может оказаться более эффективной, чем форма со всеми колонками, особенно если таблица содержит длинные текстовые строки или дру-

гие большие значения. В то же время она обладает даже более важным преимуществом: при явном указании полей, которые необходимо получить, они всегда будут сохранять тот же порядок, даже если структура таблицы изменилась. Если сказать просто **SELECT***, то каждое изменение в структуре базы данных скажется на результате. В лучшем случае программа проигнорирует новые поля, а в худшем новое поле приведет к сбою программы.

Выбор строк: использование конструкции WHERE

Даже в случае явного использования **SELECT** для возврата определенных колонок, каждый запрос возвращает все строки таблицы. Если в таблице содержится несколько сотен записей, это не проблема. С другой стороны, если вы подключены к базе данных на удаленном сервере, есть вероятность получить сотни тысяч, а то и миллионы записей. Не следует даже упоминать, что в таких случаях серьезно страдает производительность. Отправка миллиона записей через Intenet не будет преступлением, однако и не прибавит вам друзей.

Для выбора специфических строк таблицы в оператор **SELECT** добавляется условие **WHERE**:

```
SELECT Year, [Crude Production] FROM Oil WHERE Country = 'Zimbabwe'
```

Этот запрос вернет только записи, в которых значение поля **Country** установлено в Zimbabwe. Обратите внимание, что значение поля (текстовая строка) заключается в одинарные кавычки — их опускание либо применение двойных кавычек приведет к сбою запроса. К тому же заметьте, что оператором сравнения является одиночный знак равенства (=). Такой синтаксис SQL может вызвать сердечный припадок у программистов MFC, поскольку C++ использует двойные кавычки для строк и двойной знак равенства (==) для операции сравнения.

Логическое условие

Искусство написания полезных конструкций **WHERE** заключается, главным образом, в принятии решения о том, какие записи необходимы, и последующей записи определяющего их условия. Это условие называется *логическим условием, или предикатом (predicate)*, поскольку для каждой записи оно либо истинно, либо ложно. Если для конкретной записи это условие истинно, то запись включается, в противном случае — исключается. Логические условия часто называются *реляционными выражениями (relational expressions)*, поскольку они обычно включают операторы сравнения или отношения.

К счастью, правила записи предикатов в SQL похожи на правила записи условий в C++. Следующий раздел посвящен наиболее общим различиям.

Операторы сравнения

Самая обычная форма логического условия SQL сравнивает значения. В табл. 20.2 приведены операторы сравнения, которые будут использоваться при создании SQL-запросов. Они эквивалентны соответствующим операторам C++ за исключением того, что неравенство вместо оператора != выражается оператором <>, а равенство вместо знака = — оператором ==.

Таблица 20.2. Операторы сравнения SQL.

Оператор	*Значение*
=	Равно
<>	Не равно
<	Меньше
>	Больше
<=	Меньше либо равно (т.е. не больше)
>=	Больше либо равно (т.е. не меньше)

Большинство значений, используемых в сравнениях, являются числами либо строками. Необходимо, чтобы оба операнда оператора сравнения были одного типа: числа надо сравнивать с числами, а строки — со строками. Смешивание типов приводит к ошибке или к неверному результату.

Сравнение строк

Написать логическое условие сравнения строк довольно просто, хотя тут есть две маленькие тонкости. Во-первых, сравнение строк в SQL зависит от регистра, как и в C++.

Во-вторых, компьютеры кодируют символьную информацию по-разному, используя *схемы упорядочения (collating sequences)*. Если присоединить таблицу, например, к базе данных мейнфрейма, то можно заметить, что символ "Z" предшествует символу "9", тогда как в базе данных ПК "9" предшествует "Z". Идеально ваши SQL-запросы не должны зависеть от последовательностей сопоставления, поскольку любой зависящий от них запрос может работать в одной базе данных, но приводить к ошибке в другой. Когда ваше приложение требует выполнения подобного рода сравнений, посоветуйтесь с администратором базы данных и спросите его о схемах упорядочения.

Составные сравнения

Иногда одного сравнения оказывается недостаточно. Предположим, требуется получить всю нефтяную статистику по Зимбабве и Анголе. В таком случае можно использовать булевы операторы для соединения реляционных выражений в составные реляционные выражения.

В табл. 20.3 перечислены булевы операторы SQL. Например, для генерирования ожидаемого результата следует записать:

```
SELECT [Crude Production]
FROM Oil
WHERE Country = 'Zimbabwe' OR Country = 'Angola'
```

Таблица 20.3 Булевы операторы SQL.

Оператор	*Результат истинный, если:*
AND	Оба логических условия истинны
OR	Истинно одно либо оба логических условия
NOT	Логическое условие ложно

Обратите внимание, что чем большим становится запрос, тем удобней помещать каждую конструкцию в отдельной строке. Чтобы указать порядок, в котором должны выполняться сравнения, можно, как и в C++, воспользоваться круглыми скобками.

Дополнительные логические условия

Большинство реализаций SQL включают другие логические условия, упрощающие написание SQL-запросов, возвращающих именно то, что необходимо. В табл. 20.4 показаны эти дополнительные логические условия SQL; они будут обсуждаться в следующих разделах.

Таблица 20.4. Дополнительные логические условия SQL.

Логическое условие	*Значение*
BETWEEN ... AND	Включает записи со значением поля, лежащим в заданном диапазоне.
NOT BETWEEN ... AND	Исключает записи со значением поля, лежащими в заданном диапазоне.
LIKE	Включает записи со строковым полем, совпадающим с заданным образцом.
NOT LIKE	Исключает записи со строковым полем, совпадающим с заданным образцом.
IN	Включает записи со значением поля, содержащимся в заданном списке.
NOT IN	Исключает записи со значением поля, содержащимся в заданном списке.
IS NULL	Включает записи с опущенным (пустым) значением необязательного поля.
IS NOT NULL	Исключает записи с опущенным (пустым) значением необязательного поля

Логическое условие **LIKE** позволяет задавать образец "шаблона", подобного файловому шаблону в DOS. Символ подчеркивания (_) в образце означает один произвольный символ, а знак процента (%) — любую строку символов.

Например, следующий запрос вернет данные по добыче неочищенной нефти в странах, имена которых оканчиваются на "ia":

```
SELECT [Crude Production]
FROM Oil
WHERE Country LIKE '%ia'
```

Логическое условие **IN** позволяет задать список значений вместо длинной последовательности условий **AND** и **OR**, например:

```
SELECT 'Crude Production'
FROM Oil
WHERE Year IN (1990, 1992, 1994, 1996)
```

Логическое условие **NOT IN** предоставляет дополнительный результат, включающий только записи со значением поля, *не* входящим в заданный список.

NULL

При проектировании таблицы вы определяете, какие из полей обязательны, а какие — нет. При обновлении записи СУБД не позволит не ввести запись, если она является обязательной. С другой стороны, если опустить значение необязательного поля, ошибки не будет — вместо этого СУБД сохранит в поле специальное значение, называемое *null*. (Пустое поле (null field) — это не числовое поле с нулевым значением и не строковое поле, содержащее пустую или чистую строку.)

Логическое условие **NULL** можно применять для идентификации записей без значений необязательных полей, т.е. с пустыми полями. Форма запроса такова:

```
SELECT [Crude Production]
FROM Oil
WHERE Units IS NULL
```

Соответствующее логическое условие **IS NOT NULL** определяет записи с непустыми значениями необязательных полей.

Элементы управления ActiveX для работы с базами данных

Как уже известно, оператор SQL **SELECT** позволяет получать значения из таблиц. В MFC результат запроса можно сохранить в объекте набора записей. Каждый класс базы данных в MFC имеет собственный вариант набора записей: ODBC-ориентированный класс **CDatabase** имеет набор **CRecordset**, а DAO-ориентированный класс **CDaoDatabase** использует **CDaoRecordset**.

Все наборы записей, с которыми мы работали в предыдущей главе, были, по существу, постоянными. AppWizard создавал особую переменную для каждой колонки в результирующем наборе, и в процессе обмена полями данные из набора записей передавались в переменные. Процесс хорош собой, но не весьма гибок.

Программу можно сделать более гибкой, используя классы DAO для идентификации полей в таблице, которую необходимо вывести, чтобы создать базу данных "на лету". Такая работа достаточно сложна, однако существует более простой способ — использование *элементов управления ActiveX, связывающих данные* (ActiveX *data-bound controls*).

Visual C++ поставляется с двумя наборами связующих элементов управления: Remote Data Objects (RDO) и ActiveX Data Objects (ADO). Элементы управления RDO используются для подключения к реляционным базам данных, основанным на SQL, через ODBC. Элементы управления ADO — новинка; они используют OLE DB. Microsoft сообщила, что элементы управления RDO больше не будут улучшаться, поэтому последующие разработки должны использовать ADO. Однако элементы управления ADO, поставляемые с Visual C++, помечены как "Release 1.0" (Выпуск 1.0). Неплохо застраховаться при переходе к конкретному набору компонентов.

Основы связывания данных

Механизм связывания данных ActiveX требует двух видов элементов управления. Во-первых, необходимы *элементы управления источником данных*, которые подключаются к базе данных и управляют движением по записям. Для каждого соединения потребуется один элемент управления источником данных.

Во-вторых, необходим *элемент управления, связывающий данные*, чтобы автоматически выводить результаты, получаемые элементом управления источником данных. Элементы управления, связывающие данные, существуют в двух видах: простые элементы управления, которые выводят значения одной записи, и комплексные элементы управления, выводящие одновременно несколько записей.

Для каждого элемента управления источником данных, будет по одному элементу управления, связывающему данные, а также несколько элементов управления, которые можно использовать с любым элементом управления источником данных. В табл. 20.5 перечислены элементы управления, связывающие данные. Колонки ADO и RDO показывают, с каким видом элемента управления данными работает конкретный компонент.

Таблица 20.5 Поставляемые в Visual C++ 6.0 элементы управления ActiveX, связывающие данные.

Имя	*RDO*	*ADO*	*Описание*
DataCombo	Нет	Да	Связывает поле из набора записей ADO с полем со списком
DataGrid	Нет	Да	Предоставляет текстовое, прокручиваемое, подобное электронной таблице, представление набора записей ADO.
DataList	Нет	Да	Связывает поле из набора записей ADO со списком.
DataRepeater	Нет	Да	Предоставляет метод использования элементов управления в прокручиваемой сетке.
Hierarchical FlexGrid	Нет	Да	Выводит табличные данные, включая строки и рисунки.
Chart	Нет	Да	Выводит массив данных набора записей в виде диаграммы.
DBCombo	Да	Нет	Связывает поле из набора записей RDO с выпадающим списком.
DBGrid	Да	Нет	Предоставляет текстовое, прокручиваемое, подобное электронной таблице, представление набора записей ADO.
DBList	Да	Нет	Связывает поле из набора записей RDO с окном со списком.
FlexGrid	Да	Нет	Выводит табличные данные, включая строки и рисунки, в виде только для чтения.
Calendar	Да	Да	Подключается к полю данных в наборе записей ADO или RDO.
Date and Time Picker	Да	Да	Подключается к полю даты или времени в наборе записей ADO или RDO.

Имя	RDO	ADO	Описание
Masked Edit	Да	Да	Позволяет выводить форматируемый текст (этот элемент управления используется только в Visual Basic).
RichText	Да	Да	Позволяет выводить текст RTF и большинство объектов OLE.

Добавление элементов управления ActiveX в DBExplore

Быстрее всего изучить особенности функционирования элементов управления ActiveX, связывающих данные можно путем их использования. Давайте возьмем программу DBExplore и сделаем из нее полнофункциональное средство для запросов и редактирования баз данных. Для этого воспользуемся элементами управления данными ADO и сеткой данных Microsoft.

Работу над проектом можно разбить на следующие этапы:

- Во-первых, добавить в проект элементы управления ADO Data Control и Microsoft DataGrid. Это достигается при помощи галереи компонентов так же, как она использовалась для других элементов управления ActiveX.
- Затем в главную форму DBExplore добавляются несколько элементов управления: многострочный элемент редактирования, в котором можно вводить запросы, кнопка Execute Query и элемент управления ADO Data Control.
- Далее создается диалоговое окно, в которое выводятся результаты выполнения запроса. В это диалоговое окно помещается компонент ADO DataGrid.
- В заключение записывается код, который выполняет запрос, активизируя элемент управления данными ActiveX и отображая новое диалоговое окно, содержащее решетку.

Шаг 1: добавление элементов управления данными ActiveX

Для добавления в проект элементов управления данными выполните следующие шаги:

1. Убедитесь, что проект DBExplore открыт в среде Visual C++. Выберите Select Project|Add To Project|Components And Controls, чтобы открыть диалоговое окно Components And Controls Gallery. Откройте папку Registered ActiveX Controls и выберите Microsoft ADO Data Control, version 6.0, как показано на рис. 20.14. Нажмите на Insert, чтобы добавить компонент в проект.
2. Нажмите на OK, когда появится диалоговое окно Confirm Classes (см. рис. 20.15). Это классы-оболочки над ActiveX, которые Visual C++ добавляет в проект.

3. Отыщите и выберите элемент управления Microsoft DataGrid Control, Version 6.0, как показано на рис. 20.16, затем нажмите на Insert. Убедитесь, что по ошибке не выбран элемент управления DBGrid: DBGrid работает только с элементами управления данными RDO, но не с ADO.
4. Щелкните на OK в диалоговом окне Confirm Classes, как это делалось для элемента управления ADO. На рис. 20.17 показаны классы, которые Visual C++ добавляет в проект для поддержки элемента управления DataGrid. Закройте диалоговое окно Components And Controls Gallery, нажав на Close.

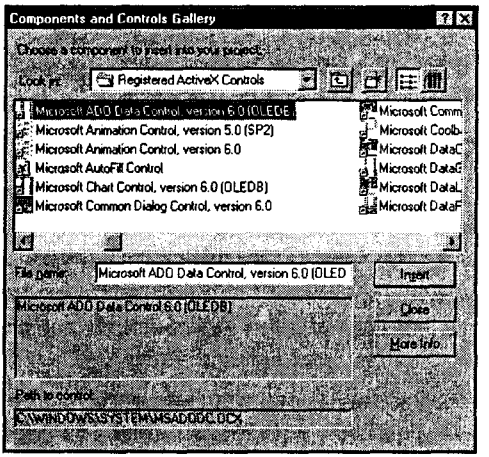


РИСУНОК 20.14. *Элемент управления Microsoft ADO Data Control.*

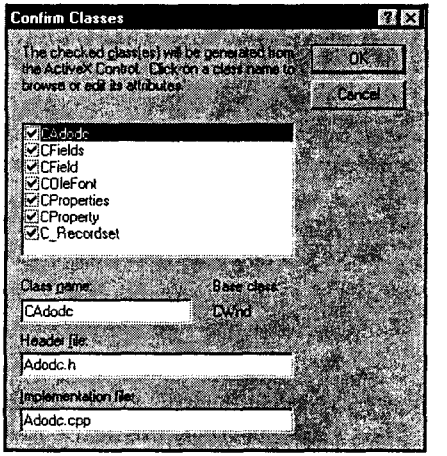


РИСУНОК 20.15. *Классы-оболочки ActiveX для элемента управления ADO Data Control.*

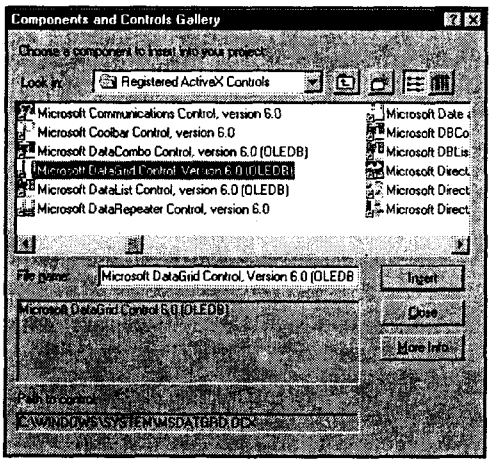


РИСУНОК 20.16. *Элемент управления Microsoft DataGrid Control.*

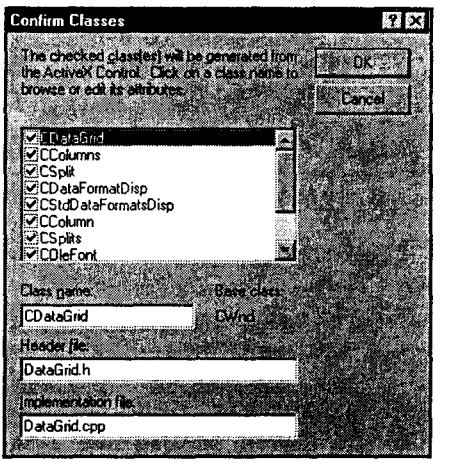


РИСУНОК 20.17. *Классы-оболочки ActiveX для элемента управления Microsoft DataGrid Control.*

Шаг 2: добавление компонентов в главную форму

После добавления в проект элементы управления DataGrid и ADO Data Control появятся в панели инструментов Controls. Теперь их можно просто перетащить на форму. Однако эти элементы управления таят несколько ловушек, поджидающих невнимательных.

Вот инструкции по добавлению и настройке новых компонентов:

1. Поместите на форму следующие элементы управления: ADO Data Control, элемент редактирования, панель статического текста и кнопку. Расположите элементы управления, как показано на рис. 20.18. Заголовок элемента статического текста установите в "SQL Query", заголовок кнопки — в "Execute Query", а заголовок элемента управления ADO Data Control — в "The ADO Grid". Идентификатор ресурса кнопки Execute Query установите в **IDC_SELECT**.
2. Откройте окно свойств нового элемента редактирования. Установите идентификатор ресурса в **IDC_QUERY**. На вкладке Styles отметьте флажки Multiline и Want Return и снимите отметку с флажка Auto HScroll. Диалоговое окно должно выглядеть, как показано на рис. 20.19.
3. Откройте окно свойств для нового элемента управления ADO Data Control. Установите идентификатор ресурса в **IDC_DB** и снимите отметку с флажка Visible на вкладке General. На вкладке Control выберите переключатель Use Connection String, как показано на рис. 20.20, и нажмите на Build.

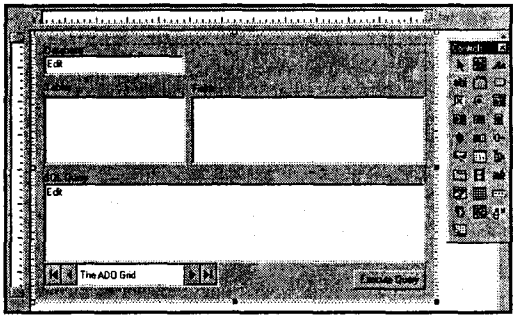


РИСУНОК 20.18. *Расположение компонентов на главной форме.*

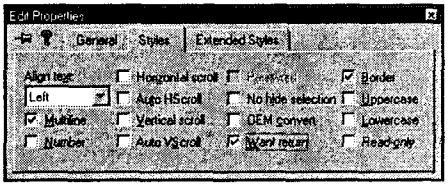


РИСУНОК 20.19. *Изменение свойств элемента редактирования IDC_QUERY.*

4. В появившемся диалоговом окне Data Link Properties выберите Microsoft Jet 3.51 OLE DB Provider, как показано на рис. 20.21. Не нажимайте на Next — просто щелкните на OK. Откройте окно свойств для элемента управления ADO Data Control, выделите текст в поле редактирования Connection String и с помощью клавиатурной комбинации Ctrl+X очистите его. В ClassView отыщите файл реализации класса **CDBEView** и вставьте вырезанную строку подключения в конец файла с помощью Edit | Paste или комбинации Ctrl+V. Закомментируйте этот текст — позже вы еще к нему вернетесь.
5. В диалоговом окне ADO Data Control Properties перейдите на вкладку RecordSource. В выпадающем списке Command Type выберите 1 — adCmdText, как показано на рис. 20.22. Эта установка сообщает элементу управления об ожидании им SQL-запроса. Сейчас вы запроса не вводите, по-

скольку предоставляете пользователю возможность вводить запросы во время выполнения. (Если DBExpolre необходимо использовать только для редактирования таблиц, измените соответственно Command Type — компонент будет автоматически открывать таблицу, когда она выбирается в списке таблиц DBExplore, как это делается сейчас с полями.)

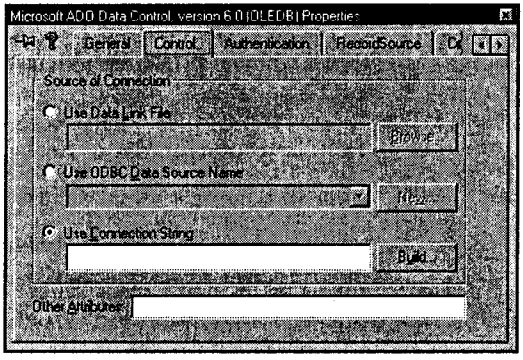


РИСУНОК 20.20. *Установка свойств элемента управления данными на вкладке Control.*

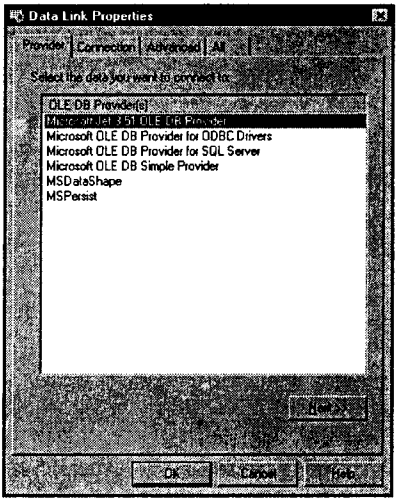


РИСУНОК 20.21. *Выбор провайдера OLE DB.*

6. После установки свойств элементов управления **IDC_QUERY** и **IDC_DB**, при помощи ClassWizard установите для каждого компонента управляющие переменные. Переменную для элемента редактирования назовите **m_Query**, а переменную для элемента управления DAO — **m_DB**.

Шаг 3: создание диалогового окна Query Results

При выполнении нового запроса необходимо вывести результаты в новом окне, размер которого можно изменять. В целях сохранения простоты программы для вывода результирующего множества воспользуемся модальным диалоговым окном. Вот как это сделать:

1. Выберите из главного меню Insert | Resource, и в появившемся окне Insert Resource выберите Dialog. Удалите из диалогового окна кнопки, очистите панель Caption, а затем установите на форме элемент управления DataGrid. Размеры диалогового окна и элемента управления установите в соответствие с рис. 20.23.
2. Откройте диалоговое окно Dialog Properties и установите на вкладке General идентификатор ресурса диалога в **IDD_GRID_DLG**. На вкладке Styles выберите из выпадающего списка Border опцию Resizing. На вкладке Extended Styles отметьте флажок Tool Window. Окна свойств должны выглядеть, как показано на рис. 20.24.
3. Откройте диалоговое окно DataGrid Properties и установите идентификатор ресурса в **ID_GRID**. На вкладке Control (см. рис. 20.25) отметьте флажки AllowAddNew и AllowDelete, чтобы в DBExplore можно было добавлять и удалять записи в таблицах.

4. Вызовите ClassWizard, выбрав View | ClassWizard либо нажав Ctrl+W. ClassWizard увидит новое диалоговое окно и предложит создать новый класс. Нажмите на OK; при открытии диалогового окна Add New Class назовите класс **CGridDlg** и породите его от **CDialog**. Нажмите на OK. В ClassWizard перейдите на вкладку Member Variables и добавьте новую переменную **CDataGrid m_Grid**, как показано на рис. 20.26.
5. Прежде чем закрыть ClassWizard и приступить к написанию кода, добавьте в класс **CGridDlg** обработчики сообщений Windows **WM_INITDIALOG** и **WM_SIZE**. Код будет добавлен на следующем шаге.

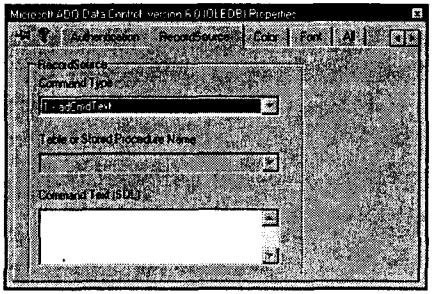


РИСУНОК 20.22. *Установка свойства Command Type элемента управления ADO Data Control.*

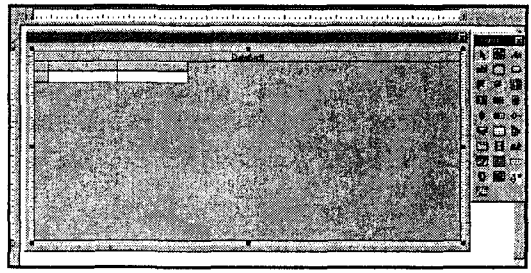


РИСУНОК 20.23. *Создание диалогового окна с элементом управления DataGrid.*

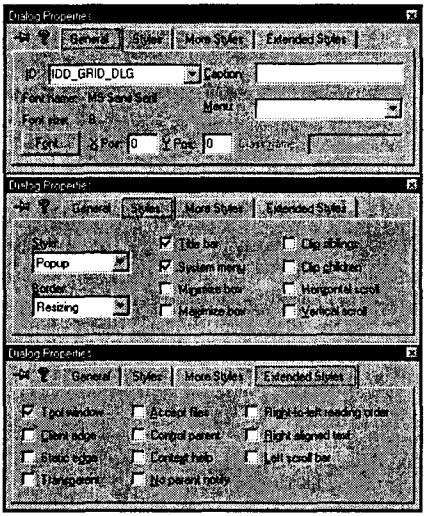


РИСУНОК 20.24. *Свойства диалогового окна IDD_GRID_DLG.*

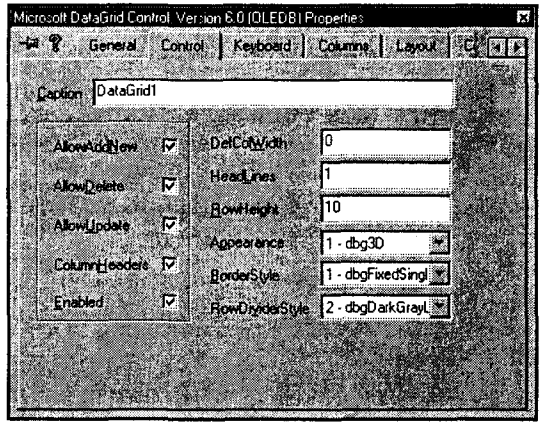


РИСУНОК 20.25. *Свойства элемента управления DataGrid.*

Шаг 4: активизация кнопки Query

В заключение потребуется соединить все части программы вместе, чтобы обеспечить ее работоспособность. Для этого просто добавьте пару переменных и напишите код обработки нажатия кнопки Execute Query. Вот как это сделать:

1. Откройте главную форму приложения в Dialog Editor и дважды щелкните на кнопке Execute Query. В окне Add Member Function нажмите OK. К функции **OnSelect()** добавьте код, приведенный в листинге 20.2. Обратите внимание, что код, описывающий выполняемые шаги, закомментирован и выделен. Воспользуйтесь текстом, который сохранен в конце файла, для установки первых двух строк в шаге 3 — они могут различаться в зависимости от провайдера OLE DB.

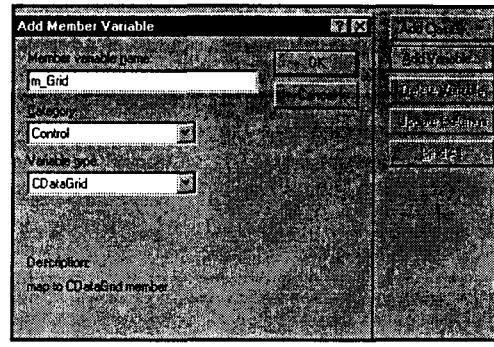


РИСУНОК 20.26. *Добавление переменной m_Grid.*

2. Прежде чем покинуть класс **CDBEView**, перейдите в начало файла и добавьте строку

```
#include "GridDlg.h"
```

чтобы код в шаге 6 смог быть откомпилирован.

Листинг 20.2. Функция CDBEView::OnSelect().

```
void CDBEView::OnSelect()
{
      // 1. Получить текст запроса от элемента управления m_Query
      CString query;
      m_Query.GetWindowText(query);

      // 2. Получить имя базы данных от элемента управления m_DBName
      CString db;
      m_DBName.GetWindowText(db);

      // 3. Получить остальную информацию о провайдере
      CString connect = "Provider=Microsoft.Jet.OLEDB.3.51;";
      connect += "Persist Security Info=False;";
      connect += "Data Source=" + db;

      // 4. Установить свойства ConnectionString и RecordSource
      m_DB.SetCommandType(1); // SQL
      m_DB.SetConnectionString(connect);
      m_DB.SetRecordSource(query);

      // 5. Прочитать данные
      m_DB.Refresh();

      // 6. Вывести сетку
      CGridDlg dlg;
      dlg.m_pCursor = m_DB.GetDSCCursor();
      query.Replace("\r\n", " ");
      dlg.m_Caption = query;
      dlg.m_Title = db;
      dlg.DoModal();
}
```

3. После добавления необходимого кода в класс представления также необходимо добавить сопровождающий код в класс диалога **CGridDlg**. Отыщите

функции **OnInitDialog()** и **OnSize()**, добавленные при создании класса, и поместите в них код из листинга 20.3.

Листинг 20.3. Функции OnInitDialog() и OnSize() класса CGridDlg.

```
void CGridDlg::OnSize(UINT nType, int cx, int cy)
{
      CDialog::OnSize(nType, cx, cy);
      // ЧТО СДЕЛАТЬ: Добавьте здесь свой код обработчика сообщений
      CRect rect;
      GetClientRect(rect);
      if (m_Grid.GetSafeHwnd() != 0)
            m_Grid.MoveWindow(rect);
}
BOOL CGridDlg::OnInitDialog()
{
      CDialog::OnInitDialog();

      // ЧТО СДЕЛАТЬ: Добавить код дополнительной инициализации
      m_Grid.SetRefDataSource(m_pCursor);
      m_Grid.SetCaption(m_Caption);
      SetWindowText(m_Title);
      return TRUE;          // Вернуть TRUE, если фокус не установлен
                            // на какой-то элемент управления
                            // ИСКЛЮЧЕНИЕ: Страницы свойств OCX будут
                            // возвращать FALSE
}
```

4. Добавьте в класс диалога три общедоступных переменных: **m_pCursor** типа **LPUNKNOWN**, а также **m_Title** и **m_Caption** типа **CString**.
5. Откомпилируйте окончательный код и протестируйте программу на нескольких SQL-запросах, изученных ранее в главе. На рис. 20.27 эти запросы показаны в работе.

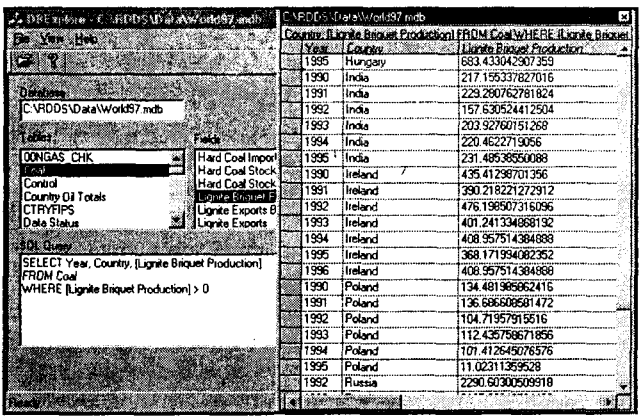


РИСУНОК 20.27.
Пример запроса в приложении DBExplore.

Как это работает — сокращенная версия

Работу элемента управления ADO Data Control определяют три свойства. Первое свойство — строка подключения, определяющая характеристики провайдера DB OLE, который посылает данные элементу управления. Этот элемент управления использовался для построения большей части строки. Последняя часть строки — параметр источника данных — вы получали из элемента редактирования **m_DBName**. Свойство устанавливалось за счет вызова функции элемента управления **SetConnectionString()**.

Свойство **Connect** сообщает элементу управления, *где* брать данные, а свойство **RecordSource** — о том, *какие именно* данные получать. В программе SQL-оператор выбирается из элемента редактирования **m_Query**. Полученный текст запроса посылается его элементу управления данными при помощи функции **SetRecordSource()**.

Для получения данных свойство **RecordSource** можно использовать тремя способами: при помощи оператора **SELECT**, как это делается здесь; получая все записи конкретной таблицы; или запуская хранимую процедуру, находящуюся на сервере базы данных. Между этими способами можно переключаться, устанавливая свойство **Command** с помощью функции **SetCommandType()**. Если установить его в 1, поле **RecordSource** будет интерпретироваться как SQL-запрос.

После установки трех свойств элемента управления данными остается лишь вызвать метод **Refresh()**, который получит требуемые записи. Ну а если у вас не будет способа их отобразить, вся процедура окажется напрасной. И вот здесь приходит на помощь элемент управления DataGrid.

Как работает DataGrid

Элемент управления DataGrid имеет много свойств, но нам нужно только одно — **DataSource**. Свойство **DataSource** связывает сетку данных с провайдером данных, таким как элемент управления ADO Data Control. Если свойства элемента управления данными определяются во время разработки, т.е. вы заранее знаете, что хотите получить — можете подключить сетку к элементу управления данными просто через окно свойств.

С другой стороны, если вы не знаете, что должно быть в **DataSource**, пока не запустите программу, как в случае DBExplore, свойство должно быть установлено во время выполнения, что вызывает некоторую проблему. Если посмотреть на классы-оболочки, генерируемые при импортировании элементов управления ActiveX, несложно заметить, что элемент управления DataGrid имеет метод **SetRefDataSource()**. К сожалению, нельзя просто передать ссылку элементу управления данными. Вместо этого потребуется вызвать функцию **CWnd GetDSCCursor()**, используя элемент управления данными, чтобы получить ссылку на набор данных, генерируемых элементом управления. Затем можно обратиться к функции **SetRefDataSource()**, чтобы сетка смогла использовать эти записи.

Этот факт может вызвать удивление, поскольку в документации по элементу управления данными и по сетке данных подобные сведения не указаны. Ответ прост: посмотрите на примеры программ, включенные в состав CD-ROM Visual C++. Документация по использованию элементов управления ActiveX в Visual C++ немногословна, однако кто-то же создал пример программы, демонстрирующий в точности то, что вам необходимо.

Следующая остановка – Web

Как видите, использование элементов управления ActiveX делает программирование баз данных в Visual C++ намного проще и гибче, чем ручное написание кода. Разумеется, есть несколько недостатков.

В связи с тем что большинство элементов управления ActiveX было написано для Visual Basic, документация по использованию их в Visual C++ часто немногословна. Некоторые элементы управления (например, Microsoft Masked Edit Control) при использовании в контейнерах Visual C++ работают некорректно. Другие (особенно новые сетки данных и элементы выбора даты и времени) не работают корректно в Dialog Editor, однако прекрасно работают в программах.

Элементы управления не только увеличивают вашу продуктивность как программиста баз данных, но и упрощают путь в мир Internet. В следующей главе будет показано, как новый класс **HTMLView** упрощает вывод содержимого HTML, и как при помощи элементов управления ActiveX, поставляемых с Internet Explorer, создать персональный Web-броузер всего лишь за несколько щелчков мышью.

Глава 21

Программирование для Internet: броузеры и другие клиенты

Если вы пользовались одним из первых Web-броузеров, которые были далеко не надежными, то знаете, что означает *dog browser* ("собачий" броузер). В этой главе фраза обретает совершенно новый смысл.

Не так давно словом *броузер (browser)* обозначался случайный покупатель. Сегодня же практически каждый знает, что броузер — это вид программного обеспечения. Тем не менее, броузеры пока не вторглись в повседневную жизнь программистов, поэтому большинство представляет броузер как загадочный черный ящик. Элемент управления Microsoft HTMLView все это изменил.

HTMLView видит весь мир

Visual C++ упрощает создание броузеров и других сетевых клиентов. Так же как класс **CEditView** помогает построить клон Notepad, а класс **CRichEditView** — имитировать приложение WordPad, HTMLView (появившийся в Visual C++ 6.0) предоставляет возможность создать собственную версию Internet Explorer, не написав ни одной строки кода.

При использовании ClassWizard для создания приложения, основанного на классе **CHtmlView**, программа после запуска автоматически подключается к Internet. Ваша программа будет визуализировать документы, созданные с использованием языка гипертекстовой разметки (Hypertext Markup Languade, или HTML), переходить по гиперссылкам и даже выполнять программы на Java. Для получения таких функциональных возможностей не потребуется писать *ни одной* строки кода. Однако, скорее всего, вы захотите добавить некоторые возможности типа кнопок вперед-назад, и, возможно, хронологический список.

Прежде чем приступить к работе, примите во внимание одно предостережение. **CHtmlView** очень похож в работе на класс **CRichEditView**, использующий в качестве основы класса представления элемент управления форматируемым текстом. Класс **CHtmlView** использует элемент управления Internet Explorer 4.0 (IE4) Web Browser Control, содержащийся в SHDOCVW.DLL. В отличии от множества элементов управления ActiveX, которые можно применять в приложении, элемент управления ActiveX Web Browser Control со своей программой распространять нельзя. Вместо этого потребуется распространять полную программу установки IE4, либо же заказчики должны иметь заранее установленный IE4 — в противном случае они не смогут запустить вашу программу. Чтобы поставлять вместе со своими приложениями IE4, необходимо подписать отдельное лицензионное соглашение; для получения подробной информации свяжитесь с Microsoft.

Сборка Web-броузера

Желание создать приложение, основанное на классе **CHtmlView**, может возникнуть по нескольким причинам. Скорее всего, появится необходимость добавить поддержку Web в существующие приложения или создать собственный броузер, предназначенный для конкретных целей. Давайте рассмотрим, как это можно сделать.

Предположим, вы — член клуба любителей собак. С помощью класса **CHtmlView** имеется возможность создать приложение, связывающее ваших приятелей с Web-сайтом American Kennel Club (AKC) и предоставляющее специальные возможности для записи информации об их собаках. Например, можно было бы использовать функциональность баз данных Visual C++, обсужденные в последних нескольких главах, чтобы предоставить членам клуба возможность хранить информацию о родословной своих питомцев и об их здоровье. Также может потребоваться добавить модуль учета, отслеживающий стоимость содержания своих любимцев и т.п.

Проявив немалое усердие, можно даже включить специальную технологию блокирования — отфильтровывание всех сайтов, содержащих, например, слово *"кот"*.

Мы не будем показывать, как это делается, но покажем, как создать базовый Web-броузер.

Давайте назовем проект Bowser — "всесобачий" Web-броузер. Выполните следующие шаги:

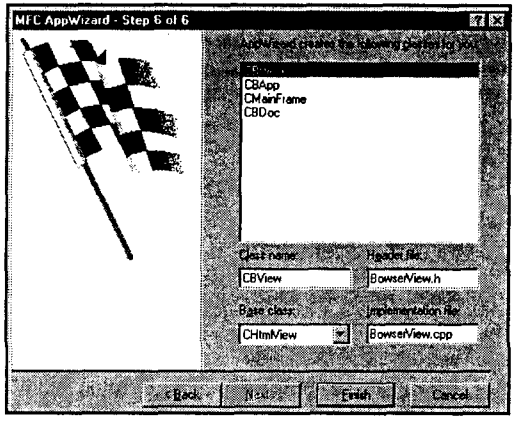


РИСУНОК 21.1. *Настройка классов броузера Bowser.*

1. Начните новый проект MFC AppWizard (exe) и назовите его Bowser. В диалоговом окне Step 1 выберите однодокументный интерфейс с поддержкой архитектуры "документ/представление". В окнах Step 2, 3, 4, 5 согласитесь с предлагаемыми значениями. В окне Step 6 измените базовый класс на **CHtmlView** и определите имена классов как **CBView**, **CBDoc** и **CBApp**. Когда окно Step 6 будет выглядеть, как показано на рис. 21.1, щелкните на Finish.
2. В окне ClassView разверните класс **CBView** и дважды щелкните на методе **OnInitialUpdate()** для загрузки окна редактора исходного кода. Отыщите строку

```
Navigate2(_T("http://www.microsoft.com/visualc/"),NULL,NULL);
```

и поместите в кавычки адрес Web-сайта American Kennel Club:

```
Navigate2(_T("http://www.akc.org"), NULL, NULL);
```

Откомпилируйте код и запустите программу. Если вы постоянно подключены к Internet — программа найдет Web-сайт AKC и выведет его начальную страницу (home page). В ином случае система будет пытаться подключиться, дозваниваясь к провайдеру услуг Inernet, как показано на рис. 21.2. Если же система не настроена на автоматический дозвон, соединиться со своим провайдером придется вручную.

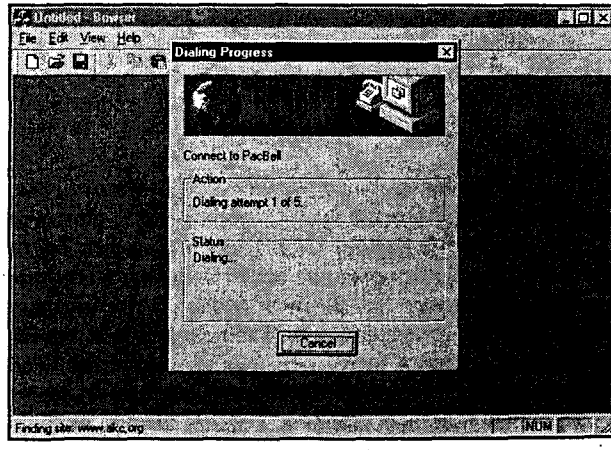


РИСУНОК 21.2.

Дозвон к провайдеру услуг Internet.

Если провайдера нет или процесс подключения прерван, программа выведет сообщение об ошибке и встроенную Web-страницу ошибки перехода. Обратите внимание, что программа имеет строку состояния, которая отображает сообщение при получении файла. Первоначальный вид Bowser можно увидеть на рис. 21.3.

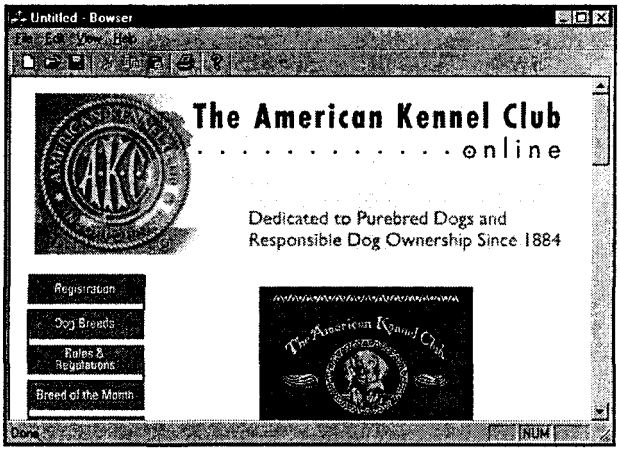


РИСУНОК 21.3.
Просмотр Web с помощью броузера Bowser.

Предоставление броузеру начальной страницы

В Bowser весьма очевидны несколько недостатков. В нем, например, нет кнопок перехода вперед и назад. Вскоре мы решим эту проблему, а сейчас давайте приступим к улучшениям в другой области: стандартной начальной странице. В некотором отношении прекрасно, что Bowser автоматически подключается к Internet — это уменьшает объем требуемой работы. С другой стороны, не совсем учтиво со стороны Bowser связываться по телефону, не спросив на это разрешения, даже если автоматический дозвон установлен. (Вероятно, вы заметите, что подобным образом ведут себя и некоторые разделы справочной системы Visual C++.)

Привлечь пользователей к управлению программой можно, начав ее с отображения модального диалогового окна. Или, что даже лучше, можно просто вывести локальную "начальную" страницу, разработанную специально для Bowser. Система начнет процедуру дозвона к провайдеру, когда Bowser вызовет функцию **Navigate2()**, использующую нелокальный унифицированный локатор ресурсов (URL). Если создать начальную страницу, которую Bowser будет выводить сразу после запуска, система не будет дозваниваться к провайдеру до тех пор, пока пользователь не щелкнет на одной из (отчетливо заметных) гиперссылок.

Начнем с добавления в проект нового HTML-файла и написания кода HTML в редакторе исходного кода. Конечно же, для создания начальной страницы можно воспользоваться редактором WYSIWYG типа Microsoft FrontPage или Netscape Composer. Однако, несмотря на то, что редактор исходного кода Visual C++ не поддерживает принцип WYSIWYG, он понимает HTML, и его возможность выделения синтаксиса выявляет ошибки в коде HTML так же, как и в коде C++.

Кроме того, при открытии нового HTML-файла Visual C++ помещает в него начальные данные.

Исходный код начальной страницы

Для создания новой начальной страницы Bowser выполните следующие шаги:

1. Убедитесь, что в Visual C++ IDE открыт проект Bowser, и выберите из главного меню File | New. В открывшемся диалоговом окне New активной будет вкладка Files. (Если выбрать File | New, когда в среде не открыт проект, активной окажется вкладка Projects.) В списке Files выберите HTML Page, как показано на рис. 21.4. Убедитесь, что отмечен флажок Add To Project, а в выпадающем списке выбран проект Bowser. Назовите файл Bowser.html. Установите подкаталог Debug проекта и нажмите на OK.

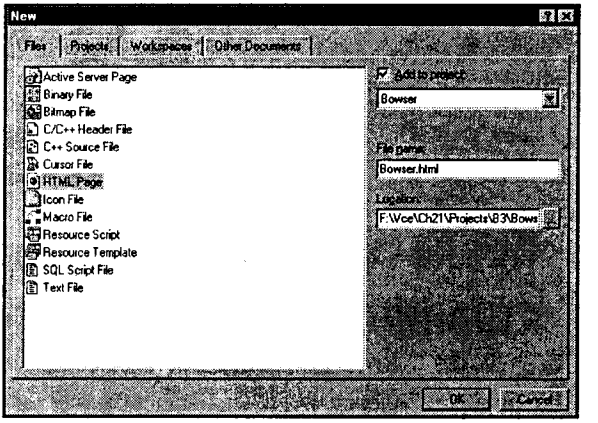


РИСУНОК 21.4.
Добавление в проект новой HTML-страницы.

2. Как показано на рис. 21.5, Visual C++ создает новый HTML-файл, генерирует его базовый каркас и уведомлением "Insert HTML here" ("Вставить здесь код HTML") предоставляет вам полную свободу действий. Поместите в Bowser.html код, приведенный в листинге 21.1 (новые стреки кода в листинге выделены).

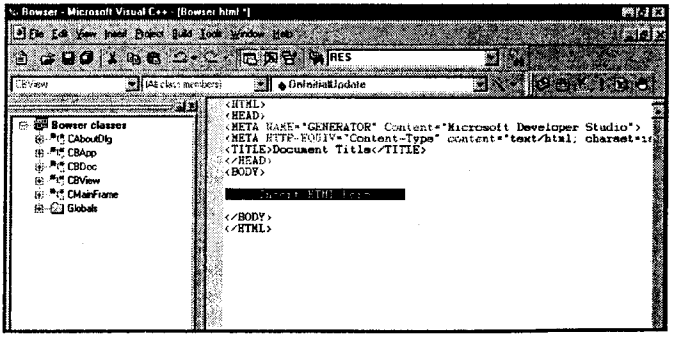


РИСУНОК 21.5.
Visual C++ создал каркас HTML-файла.

Листинг 21.1. Bowser.html.

```
<HTML>
<HEAD>
```

```
<META NAME="GENERATOR" Content="Microsoft Developer Studio">
<META HTTP-EQUIV="Content-Type"
   content="text/html; charset=iso-8859-1">
<TITLE>The Bowser Home Page</TITLE>
</HEAD>
<BODY>
<!-- Insert HTML here -->
<CENTER>
<IMG SRC=Dalmatian.gif ALIGN=LEFT>
<H1>The Bowser Home Page</H1>
<H2>All Dogs - All the Time</H2>
<P>
<HR ALIGN=LEFT>
<H2>Internet Links</H2>
</CENTER>
<H3>
<DL>
<DD><A HREF="http://www.akc.org">
<IMG SRC=./Collie.gif ALIGN=CENTER>
 The American Kennel Club</A>
<DD><A HREF="http://www.tiercom.com/">
<IMG SRC=./Chihuahua.gif ALIGN=CENTER>
 Tiercom Veterinary Information Service</A>
<DD><A HREF="http://www.avma.org/">
<IMG SRC=./Hound.gif ALIGN=CENTER>
 The American Veterinary Medicine Association</A>
<DD><A HREF="http://www.napcc.aspca.org">
<IMG SRC=./Poodle.gif ALIGN=CENTER>
ASPCA - The National Animal Poison Control Center</A>
</DL>
</H3>
</BODY>
</HTML>
```

3. HTML-файл может содержать ссылки на внешние файлы, включая и файлы изображений в форматах JPEG и GIF. Начальная страница Bowser выводит пять GIF-файлов: Dalmation.gif, Collie.gif, Chihuahua.gif, Hound.gif и Poodle.gif. Найдите эти файлы на сопровождающем книгу CD-ROM и скопируйте их в каталог, в котором расположен Bowser.html.

4. При работе, вероятно, потребуется увидеть, как будет выглядеть создаваемая страница во время выполнения. Для этого щелкните правой кнопкой на HTML-файле и выберите из контекстного меню команду Preview (см. рис. 21.6).

5. После выбора Preview Visual C++ запустит IE4 и выведет вашу Web-страницу, как показано на рис. 21.7. Если повторить это действие несколько раз, можно заметить, что Visual C++ не сохраняет HTML-файл перед запуском IE4; поэтому, если хотите учесть последние изменения, сохраните файл самостоятельно. Все файлы вашего проекта Visual C++ сохранит во время его сборки.

Исследование HTML

Если вы уже являетесь гуру в области HTML, можете смело переходить к следующему разделу, где Bowser будет завязываться со своей начальной страни-

цей. Чтобы помочь тем, кто никогда не занимался программированием в HTML, кратко рассмотрим его основы.

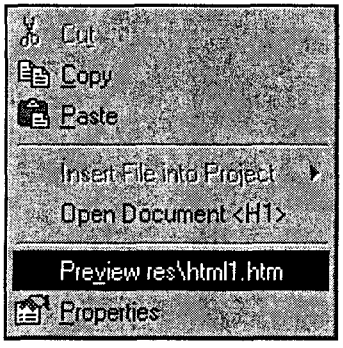


РИСУНОК 21.6. *Предварительный просмотр HTML-файла.*

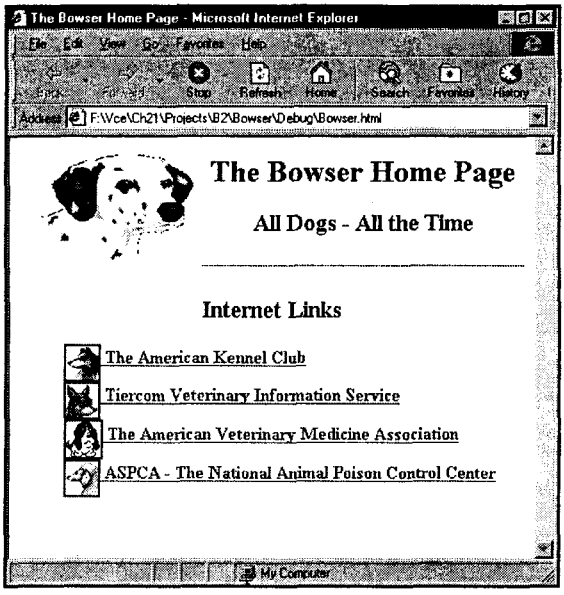


РИСУНОК 21.7. *Просмотр начальной страницы Bowser в IE4.*

HTML — это язык для создания Web-страниц. Web-страницы обычно располагаются на *Web-сервере*, хотя, как вы видели, их можно хранить и на своем локальном диске. Программа, выводящая Web-страницу, называется *клиентом (client)*; наиболее известным клиентом является вездесущий Web-броузер.

Web-сервер посылает Web-страницу типа Bowser.html клиенту (например, IE или Bowser) в виде обычного текста. Ответственность за форматирование страницы в соответствии с инструкциями, содержащимися в ее исходном тексте, полностью возлагается на клиента.

Теги

HTML, как и C++, использует ключевые слова. В контексте HTML ключевые слова называются *тегами (tags)* — их всегда необходимо помещать в угловые скобки. Например, Bowser.html содержит теги **<HTML>**, **<HEAD>**, **<TITLE>** и **<BODY>**. Большинство тегов HTML не зависят от регистра, а пробелы между словами и скобками не играют никакой роли.

Большинство тегов следуют парами. Например, открывающий тег (**<HEAD>**) помечает начало секции, а его визави (**<HEAD>**) — конец. Три строительных, или структурных, тега — **<HTML>**, **<HEAD>** и **<BODY>** — определяют основные части HTML-файла. Теги **<HTML>** обозначают начало и конец файла. Секция **<HEAD>** содержит заголовок и различную мета-информацию, тогда как блок **<BODY>** содержит большую часть видимого текста (за исключением заголовка).

Текст в HTML

Как уже было сказано, клиенты отвечают за форматирование тела текста HTML-документа. Если в HTML-файле содержатся длинные строки текста, броузер должен выровнять его по границам окна. С другой стороны, если строки исходного текста слишком коротки, броузер должен растянуть эти строки, чтобы заполнить окно — нечто, обратное выравниванию слов. Какой бы ни была длина строк, Web-броузер должен выводить их корректно.

Конечно, если весь текст в документе следует вместе, Web-страница будет выглядеть хаотичной смесью. Для обозначения конца строки используется тег **
** (означающий break — останов) или тег **<P>** (Paragraph — абзац). Ни один из них не требует соответствующего закрывающего тега.

Заголовки и стили текста HTML

Кроме обычного текста тела, можно определить шесть уровней строк заголовков: **<H1>** — максимально заметный, а **<H6>** — минимально заметный. Все стили заголовков требуют открывающего и закрывающего тегов.

При использовании тега заголовка неявный конец строки вставляется перед тегом **<H*n*>**, а следующий — за тегом **</H*n*>**. Например, строка HTML

```
<H2>This<H1>IS</H1>Just Fine</H2>
```

приведет к выводу трех строк. Слово "This" появится в одной строке в стиле заголовка **H2**, слово "IS" — в следующей строке в стиле **H1**, а фраза "Just Fine" — в третьей строке, снова в стиле **H2**. Текст, следующий за закрывающим тегом **</H2>**, появится в следующей строке в стиле обычного текста.

Помимо текста тела, не требующего тегов, и стилей заголовка, можно задать стиль **<PRE>** (Preformatted — предварительно отформатированный) для вывода листингов или другого фиксированного текста. Блок заранее отформатированного текста завершается тегом **</PRE>**. В заключение можно воспользоваться стилями символов, чтобы сделать заданный текст полужирным (**<B></B>**), курсивом (**<I></I>**) или подчеркнутым (**<U></U>**).

Изображения в HTML

Изображения на HTML-страницах выводятся с помощью тега **<IMG>**. HTML не требует закрывающего тега **</IMG>**, поскольку **<IMG>** использует *параметр*, который представляет собой ключевое слово, определяющее специфическую информацию, используемую тегом. Тег **<IMG>** требует указать только один параметр — **SRC**. **SRC** определяет URL, содержащий файл изображения в формате JPEG или GIF. URL может ссылаться на файл, расположенный в одном каталоге с HTML-файлом, например:

```
<IMG SRC="SomePicture.GIF">
```

Кроме того, URL может ссылаться на каталог, заданный относительно каталога, содержащего HTML-файл. При задании каталогов используется косая черта — разделитель пути, принятый в Unix (/). (В DOS принят другой разделитель — "\".) Вот пример использования относительного URL:

```
<IMG SRC="../images/AnotherPicture.jpg">
```

В заключение, для задания местонахождения файла изображения можно использовать абсолютный URL. Если URL начинается с / — это абсолютный путь в машине, содержащей HTML-файл. Если URL начинается с *http://*, в нем дол-

жно также быть задано имя узла. Вот пример ссылки на файл изображения с использованием абсолютного URL:

```
<IMG SRC="http://www.nasa.gov/images/jglenn.gif">
```

Конечно, если вы включили ссылку на абсолютный URL (особенно на сервер, управляемый не вами), не стоит удивляться, если файла изображения в этом месте не окажется уже завтра. При загрузке страницы броузер получит рисунок с помощью заданного вами параметра **SRC**.

Кроме **SRC**, тег **<IMG>** использует еще несколько параметров. Можно на ходу изменить масштаб рисунка, либо выровнять его по одному краю, чтобы текст обтекал рисунок.

Гиперссылки

Помимо возможности простого добавления рисунков в Web-страницу, HTML также позволяет включать *гипертекстовые ссылки (hyperlinks)*. Подобно **<IMG>**, гипертекстовая ссылка (или гиперссылка) — это ссылка на другой документ, который может находиться как на том же сервере, так и на другом сервере, находящемся где угодно.

Гиперссылка не вызывает получения броузером документа, на который она ссылается. Вместо этого броузер (обычно) выделяет ссылку особым цветом, указывая, что текст или рисунок — "горячий", или "живой". Если пользователь щелкнет на гиперссылке, Web-броузер попытается получить и вывести документ, на который она ссылается.

Создается гиперссылка с помощью пары тегов **<A></A>** (**A** означает Anchor — привязка). Любой текст или рисунки между тегами становятся частью живого тела гиперссылки. Подобно тегу **<IMG>**, тег **<A>** требует дополнительного параметра **HREF**. **HREF**, как и **SRC**, определяет абсолютный или относительный URL. Но обычно **HREF** указывает на HTML-файл, а не на JPEG- или GIF-файл изображения. Если же **HREF** указывает на упомянутый файл, броузер после щелчка на такой гиперссылке выводит соответствующее изображение.

Эта гиперссылка открывает HTML-страницу, находящуюся в том же каталоге, что и страница, содержащая гиперссылку:

```
<A HREF="Page2.html">Go to page two</A>
```

При отображении этой гиперссылки броузер выводит строку "Go to page two", обычно применяя характерный цвет — пользователю не нужно видеть теги или целевой URL. Связанные файлы не обязательно должны находиться на сервере, содержащем файл с гиперссылкой. Кроме того, URL гиперссылки может задавать файл не полностью: параметр **HREF** может быть файлом, каталогом или Web-сервером. Если URL указывает на каталог или Web-сервер, сервер возвращает стандартный файл, определенный администратором сервера (часто это index.html, default.htm или Default.asp). Следующие аргументы **HREF** являются корректными:

```
<A HREF="http://www.microsoft.com">Microsoft</A>
<A HREF="http://www.microsoft.com/visualc/">Visual C++ page</A>
<A HREF="http://www.microsoft.com/misc/cpyright.htm>Rules</A>
```

Поможем Bowser попасть домой

Теперь, когда вы достаточно знаете HTML, чтобы разобраться в Bowser.html, давайте подключим последний к программе Bowser. Несмотря на то что ваша

начальная страница работает в IE, потребуется заставить Bowser использовать новую страницу вместо указанной ранее. Вы, вероятно, догадались, что изменить необходимо начальное местоположение, используемое в функции **OnInitialUpdate()** — вопрос в том, *что* там поместить?

Если ввести просто имя файла, Bowser подумает, что это имя Web-сервера и попытается к нему подключиться. Если ввести полный путь к файлу Bowser.html, IE будет достаточно осведомлен, чтобы распознать ее как локальный ресурс. Однако жесткое встраивание местонахождения неприменимо, поскольку когда члены клуба любителей собак будут устанавливать Bowser на своих машинах, каждый из них может поместить программу в другой каталог.

Решить эту проблему можно, сохранив HTML-файл в одном каталоге с выполняемым файлом. Воспользуйтесь глобальной переменной **__argv[0]**, чтобы получить полный путь к выполняемому файлу программы, и функцией библиотеки C Runtime Library (RTL) для извлечения буквы диска и пути. Далее в конец строки следует доставить имя файла, т.е. Bowser.html, и вызвать функцию **Navigate2()**, как это делалось и раньше.

В листинге 21.2 показан соответствующий код.

Листинг 21.2. Функция CBView::OnInitialUpdate().

```
void CBView::OnInitialUpdate()
{
      CHtmlView::OnInitialUpdate();

      char drive[_MAX_PATH];
      char dir[_MAX_DIR];
      char fname[_MAX_FNAME];
      char ext[_MAX_EXT];

      _splitpath(__argv[0], drive, dir, fname, ext);

      CString path(drive);
      path += dir;
      path += "Bowser.html";

      Navigate2(path,NULL,NULL);
}
```

Где, вы говорите, это было?

Оперативная справка по глобальной переменной __argv может в какой-то мере огорчить. Эта возможность не является документированной — она скрыта. __argv рассматривается в одной из статей Knowledge Base — "HOWTO: Obtain the Program Name in a Windows-Based Application" (Article ID: Q126571). Индексы и другая документация не предоставляют дополнительной информации, хотя, как минимум, 10 примеров программ используют__argv. Чтобы найти единственную статью, введите в окне Search строку "__argv", не забыв набрать два символа подчеркивания.

Если вы думаете, что документация довольно скудна, потому что вы определяете путь к программе другим способом, то частично будете правы. Вместо этого для получения командной строки можно вызвать функцию Windows API GetCommandLine(). GetCommandLine() возвращает параметр программы в кавычках. Перед вызовом _splitpath() инкрементируйте указатель на строку, чтобы пропустить символ кавычки.

Новинка: ресурсы HTML

В состав пакета поставки приложения Bowser потребуется включить дополнительные файлы изображений, а также главную начальную HTML-страницу. Неплохо все это поместить в выполняемый модуль, подобно панелям инструментов и пиктограммам. Теперь это доступно, благодаря *ресурсам* HTML — еще одной новой возможности Visual C++ 6.0.

Ресурсы HTML позволяют включать HTML-страницы, а также и необходимые рисунки, в выполняемый модуль. Допускается включать произвольное количество страниц, причем они могут быть связанными между собой. Кроме того, в выполняемый модуль можно поместить ресурсы изображений, используемых HTML-страницами.

Однако, прежде чем отказаться от обычных старых добрых HTML-страниц, рассмотрим преимущества и недостатки каждого подхода. Если связать ресурсы HTML с выполняемым модулем, придется компилировать и поставлять новый EXE-файл при каждом изменении связей. В случае же хранения отдельных HTML-файлов можно просто поставлять обновленные HTML-страницы. С другой стороны, если материалы относительно постоянны, ресурсы HTML значительно упростят распространение приложения.

Импортирование файла Bowser.html

Чтобы Bowser смог использовать встроенный HTML-файл (используя уже созданный вами файл), выполните следующие шаги:

1. Откройте проект Bowser и выберите из главного меню Insert|Resource (либо нажмите Ctrl+R). Когда откроется диалоговое окно Insert Resource, выберите HTML, как показано на рис. 21.8, и нажмите на Import.

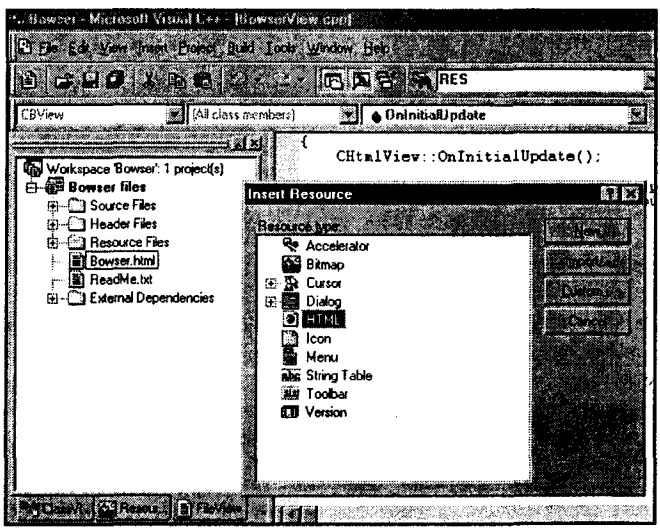


РИСУНОК 21.8.
Вставка Bowser.html как ресурса HTML.

2. В результате откроется диалоговое окно *Import* Resource (*Импортирование* ресурса) — не путайте его с только что оставленным вами диалоговым окном *Insert* Resource (*Вставка* ресурса). Отыщите файл Bowser.html, выбрав сперт-

ва HTML Files в выпадающем списке Files Of Type. Убедитесь, что в списке Open As выбрано значение Auto. Когда диалоговое окно будет выглядеть, как показано на рис. 21.9, нажмите на Import.

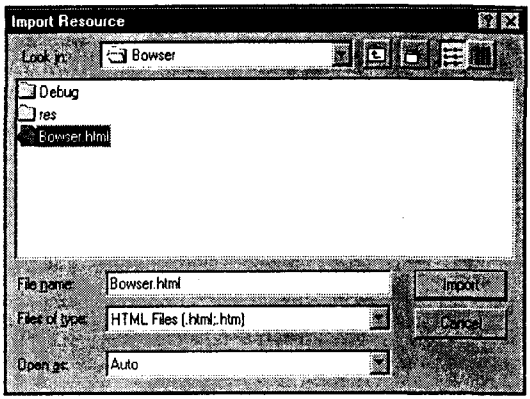


РИСУНОК 21.9.
Импортирование Bowser.html как ресурса HTML.

3. В окне Workspace переключитесь на панель ResourceView и обратите внимание на новую папку HTML. Откройте папку для отображения **IDR_HTML1**. Щелкните правой кнопкой на **IDR_HTML1**, чтобы вывести окно свойств Custom Resource Properties, а затем установите идентификатор ресурса в **IDR_HOME_PAGE**. Когда окно свойств будет выглядеть подобно показанному на рис. 21.10, закройте его.

Это все, что необходимо для добавления нового ресурса HTML. Осталось только включить все графические файлы. Для этого придется потрудиться чуть больше.

Импортирование графических файлов

Из раздела, посвященного HTML, вы узнали, что Web-страницы могут отображать только графические файлы форматов JPEG и GIF. К сожалению, ресурсы изображений Windows не являются рисунками JPEG или GIF. Вы потерпели бы неудачу, если бы не два следующих факта:

- Visual C++ позволяет открывать файлы GIF и JPEG и записывать их содержимое в виде BMP-файлов. Следовательно, можно запросто преобразовать все рисунки в формат BMP.
- Элемент управления IE Web Browser Control поддерживает специальный протокол **res** — протокол для вывода Web-содержимого, сохраненного в виде ресурса в выполняемом файле.

Для привязки рисунков к файлу Bowser.html необходимо выполнить две задачи: преобразовать все рисунки в BMP-файлы, а затем отредактировать Bowser.html, чтобы он ссылался на новые ресурсы, а не на оригинальные GIF-файлы.

Вот соответствующие шаги:

1. Скопируйте все GIF-файлы изображений в подкаталог res вашего проекта. Здесь уже находится изображение панели инструментов приложения, а также пиктограмма документа. Откройте в редакторе изображений (Bitmap Editor) все GIF-файлы и сохраните их как файлы .bmp (см. рис. 21.11).
2. Как только будут преобразованы все рисунки, выберите Insert | Resource (или нажмите Ctrl+R), чтобы открыть диалоговое окно Insert Resource. В качестве

типа ресурса выберите Bitmap и нажмите на Import. Перейдите в каталог res и загрузите все изображения. После загрузки каждого рисунка установите его идентификатор ресурса в соответствии с именем файла рисунка. На рис. 21.12 показан процесс загрузки последнего рисунка.

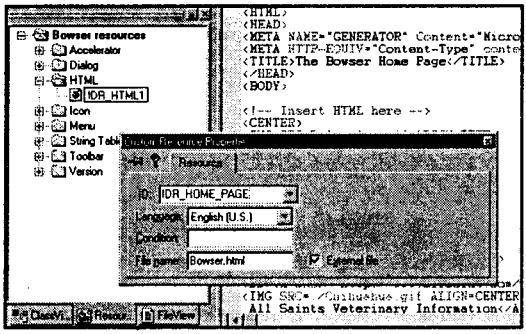


РИСУНОК 21.10. *Установка свойств HTML для IDR_HOME_PAGE.*

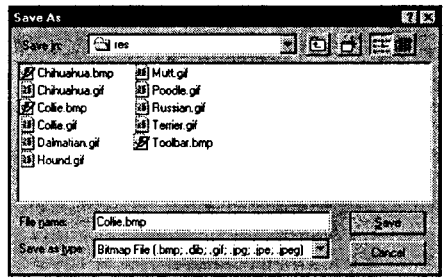


РИСУНОК 21.11. *Преобразование GIF-файлов в формат BMP.*

Использование протокола res:

После загрузки рисунков в виде растровых изображений параметр **SRC**, используемый в Bowser.html, станет некорректным. Теперь каждый параметр потребуется изменить так, чтобы он указывал на новый рисунок с использованием протокола **res:**.

При описании файла с помощью протокола **res**: необходимо ответить на следующие вопросы:

- *Где находится ресурс?* В данном случае он находится в Bowser.exe. При желании можно обратиться к растровым изображениям, расположенным вне выполняемого модуля.
- *Какой тип ресурса загружается?* Стандартные значения заданы в файле Winuser.h. Если хотите увидеть их, поищите ***RT_***. Вам необходимы два типа ресурса — **RT_BITMAP** и **RT_HTML**, имеющие значения, соответственно, 2 и 23. Для использования одного из этих значений в теге протокола **res:** введите его числовой эквивалент после символа # — символическое имя использовать нельзя.
- *Какой именно ресурс требуется загрузить?* Как и в протоколе типа ресурса, после символа # вводится численное значение — нет возможности воспользоваться символическим именем, наподобие **IDR_POODLE**.

Рассмотрим шаги, необходимые для изменения и подключения Bowser.html.

1. Выберите из главного меню View|Resource Symbols и впишите действительные значения идентификаторов для каждого ресурса растрового изображения. На рис. 21.13 показаны числа, введенные для текущего проекта.
2. В редакторе ресурсов откройте Bowser.html, и каждое вхождение

```
SRC="somefile.gif"
```

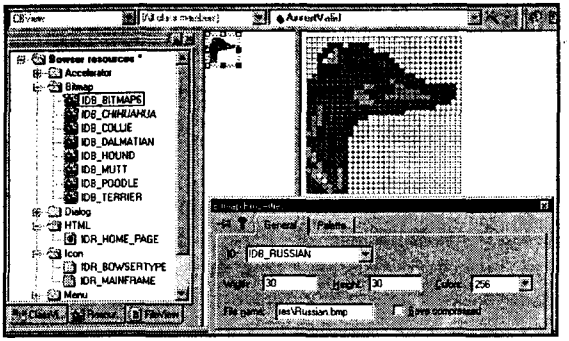


РИСУНОК 21.12. *Загрузка изображений.*

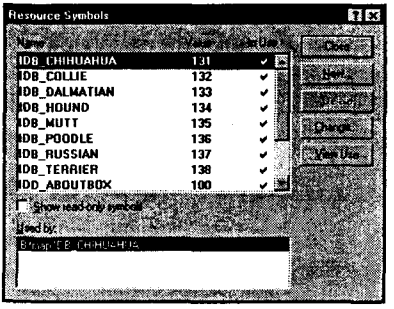


РИСУНОК 21.13. *Просмотр идентификаторов ресурса.*

замените на

```
SRC="res://Bowser.exe/#2/#nnn"
```

где ***nnn*** — идентификатор ресурса для рисунка.

3. Отыщите функцию **CBView::OnInitialUpdate()**. Приведите ее в соответствие с листингом 21.3.

Листинг 21.3. Функция OnInitialUpdate().

```
void CBView::OnInitialUpdate()
{
      CHtmlView::OnInitialUpdate();
      LoadFromResource(IDR_HOME_PAGE);
}
```

4. Откомпилируйте и запустите программу. Теперь она работает везде без необходимости переноса вместе с ней HTML-файла и файлов изображений. Кроме того, она ведет себя вполне прилично, не пытаясь дозваниваться до провайдера, пока не будет выполнен щелчок на гиперссылке Internet.

Исследование навигации

Теперь, когда Bowser выдрессирован чуть лучше, наступило время обратить внимание на меню и панели инструментов программы. В текущем состоянии Bowser позволяет вернуться на предыдущую страницу только при помощи гиперссылки на следующей странице. Поскольку начальная страница Bowser является встроенным ресурсом, ни одна из страниц в мире на нее не ссылается. Другими словами, покинув свою начальную страницу, вы уже никогда не сможете вернуться к ней. Итак, нашим следующим усовершенствованием будет кнопка Back (Назад), возвращающая Bowser к предыдущему посещенному им URL.

Естественно, после использования кнопки Back может возникнуть желание изменить свое решение и вернуться к оригинальной странице. В этой связи будет также добавлена кнопка Forward (Вперед). Кроме того, в процессе работы добавляется и три других кнопки:

- *Кнопка Stop* — Позволяет прервать загрузку, продолжающуюся слишком долго.
- *Кнопка Web* — Выводит диалоговое окно, в котором можно ввести URL.
- *Кнопка Search* — Автоматически открывает поисковый сервер.

Помимо прочего, обеспечится работоспособность диалогового окна File Open.

Может показаться, что такая работа потребует значительных усилий, но как оказывается, наиболее трудной частью будет создание кнопок в панели инструментов. Когда придет время написания кода, вы увидите, что класс **CHtmlView** уже имеет функции, реализующие большинство необходимых вам возможностей.

Создание панели инструментов

Сперва следует создать панель инструментов и пункты меню, необходимые для доступа к каждой из команд. Дабы не рисовать рисунки для кнопок панели инструментов от руки, воспользуйтесь графическими файлами, поставляемыми вместе с Visual C++. Необходимые файлы находятся в подкаталоге Common\Graphics\Icons каталога Visual Studio. (Если при инсталляции Visual C++ графические файлы не устанавливались, прочитайте их непосредственно с установочного CD-ROM, опять-таки из подкаталога Common\Graphics\Icons.)

Для создания новых пунктов меню и панели инструментов выполните следующие шаги:

1. Убедитесь, что открыт проект Bowser. Выберите в окне Workspace панель ResourceView, разверните папку Toolbars и дважды щелкните на ресурсе **IDR_MAINFRAME**. Удалите кнопки New, Cut, Copy и Paste, оставив только Open, Save, Print и Help.
2. Выберите из главного меню File | Open и перейдите в подкаталог Visual Studio Common\Graphics\Icons. В выпадающем списке Files Of Type диалогового окна Open выберите Image Files. В подкаталоге Arrows выберите файлы Arw04lt.ico и Arw04rt.ico, как показано на рис. 21.14; нажмите на Open. Аналогично загрузите файл W95mbx01.ico (белый крестик в красном кругу) из подкаталога Computer. Откройте Earth.ico из подкаталога Elements и Binoculr.ico из подкаталога Misc.
3. Эти пиктограммы в проект не добавляются — они используются в качестве рисунков для кнопок панели инструментов. Каждый файл .ico содержит две версии одной пиктограммы — пиктограмму стандартного размера 32 × 32 (пиксела) и малую пиктограмму форматом 16 × 16. Хотя кнопки панели инструментов обычно имеют размеры 16 × 15, пиктограммы придется немного увеличить, дабы приспособить их к имеющимся рисункам. Используя меню Window, выведите каждую из пяти пиктограмм и выберите их уменьшенные версии (см. рис. 21.15).

Создание кнопок панели инструментов

Теперь создайте новые кнопки панели инструментов для каждого изображения пиктограммы. Для каждого из пяти новых изображений выполните следующие шаги:

1. Используя меню Window, выберите окно, содержащее одну из пиктограмм. Щелкните на левой панели, тем самым выбрав изображение с реальными размерами. Нажмите Ctrl+C (или выберите из главного меню Edit | Copy), чтобы скопировать изображение пиктограммы в буфер обмена.
2. Загрузите панель инструментов в редактор. Для этого в окне ResourceView дважды щелкните на идентификаторе ресурса панели инструментов **IDR_MAINFRAME**, либо выберите в меню Window окно панели инструмен-

тов. Создайте новую кнопку панели инструментов и активизируйте ее. Кнопка с реальными размерами и ее увеличенное изображение должны появиться в виде сплошных серых пустых прямоугольников. Выберите из главного меню Edit | Paste (или нажмите Ctrl+V) для вставки изображения пиктограммы в новую кнопку панели инструментов. При первой вставке Visual C++ предупредит, что изображение слишком велико и предложит увеличить формат кнопки панели инструментов. Нажмите на OK, тем самым увеличив все кнопки панели инструментов до формата 16 × 16.

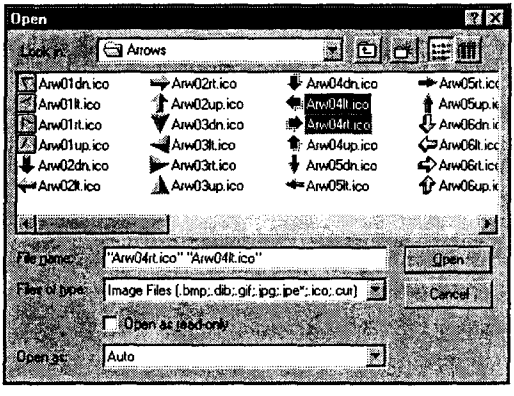


РИСУНОК 21.14. *Получение графических файлов из установочного CD-ROM Visual C++.*

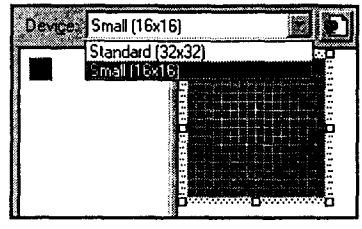


РИСУНОК 21.15. *Выбор пиктограмм малого формата.*

3. Повторяйте шаги 1 и 2, пока не получите пять новых кнопок панели инструментов, показанных на рис. 21.16. Идентификатор ресурса и строку подсказки для каждой кнопки установите в соответствии с информацией, приведенной в табл. 21.1.

Таблица 21.1. Кнопки панели инструментов Bowser.

Изображение	*Идентификатор ресурса*	*Строка подсказки*
Arw04lt.ico	ID_WEB_BACK	Вернуться к предыдущей Web-странице\nНазад
Arw04rt.ico	ID_WEB_FORWARD	Перейти на следующую Web-страницу\nВперед
W95mbx01.ico	ID_WEB_STOP	Остановить загрузку текущей Web-страницы\nСтоп
Earth.ico	ID_WEB_OPEN	Открыть Web-сайт\nОткрыть сайт
Binoculr.ico	ID_WEB_SEARCH	Искать в Web\nПоиск

Создание пунктов меню

С каждой кнопкой панели инструментов потребуется связать команду меню, чтобы члены клуба, не имеющие мыши, также могли пользоваться упомянутыми командами. (Многим членам клуба мышь напоминает об отвратительных котах.) Выполните следующие шаги:

1. Добавьте в меню File пункт Open Web Site сразу после пункта Open, установив идентификатор ресурса в **ID_WEB_OPEN**. Удалите из меню File пункт Save. В конечном итоге меню должно выглядеть, как показано на рис. 21.17.

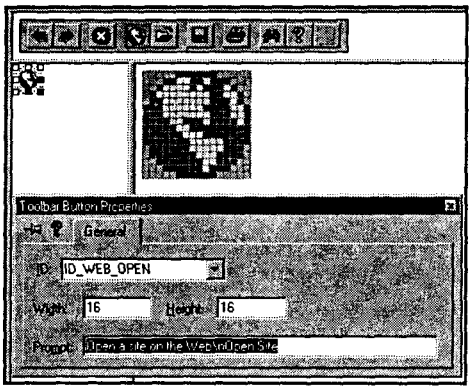


РИСУНОК 21.16. *Кнопки панели инструментов Bowser.*

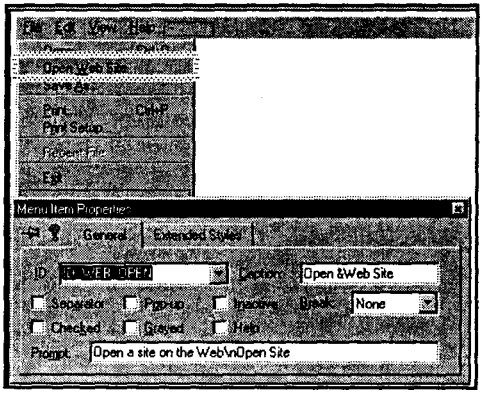


РИСУНОК 21.17. *Добавление новой команды меню Open Web Site.*

2. Удалите меню Edit, выбрав его и нажав Delete. Нажмите на OK, когда Visual C++ запросит подтверждение удаления.
3. Добавьте новое меню Navigate и поместите в него пункты Back, Forward и Stop, как показано на рис. 21.18. Убедитесь, что для них используются те же идентификаторы ресурса, что и для кнопок панели инструментов.

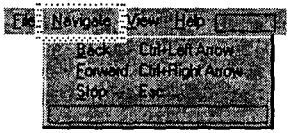


РИСУНОК 21.18. *Добавление меню Navigate.*

4. В меню Help добавьте команду **ID_WEB_SEARCH**, сразу за пунктом меню About. Установите ее заголовок в "&Search the Web".
5. Откройте таблицу акселераторов **IDR_MAINFRAME**, а затем при помощи возможности Use Next Key Typed добавьте новые клавиатурные акселераторы для трех пунктов меню Navigate. Для кнопки Forward нажмите Ctrl+стрелка вправо, для Back — Ctrl+стрелка влево, а для кнопки Stop — Esc.

Добавление кода

Вместо обычных маленьких кнопок, давайте создадим кнопки побольше, в которые поместим и текст, и пиктограмму. Сделать это можно с помощью функции **CToolBar::SetButtonText()**. **SetButtonText()** принимает два параметра: смещение для кнопки (0 соответствует левому краю) и выводимую строку.

При первом использовании **SetButtonText()** может вызвать удивление, что текст и кнопки не совпадают. Это происходит в связи с тем, что каждый разделитель в панели инструментов (например, промежуток между кнопками Stop и Forward) учитывается как обычная кнопка.

Вот код, который необходимо добавить в функцию **OnCreate()** класса **CMainFrame** сразу перед закреплением панели инструментов:

```
m_wndToolBar.SetButtonText( 0,  "Back");
m_wndToolBar.SetButtonText( 1,  "Forward");
m_wndToolBar.SetButtonText( 3,  "Stop");
m_wndToolBar.SetButtonText( 5,  "Web");
m_wndToolBar.SetButtonText( 6,  "File");
m_wndToolBar.SetButtonText( 8,  "Save");
m_wndToolBar.SetButtonText(10,  "Print");
m_wndToolBar.SetButtonText(12,  "Search");
m_wndToolBar.SetButtonText(13,  "About");
```

Когда на кнопках будет выводиться текст, уделите внимание еще одной небольшой детали: с помощью функции **SetSizes()** измените их размеры. **SetSizes()** принимает в качестве параметров два объекта **CSize**, где первый задает полные размеры кнопки, а второй — размеры изображения. Поместите этот код непосредственно после множества вызовов **SetButtonText()**:

```
m_wndToolBar.SetSizes(CSize(56,  40), CSize(16,16));
```

Обработка CBView

Теперь, когда работа над интерфейсом панели инструментов завершена, давайте подключим код для кнопок. Вот как это сделать:

1. *С помощью ClassWizard добавьте обработчики сообщения* **COMMAND** *для идентификаторов ресурсов* **ID_WEB**. Убедитесь, что обработчики добавляются в класс **CBView**, а не в **CMainFrame**. Окно ClassWizard по завершении должно выглядеть, как на рис. 21.19.

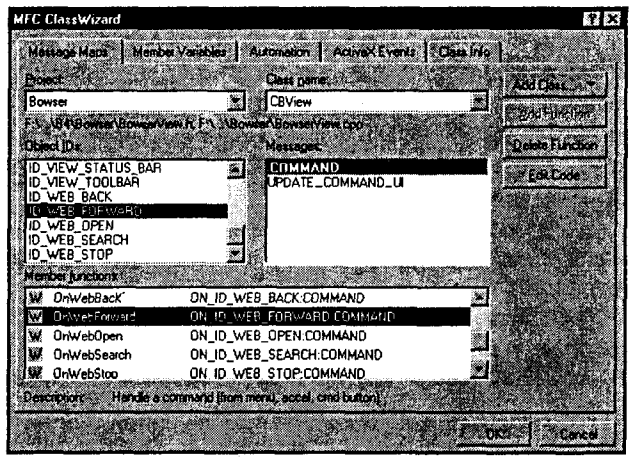


РИСУНОК 21.19.
Добавление обработчиков сообщений COMMAND для Bowser.

2. В открытом ClassWizard выберите из списка Object IDs класс **CBView**, а из списка Messages — **OnDocumentComplete**. Щелкните на Add Function для перекрытия этой виртуальной функции класса **CHtmlView**. Окно ClassWizard показано на рис. 21.20.
3. Добавьте код из листинга 21.4 в функции, сгенерированные ClassWizard для класса **CBView**. *Обратите внимание, что код не включает функции* **OnWebOpen()** — мы обратимся к ней позже.

Листинг 21.4. Обработка команд Bowser в CBView.

```
void CBView::OnWebBack()        {       GoBack();       }
void CBView::OnWebForward()     {       GoForward();    }
void CBView::OnWebSearch()      {       GoSearch();     }
void CBView::OnWebStop()        {       Stop();         }
void CBView::OnDocumentComplete(LPCTSTR lpszURL)
{
      CHtmlView::OnDocumentComplete(lpszURL);
      AfxGetMainWnd()->SetWindowText(GetLocationName());
}
```

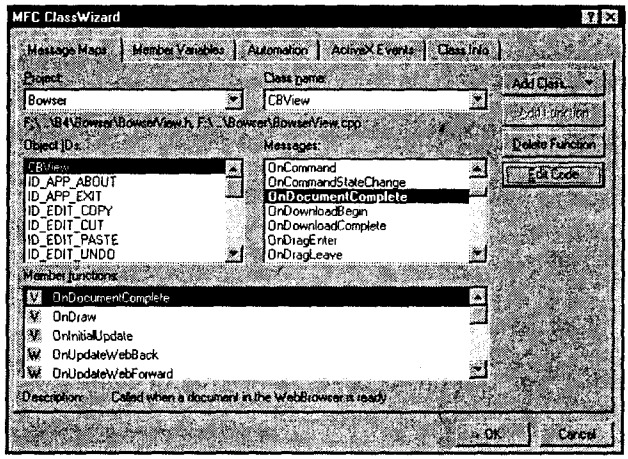


РИСУНОК 21.20.
Добавление в класс CBView виртуальной функции OnDocumentComplete().

Каждая функция **CBView** просто обращается к соответствующей функции класса **CHtmlView**. **OnDocumentComplete()** вызывается MFC при завершении загрузки документа. Вызов **OnDocumentComplete()** приводит к передаче в объект **CHtmlView** URL загруженной страницы. Если требуется создать список хронологии, воспользуйтесь параметром **lpszURL**.

В данном случае после загрузки страницы необходимо просто установить заголовок главного окна Bowser в значение заголовка текущей HTML-страницы. Это делается во второй строке **OnDocumentComplete()** за счет обращения к функции **GetLocationName()** класса **CHtmlView**.

Обработка диалоговых окон открытия

Перед завершением работы над приложением следует реализовать еще две вещи. Во-первых, несмотря на то что кнопка File Open работает, она не загружает HTML-файл. Во-вторых, потребуется создать диалоговое окно для открытия URL.

На самом деле хорошо, что File Open не работает, поскольку это дает возможность исправить неудобную "возможность" IE. При каждом открытии в IE локального HTML-файла, к месту расположения файла приходится двигаться, начиная с корневого каталога. Если необходимо открыть несколько файлов из одного каталога, лишние переходы могут вызвать раздражение. Ваше усовершенствованное диалоговое окно File Open будет сохранять позицию, где последний раз открывался файл.

Сначала давайте возьмемся за решение проблемы File Open. Для этого выполните следующие шаги:

1. Вместо создания нового объекта **CFileDialog** при каждом открытии файла, класс представления будет содержать один объект **CFileDialog** как один из своих элементов данных. В окне ClassView отыщите класс **CBView** и щелкните на нам правой кнопкой мыши для вызова контекстного меню. Выберите из меню Add Member Variable и создайте переменную **private CFileDialog m_FileDialog**.
2. Поскольку **CFileDialog** должен инициализироваться в своем конструкторе, при вызове конструктора **CFileDialog** потребуется воспользоваться синтаксисом инициализации. Найдите конструктор **CBView** и добавьте в него выделенные строки из листинга 21.5. (Не забудьте поставить двоеточие перед **m_FileDialog**.)

Листинг 21.5. Инициализация m_FileDialog в конструкторе CBView.

```
CBView::CBView()
: m_FileDialog(TRUE, "*.html", "*.html",
                     OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT,
                     "HTML Files (*.html)|*.htm;*.html||")
{
      // ЧТО СДЕЛАТЬ: поместить здесь код конструктора
}
```

3. С помощью ClassWizard добавьте в класс **CBView** функцию обработки для идентификатора **ID_FILE_OPEN** (назовите ее **OnFileOpen()**). Поместите в нее код из листинга 21.6, обеспечивающий вывод диалогового окна File Open и переход на новую страницу в случае нажатия на OK.

Листинг 21.6. Функция CBView::OnFileOpen.

```
void CBView::OnFileOpen()
{
      if (m_FileDialog.DoModal() == IDOK)
            Navigate2(m_FileDialog.GetPathName(), NULL, NULL);
}
```

Диалоговое окно Open A Web Site

Диалоговое окно Open A Web Site в Visual C++ не встроено (что довольно удивительно, если принять во внимание такую направленность на Internet). К счастью, построение такого окна сложностей не вызывает.

Вот что вам для этого необходимо сделать:

1. В главном меню выберите Insert|Resource или нажмите Ctrl+R. Выберите Dialog в списке Insert Resource и нажмите на New.
2. Озаглавьте новое диалоговое окно как "Open A Web Site" и установите идентификатор ресурса в **IDD_OPEN_WEB**. Добавьте в диалоговое окно элемент статического текста и элемент редактирования. Идентификатор ресурса элемента редактирования установите в **IDC_URL**, а заголовок статического текста — в "URL". Расположите элементы управления, как показано на рис. 21.21.
3. Перейдите в ClassWizard и нажмите на OK, когда Visual C++ предложит создать новый класс. Назовите класс **CURLDlg**. Родительским классом должен быть CDialog, как показано на рис. 21.22. Нажмите на OK.

4. Здесь же, в ClassWizard, убедитесь, что выбран новый класс **CURLDlg** и переключитесь на вкладку Member Variables. Выберите **IDC_URL** и нажмите на Add Variable. В диалоговом окне Add Member Variable назовите переменную **m_URL**, из списка Category выберите Value, а из списка Variable Type — **CString**.

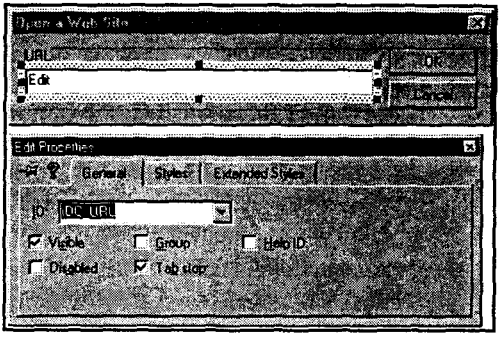


РИСУНОК 21.21. *Создание нового диалогового окна Open A Web Site.*

5. В ClassWizard переключитесь на вкладку Message Maps. В выпадающем списке Class Name еще раз выберите класс **CBView**, отыщите его функцию **OnWebOpen()** и нажмите на Edit Code. Добавьте код из листинга 21.7 к коду, сгенерированному ClassWizard.

Листинг 21.7. Функция CBView::OnWebOpen().

```
void CBView::OnWebOpen()
{
    CURLDlg dlg;
    if (dlg.DoModal() == IDOK)
        Navigate2(dlg.m_URL, NULL, NULL);
}
```

6. Перейдите в начало открытого файла BowserView.cpp и добавьте следующую строку, чтобы класс **CBView** знал, чем является объект **CURLDlg**:

```
#include "URLDlg.h"
```

По завершении откомпилируйте программу и выведите Bowser на прогулку. Как видно из рис. 21.24, броузер выглядит как магазин со множеством товаров. Нет, даже лучше — если вам не нравится, как что-либо работает, вы можете это изменить!

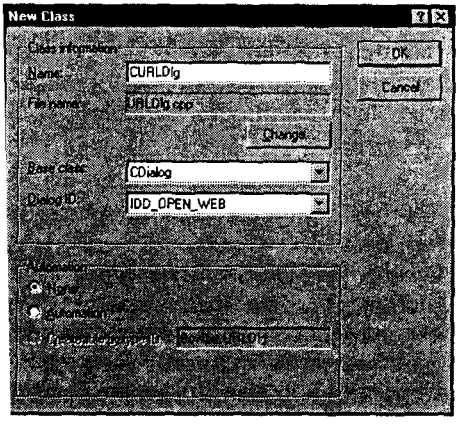


РИСУНОК 21.22. *Добавление класса CURLDlg.*

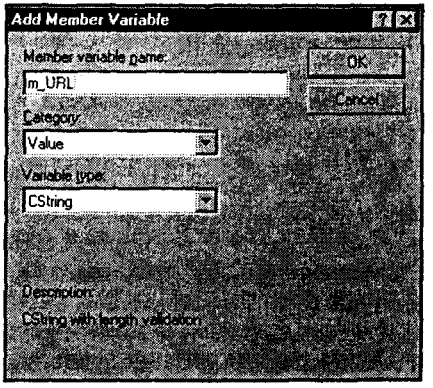


РИСУНОК 21.23. *Добавление в класс CURLDlg переменной m_URL.*

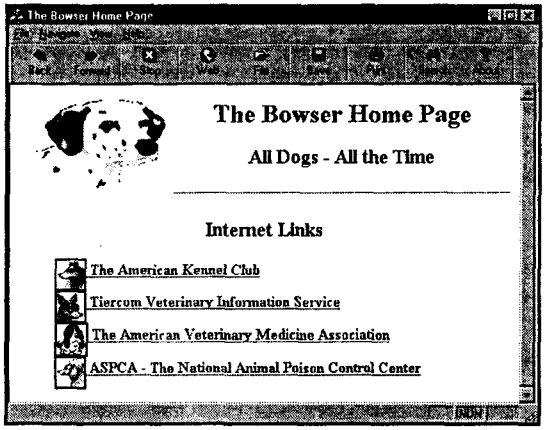


РИСУНОК 21.24.
Броузер Bowser в работе.

Использование классов WinInet

Если вы читали документацию по классу **CHtmlView** и смотрели примеры, то несомненно заметили, что при использовании Bowser (либо примера MFCIE из CD-ROM с Visual C++) для сохранения HTML-файла результат чуть-чуть не дотягивал до желаемого. Если нажать на Save As, то Bowser создаст файл, только *пустой*. Не нужно далеко идти, чтобы понять, почему так происходит. Давайте взглянем на метод **CBDoc::Serialize()**: в нем ничего нету! Очевидно, что класс **CHtmlView**, т.е. элемент управления IE Web Browser Control, загружает HTML-код страницы. В конце концов, Web-страница отображается. По всей видимости, существует какой-то способ обратиться к элементу управления IE, чтобы получить выводимый им код и сохранить его на диске.

К сожалению, хоть это и кажется простым (особенно когда вы увидите, что в классе **CHtmlView** есть метод **GetHtmlDocument()**), вы быстро окажетесь в тупике. Функция **GetHtmlDocument()** не возвращает *текстового* представления страницы — вместо этого она возвращает указатель на документ ActiveX **IDispatch**. Чтобы работать с документом, потребуется вызвать функцию OLE **QueryInterface()**, чтобы получить указатель на интерфейс **IHtmlDocument2**. И после всего этого, работа *только начинает* усложняться...

На самом деле вам не нужен документ в смысле "документ/представление" ActiveX — необходим всего лишь обычный текстовый документ, хранящийся на сервере. Оказывается, что этот текст получить легко, даже не полагаясь на установленный на машине IE. Можно просто воспользоваться классами WinInet (Windows Internet Extension) — Internet-расширение Windows.

Что такое WinInet?

Предположим, необходимо отправить своему другу файл через Internet. Каким образом файл с вашего компьютера попадает на компьютер вашего друга? Очень просто, ваш компьютер разбивает файл на "кусочки", каждый из которых содержит Internet-адрес вашего компьютера и Internet-адрес компьютера вашего друга (плюс дополнительную информацию).

Ваш компьютер передает эти кусочки провайдеру, который, в свою очередь, передает их провайдеру вашего друга (скорее всего, через каких-то промежуточных провайдеров). В заключение провайдер отправит эти кусочки на компьютер вашего друга, который соберет из них файл. Волшебство, управляющее всеми этими разбивками, сборками, отправкой и получением, называется *протоколом*. Internet использует протокол, который носит название *протокол управления передачей/протокол Internet (Transmission Control Protocol/Internet Protocol*, или *TCP/IP)*.

Хотя программировать на чистом TCP/IP и допускается, это влечет за собой массу деталей. Так, большинство программистов работают на самом высоком уровне, с *сокетами* (*sockets*) — логическими соединениями между двумя компьютерами в сети. Один из компьютеров выступает в качестве *источника* данных, а другой — в качестве *стока*, или приемника, данных. Если вы ранее использовали перенаправление файлов, то знаете что программа консольного режима может получать ввод из файла или клавиатуры, но таким же образом можно использовать и источники данных. Сетевые сокеты работают подобным образом: вы открываете сокет, пишете или читаете из него, а затем его закрываете. Для использования сокетов в программах воспользуйтесь интерфейсом Winsock (либо Winsock2).

Программирование сокетов намного проще непосредственного программирования TCP/IP, однако оно все равно связано с массой тонкостей. Классы MFC WinInet инкапсулируют большинство повторяющейся работы, требуемой для установки сокетного соединения между двумя компьютерами и отслеживания различных ошибок, которые могут произойти во время обмена данными. При помощи классов WinInet программирование сокетов настоль же просто, как чтение и запись в локальные файлы.

Простой WinInet

Хотите — верьте, хотите — нет, но простое приложение Internet-клиента можно создать всего за 4 простых шага:

- Создать объект **CInternetSession**, который подключается к провайдеру, если вы еще к нему не подключены.
- Создать указатель на **CStdioFile** и инициализировать его, передав строковое представление URL в функцию **OpenURL()** класса **CInternetSession**. Возвращаемый указатель является специализированной формой **CStdioFile** или **CInternetFile**, но обычно об этом не беспокоятся; воспринимайте его как обычный файл.
- Прочитать из файла при помощи **Read()** или **ReadString()**.
- Закрыть файл и сеанс Internet.

Вы все еще не верите, что программирование для Internet оказывается таким простым? Чтобы убедить вас, давайте создадим простой Web-броузер, который загружает и выводит текст любой HTML-страницы из любого уголка мира. На все это потратится каких-нибудь 10 минут, при этом даже не будут использоваться какие-то элементы управления ActiveX.

Выполните следующие шаги:

1. С помощью AppWizard создайте SDI-приложение с именем SimpleNet без поддержки элементов управления ActiveX и основанное на классе **CEditView**.

2. Воспользуйтесь Insert|Resource для добавления нового диалогового окна. Поместите в него элемент редактирования и элемент статического текста. Идентификатор ресурса элемента редактирования установите в **IDC_URL**; расположите и установите заголовки элементов управления так, как показано на рис. 21.25.

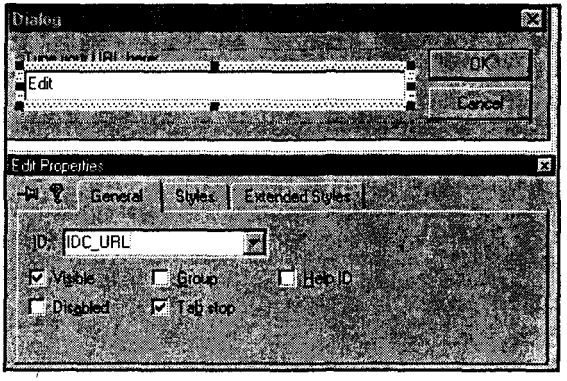


РИСУНОК 21.25.
Добавление диалогового окна Open URL.

3. Создайте в ClassWizard для вашего диалогового окна новый класс **CURLDlg**, как это делалось в Bowser. Здесь же, в ClassWizard, создайте переменную **CString m_URL** для элемента управления редактированием.
4. Откройте SampleNet.cpp и найдите в карте сообщений записи для File Open и File New. Закомментируйте обе эти строки. Они должны выглядеть так:

```
//      ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
//      ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
```

5. При помощи ClassWizard создайте в классе **CSimpleNetView** обработчик **COMMAND** для **ID_FILE_NEW**. Назовите обработчик **OnFileNew()** и установите его же для **ID_FILE_OPEN**. В **OnFileNew()** поместите код, показанный в листинге 21.8.

Листинг 21.8. Обработчик сообщений CSimpleNetView::OnFileNew().

```
void CSimpleNetView::OnFileNew()
{
      // ЧТО СДЕЛАТЬ: Поместить здесь код обработчика
      CURLDlg dlg;
      if (dlg.DoModal() == IDOK)
      {
            CWaitCursor wait;
            CInternetSession session("Simple Net");
            CStdioFile * pFile = NULL;
            pFile = session.OpenURL(dlg.m_URL);
            if (pFile != NULL)
            {
                  SetWindowText("");
                  CString str, allText, crlf = "\r\n";
                  while (pFile->ReadString(str))
                  {
                        allText += crlf + str;
                  }
                  SetWindowText(allText);
```

```
                pFile->Close();
            }
            session.Close();
        }
}
```

6. Пока открыт SimpleNet.cpp, добавьте в начало файла две следующих строки, чтобы могли распознаваться классы WinInet и класс **CURLDlg**:

```
#include "URLDlg.h"
#include <afxinet.h>
```

7. Отыщите функцию **OnPreCreateWindow()** и закомментируйте приведенную ниже строку (в результате выравнивание слов отключится, и ваш исходный HTML-файл будет выглядеть нормально):

```
BOOL CSimpleNetView::PreCreateWindow(CREATESTRUCT& cs)
{
    BOOL bPreCreated = CEditView::PreCreateWindow(cs);
//  cs.style &= ~(ES_AUTOHSCROLL|WS_HSCROLL);
    return bPreCreated;
}
```

Откомпилируйте и запустите программу на выполнение. Выберите Open из главного меню, а затем введите в элементе редактирования требуемый URL. Нажмите на OK и программа подключится к вашему провайдеру (здесь предполагается, что система сконфигурирована на автодозвон к провайдеру, иначе соединение придется установить вручную). Затем программа загрузит в главное окно исходный код HTML. При помощи функций печати или кнопки Save as можно распечатать или сохранить каждую страницу в файлах. На рис. 21.26 показана программа SimpleNet, выполнившая соединение с Web-сайтом компании Coriolis Group.

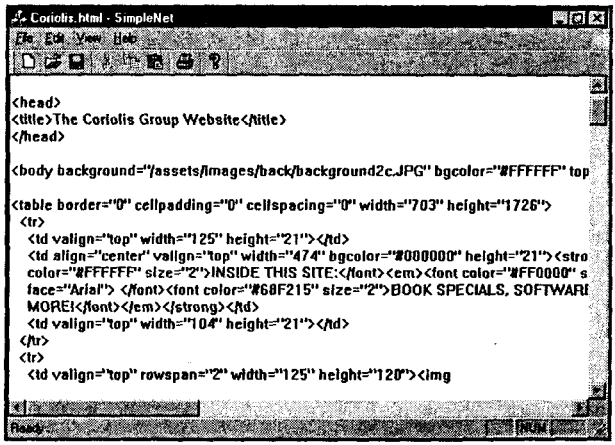


РИСУНОК 21.26.
Программа SimpleNet в работе.

Еще о WinInet

WinInet умеет делать гораздо больше, чем показано в последнем примере. Можно, скажем, унаследовать свои классы от **CInternetSession** и автоматически получать сообщения о состоянии соединения, перекрыв функцию **OnStatusCallback()**.

WinInet автоматически поддерживает сеансы FTP (File Transfer Protocol — протокол передачи файлов) так же просто, как и HTML-файлы, позволяя записывать и считывать с FTP-сервера. В таком случае вместо **GetURL()** должна использоваться **GetFtpConnection()**, которая позволяет выгрузить файл с удаленной машины одним только вызовом **GetFile()** — здесь работы еще меньше, чем в SimpleNet.

Сохранение персонального броузера

Прежде чем окончательно проститься, давайте завершим работу над приложением Bowser. Вот что еще потребуется сделать:

1. Добавьте в класс **CBDoc** общедоступную переменную типа **CString**. Назовите ее **m_URL**, а в функции **OnNewDocument()** проинициализируйте ее пустой строкой ("").
2. В функцию **CBDoc::Serialize()** поместите код из листинга 21.9.

Листинг 21.9. Функция CBDoc::Serialize().

```
const MAX_BUF = 1024;
void CBDoc::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
        if (m_URL.Left(4) == "res:")
        {
            AfxMessageBox(
              "Cannot save internal HTML files");
            return;
        }
        if (! m_URL.IsEmpty())
        {
            char buf[MAX_BUF];
            CInternetSession session;
            CStdioFile * pFile = NULL;
            pFile = session.OpenURL(m_URL);
            int nBytesRead;
            while ((nBytesRead = pFile->Read(buf,
                        MAX_BUF)) > 0)
            {
                ar.Write(buf, nBytesRead);
            }
            delete pFile;
            session.Close();
        }
    }
    else
    {
        // ЧТО СДЕЛАТЬ: Поместить здесь код загрузки
    }
}
```

3. В начало файла BowserDoc.cpp добавьте строку **#include <afxinet.h>**.
4. В функцию **CBView::OnDocumentComplete()** вставьте код, приведенный в листинге 21.10.

Листинг 21.10. Функция CBView::OnDocumentComplete().

```
void CBView::OnDocumentComplete(LPCTSTR lpszURL)
{
    CHtmlView::OnDocumentComplete(lpszURL);
    AfxGetMainWnd()->SetWindowText(GetLocationName());
    (GetDocument())->m_URL = lpszURL;
}
```

Откомпилируйте новый броузер Bowser, загрузите несколько интересующих страниц и сохраните их. Вы увидите, что теперь Bowser прекрасно записывает HTML. Конечно, если страница ссылается на ресурсы, подобные файлам изображений с относительным URL, сохраненная страница не будет корректно функционировать после нового открытия, поскольку на локальном диске не окажется необходимых ей ресурсов. Конечно, можно было бы добавить в Bowser код, сканирующий HTML-файлы на предмет наличия таких ресурсов, и сохранять их вместе с кодом HTML. Но это уже другая история и, возможно, другая книга.

Пришло время проститься...

Поздравляем с завершением нашего путешествия! Полагаем, что вы получили от всего этого определенное наслаждение. Мы надеемся, что приведенная информация и примеры стоят потраченного на них времени — мы уверены, что они того заслуживают. Теперь вы имеете солидную основу для продолжения изучения MFC. При самостоятельном исследовании MFC вы еще получите массу удовольствия. Читайте, учитесь и программируйте! Пока!