bhv®
www.bhv.ru
www.bhv.kiev.ua

Николай Секунов

САМОУЧИТЕЛЬ

Visual C++ .NET

Создание основных типов приложений

Концепция "документ/представление"

Расширенные возможности Visual C++

Разработка справочной системы

Создание библиотек динамической компоновки

Дискета содержит файлы примеров и описание классов и функций Visual C++ .NET

Мощная среда программирования для разработки Windows–приложений

Николай Секунов

САМОУЧИТЕЛЬ
VISUAL C++ .NET

Санкт-Петербург
«БХВ-Петербург»

УДК 681.3.06

Книга посвящена методам объектно-ориентированного программирования для 32-разрядных операционных систем Windows. Рассмотрен широкий круг вопросов разработки диалоговых и многооконных приложений. Обсуждаются вопросы создания различных типов справочных систем приложения, их русификации и преобразования из одного типа в другой. Большое внимание уделено применению механизма исключений для обработки ошибок, работе с шаблонами, многозадачности и взаимодействию потоков. Отдельная глава посвящена созданию библиотек динамической компоновки (DLL). Приведен обширный справочный материал по функциям и классам Visual C++.

Для начинающих программистов

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зав. редакцией	*Анна Кузьмина*
Редактор	*Владислав Борисов*
Компьютерная верстка	*Натальи Смирновой*
Корректор	*Наталия Першакова*
Дизайн обложки	*Игоря Цырульникова*
Зав. производством	*Николай Тверских*

Секунов Н. Ю.
Самоучитель Visual C++ .NET. — СПб.: БХВ-Петербург, 2002. — 736 с.: ил.

ISBN 5-94157-032-5

© Н. Ю. Секунов, 2002
© Оформление, издательство "БХВ-Петербург", 2002

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 28.02.02.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 59,34.
Тираж 4000 экз. Заказ №
"БХВ-Петербург", 198005, Санкт-Петербург, Измайловский пр., 29.

Гигиеническое заключение на продукцию, товар № 77.99.1.953.П.950.3.99
от 01.03.1999 г. выдано Департаментом ГСЭН Минздрава России.

Отпечатано с готовых диапозитивов
в ФГУП ордена Трудового Красного Знамени "Техническая книга"
Министерства Российской Федерации по делам печати,
телерадиовещания и средств массовых коммуникаций.
198005, Санкт-Петербург, Измайловский пр., 29.

Содержание

Введение **1**

Для кого предназначена эта книга? 2
Структура данной книги 3
Соглашения, принятые в данной книге 7
Требования к аппаратным средствам и к программному обеспечению 9

Часть I. Создание простейших приложений **11**

Глава 1. Использование мастера создания приложений **13**

Диалоговое приложение 13
Многооконное приложение Windows 17
Другие виды приложений MFC 19
Консольное приложение 20

Глава 2. Классы приложений, документов и представлений **25**

Класс документа 25
Класс представления 29
Класс приложения 35
Дизайн элементов управления и системный реестр 39
Шаблон документа 40
Создание окон 41
Работа с несколькими типами документов 43

Глава 3. Диалоговые окна и простейшие элементы управления **55**

Диалоговое окно 55
Формирование ресурсов диалогового окна 55
Внесение изменений в класс диалогового окна 69
Вкладки и мастера 82
Создание вкладок диалогового окна 82
Создание мастера 87
Некоторые модификации окна мастера 93

Часть II. Программирование интерфейса пользователя **99**

Глава 4. Классы элементов управления **101**

Класс списка 101

Классы линейного регулятора и линейного индикатора....110
Создание пользовательского линейного индикатора....119
Обработка даты и времени....125

Глава 5. Сообщения и команды....136

Обработка сообщений....136
Карта сообщений....138
Сообщения в Windows 3.*x*....141
Сообщения в Win32....141
Сообщения, посылаемые всеми новыми элементами управления Windows....142
Обработка извещений....143
Обработка отраженных сообщений....145
Макросы карты сообщений и заготовки функций обработки отраженных сообщений....146
Пример обработки отраженных сообщений....148
Использование карты сообщений приложением....153
Создание функций обработки сообщений....154
Диалоговое окно *Properties*....154
Список сообщений....159
Обновление команд....159

Глава 6. Вывод информации на экран....162

Интерфейс графических устройств (GDI)....162
Контекст устройства....163
Отображение текста....165
Шрифты....166
Вывод текста....173
Форматирование текста....173
Вывод текста....175
Программа вывода текста....176
Перерисовка окна....179
Использование перьев....181
Работа с кистью....185
Использование диалогового окна для настройки параметров....189
Работа с битовыми образами....195
Аппаратно-зависимые битовые образы....195
Аппаратно-независимые битовые образы....202

Глава 7. Работа с файлами документов....208

Работа с архивом....208
Непосредственное чтение и запись файлов....218
Использование объектов класса *CFile* при работе с классом *CArchive*....218
Автономное использование класса *CFile*....222
Работа с системным реестром....232

Глава 8. Работа с текстовыми документами 236

Создание простейшего текстового редактора 236
Создание более сложного редактора 238
Форматирование документов 244
Задание пользовательского шрифта 250

Глава 9. Панели инструментов и строка состояния 254

Работа с панелью инструментов 255
Удаление кнопок из панели инструментов 256
Добавление кнопок в панель инструментов 258
Работа со строкой состояния 272

Глава 10. Печать документов и организация прокрутки в окне 285

Организация прокрутки в окне 286
Режимы отображения 293
Распечатка и предварительный просмотр 294
Работа с окном предварительного просмотра печати 294
Распечатка многостраничного документа 299
Использование функций библиотеки MFC при печати 309

Часть III. Особенности программирования в среде Visual C++ 315

Глава 11. Исключения, шаблоны и новые возможности Visual C++ 317

Работа с исключениями 317
Аргументы исключений 318
Механизмы исключений Visual C++ 320
В каких случаях следует вызывать исключения 322
Перехват и уничтожение исключений 324
Уничтожение объектов в исключениях 326
Вызов исключений из функций пользователя 328
Преобразование макросов исключений в операторы C++ 329
Совместное использование макросов и операторов C++ 333
Шаблоны 335
Понятие шаблона 336
Шаблоны функций 338
Шаблоны классов 341
Работа с классами коллекций 344
Виды классов коллекций 344
Классы коллекций, использующие шаблоны 347
Классы коллекций, не использующие шаблоны 352
Доступ к элементам классов коллекций 353
Удаление элементов классов коллекций 355
Использование классов коллекций 357

Глава 12. Многозадачность на основе потоков Windows.....360

Независимая работа потоков 361
Создание рабочего потока.....361
Создание интерфейсных потоков.....363
Прекращение работы потока 365
Взаимодействие между потоками.....366
Взаимодействие между потоком и приложением 367
Использование классов синхронизации 367
Простейший пример работы с потоками 369
Более сложный пример работы с потоками.....377

Глава 13. Справка в приложении 387

Описание справочной системы приложения.....387
Способы доступа к справочной системе 388
Способы представления справочной информации 389
Формы представления справочной информации 403
Программирование справочной системы 403
Компоненты справочной системы 404
Обработка сообщений справочной системы 407
Русификация файла ресурсов 412
Создание системы командной справки 437
Создание системы контекстной справки.....441
Подготовка справочных текстов.....446
Русификация справочной системы приложения 446
Добавление новых тем 473
Форматирование текстовых файлов справки 477
Использование графики и гиперграфических ссылок.....480
Использование макросов справочной системы 481
Внесение изменений в оглавление справочной системы 481
Использование справочной системы HTML 485
Преобразование справочной системы приложения 486
Диалоговое окно *О программе*.....511

Глава 14. Отладка приложения.....513

Средства отладки, предоставляемые интерфейсом пользователя.....513
Точки останова 515
Анализ исполнения программы.....519
Настройка уровня предупреждений транслятора 524
Программные средства отладки 526
Макросы *ASSERT* и *TRACE* 526
Отладочные функции.....528
Глобальные диагностические функции 529
Отладка распространяемой версии приложения 530
Устранение утечки памяти 531
Основные причины возникновения утечек памяти 531
Отладочные версии операторов *new* и *delete*.....535

Глава 15. Создание пользовательской библиотеки 538

Использование библиотек динамической компоновки 539
Главная функция библиотеки динамической компоновки 539
Экспорт и импорт функций 540
Неявная компоновка 542
Явная компоновка 542
Использование для работы с библиотеками динамической компоновки 543
Регулярные библиотеки динамической компоновки MFC 544
Библиотеки динамической компоновки расширения MFC 545
Функция *DllMain* 547
Экспорт классов и функций 548
Передача ресурсов 549
Использование файлов DEF 550
Пример создания и использования библиотеки динамической компоновки 555
Создание регулярной библиотеки динамической компоновки 555
Создание библиотеки динамической компоновки расширения MFC 562
Создание демонстрационного приложения 579
Описание созданных проектов 583

Глава 16. Создание простейшего приложения Internet 587

Классы WinInet 587
Чтение Web-страницы 588

Часть IV. Приложения 599

Приложение 1. Объектно-ориентированное программирование и классы 601

Обзор объектно-ориентированных методов программирования 601
Инкапсуляция 602
Наследование 603
Полиморфизм 604
Классы 605
Классы как типы данных 605
Файлы заголовков и файлы реализации 605
Когда следует, а когда не следует использовать классы 612
Перегрузка функций и операторов 612
Использование виртуальных функций 614
Область действия класса 615

Приложение 2. Интерфейс пользователя Visual C++ 617

Первая страница 618
Панели инструментов 620
Панель инструментов *Standard* 623
Система меню 625
Меню *File* 625
Команда *File / New* 625
Команда *File / Open* 628

Команда *File / Close* 630
Команды *File / Add New Item* и *File / Add Existing Item* 630
Команда *File / Add Project* 630
Команда *File / Open Solution* 631
Команда *File / Close Solution* 631
Команда *File / Save* 632
Команда *File / Save As* 632
Команда *File / Advanced Save Options* 632
Команда *File / Save All* 633
Команда *File / Source Control* 633
Команда *File / Page Setup* 634
Команда *File / Print* 634
Команда *File / Recent Files* 635
Команда *File / Recent Projects* 635
Команда *File / Exit* 635
Меню *Edit* 636
Команда *Edit / Undo* 636
Команда *Edit / Redo* 637
Команда *Edit / Cut* 637
Команда *Edit / Copy* 638
Команда *Edit / Paste* 638
Команда *Edit / Delete* 638
Команда *Edit / Select All* 638
Команда *Edit / Find and Replace* 638
Команда *Edit / Go To* 647
Команда *Edit / Insert File As Text* 647
Команда *Edit / Advanced* 648
Команда *Edit / Bookmarks* 650
Команда *Edit / Outlining* 651
Команда *Edit / IntelliSense* 655
Меню *View* 657
Команда *View / Open* 657
Команда *View / Open With* 657
Команда *View / Solution Explorer* 657
Команда *View / Class View* 659
Команда *View / Server Explorer* 659
Команда *View / Resource View* 659
Команда *View / Properties Window* 659
Команда *View / Toolbox* 661
Команда *View / Web Browser* 661
Команда *View / Other Windows* 661
Команда *View / Show Tasks* 662
Команда *View / Toolbars* 662
Команда *View / Full Screen* 662
Команда *View / Navigate Backward* 664
Команда *View / Navigate Forward* 664
Команда *View / Property Pages* 664

Меню *Project*....664
Команда *Project / Add Class*....666
Команда *Project / Add Function*....666
Команда *Project / Add Variable*....666
Команда *Project / Add Resource*....668
Команда *Project / Add New Item*....668
Команда *Project / Add Existing Item*....669
Команда *Project / New Folder*....669
Команда *Project / Unload Project*....669
Команда *Project / Add Web Reference*....670
Команда *Project / Set as StartUp Project*....670
Команда *Project / Properties*....671
Меню *Build*....671
Команда *Build / Build*....671
Команда *Build / Rebuild All*....671
Команда *Build / Clean*....672
Команда *Build / Batch Build*....672
Команда *Build / Configuration Manager*....673
Команда *Build / Compile*....673
Команда *Build / Deploy*....673
Меню *Debug*....673
Команда *Debug / Windows*....673
Команда *Debug / Start*....675
Команда *Debug / Break All*....675
Команда *Debug / Stop Debugging*....675
Команда *Debug / Detach All*....675
Команда *Debug / Restart*....675
Команда *Debug / Apply Code Changes*....675
Команда *Debug / Processes*....675
Команда *Debug / Exceptions*....676
Команда *Debug / Step Into*....676
Команда *Debug / Step Over*....676
Команда *Debug / Step Out*....677
Команда *Debug / QuickWatch*....677
Команда *Debug / New Breakpoint*....677
Команда *Debug / Clear All Breakpoints*....679
Команда *Debug / Disable Breakpoint*....679
Команда *Debug / Save Dump As*....679
Меню *Tools*....679
Команда *Tools / Debug Processes*....680
Команда *Tools / Connect to Database*....680
Команда *Tools / Connect to Server*....680
Команда *Tools / Customize Toolbox*....680
Команда *Tools / Add-in Manager*....681
Команда *Tools / Build Comment Web Pages*....683
Команда *Tools / Macros*....684

Команда *Tools / ActiveX Control Test Container* 684
Команда *Tools / Create GUID* 684
Команда *Tools / Error Lookup* 686
Команда *Tools / MFC/ATL Trace Tool* 686
Команда *Tools / OLE/COM Object Viewer* 687
Команда *Tools / Spy++* 687
Команда *Tools / External Tools* 687
Команда *Tools / Customize* 689
Команда *Tools / Options* 689
Меню *Window* 691
Команда *Window / New Window* 691
Команда *Window / Split* 691
Команда *Window / Dockable* 692
Команда *Window / Hide* 692
Команда *Window / Floating* 692
Команда *Window / Auto Hide* 692
Команда *Window / Auto Hide All* 692
Команда *Window / New Horizontal Tab Group* 694
Команда *Window / New Vertical Tab Group* 694
Команда *Window / Move to Next Tab Group* 695
Команда *Window / Move to Previous Tab Group* 695
Команда *Window / Close All Documents* 695
Список открытых окон 696
Команда *Window / Windows* 696
Меню *Help* 696
Команда *Help / Dynamic Help* 697
Команда *Help / Contents* 698
Команда *Help / Index* 698
Команда *Help / Search* 699
Команда *Help / Index results* 699
Команда *Help / Search results* 700
Остальные команды меню 700
Окна Visual Studio.NET 701
Окно *Solution Explorer* 701
Окно *Class View* 701
Окно *Properties* 704
Окно *Watch* 704
Окно *Breakpoints* 705

Приложение 3. Описание дискеты 707

Предметный указатель 709

Введение

Windows изначально разрабатывалась как графическая среда для "домохозяек", которая впоследствии превратилась в операционную систему для "домохозяек", сохранив при этом основной подход к данной категории программных продуктов. Отличительной чертой подобных программных продуктов является то, что они устанавливаются на компьютер до его передачи пользователю специалистом высокой квалификации, способным устранить все конфликты, возникающие в установленной программной конфигурации. После того как компьютер передается пользователю, на него не может быть установлена ни одна новая программа, поскольку это может привести к краху всей системы. И, действительно, о какой надежности может идти речь в операционной системе, при установке в которую нового приложения может быть замещена добрая половина системных библиотек. Следствием данного подхода является недопустимость создания в данной операционной системе новых программ.

Поскольку писать программы под Windows все-таки приходится, для этого лучше всего использовать систему программирования, разработчики которой хорошо знакомы с используемой операционной системой и которая требует от программиста минимального вмешательства в процесс написания программ. Этим требованиям как нельзя лучше удовлетворяет визуальная среда программирования Visual Studio.NET, разработанная корпорацией Microsoft. Наиболее весомым доводом в пользу использования именно этой системы является тот факт, что она разработана в той же корпорации, что и операционная система, в которой ей предстоит работать.

Однако всегда следует помнить, что даже разработка операционной системы и среды программирования в одной корпорации не гарантирует их нормальной совместной работы, особенно под управлением новых версий операционных систем, которые отсутствовали на момент создания среды разработки. Например, среда программирования Visual Studio 6.0, нормально работавшая под управлением Windows 95, под управлением Windows 98 уже работает некорректно. После завершения работы с созданным ею простейшим диалоговым приложением, обрабатывающим мультимедийный файл и не работающим с панелями инструментов, помещение указателя мыши на панель управления (панель инструментов задачи или Панель задач рабочего стола) считается недопустимой операцией, после которой необходимо прекратить выполнение задачи. Отсюда следуют два вывода: во-первых, среду программирования нужно обновлять одновременно с операционной системой и, во-вторых, нужно пользоваться продуктами Microsoft, поскольку информация об операционной системе является недоступной разработчикам других сред программирования и их продукты будут работать крайне ненадежно.

Среда программирования Visual Studio.NET, являющаяся мощным средством для разработки 32-разрядных приложений в операционных системах Windows 95,

Windows 98 и Windows NT. Она содержит полный набор средств для быстрой разработки Internet-приложений масштаба предприятия, средства упрощения совместной разработки приложений группой программистов и средства распространения созданных приложений. Этот достаточно сложный продукт позволяет создавать намного более объемные и более сложные приложения, чем его предшественники, разработанные для 16-разрядного Windows или для DOS.

В принципе, программу любой сложности можно написать на любом языке программирования и с использованием любой среды программирования. Это, конечно, так, но хорошая среда программирования отличается от плохой тем, что в ней на программирование той же самой задачи будет затрачено меньше времени, а получившаяся в результате программа будет работать быстрее и надежнее. Для создания подобной среды программирования необходимо досконально изучить среду, в которой будет работать скомпилированная программа, а кто изучил эту среду лучше ее разработчика. При этом следует особенно учесть, что вместо подробного описания корпорация Microsoft предоставила общественности схематический план своего творения.

Пользователь, скорее всего, оценит возможности, предоставляемые ему Visual Studio.NET. Основой этой среды является новый компонентно-ориентированный язык программирования C#, обеспечивающий надежную совместную работу компонентов, написанных с использованием различных языков программирования. Однако наиболее полно возможности этого языка используются при работе с компонентами, написанными на языке C++. Язык Visual C++, включенный в Visual Studio.NET, сохранил все основные особенности своей предыдущей версии 6.0, в том числе связь с библиотекой базовых классов Microsoft Foundation Classes или MFC и использование мастеров, позволяющих получать заготовки больших фрагментов исходного кода программы путем выбора соответствующих опций в панелях специальным образом организованных диалоговых окон.

Для кого предназначена эта книга

Данная книга посвящена языку программирования Visual C++, а не C++. В ней будут рассмотрены не конструкции данного языка, а классы библиотеки MFC, позволяющие выполнить ту или иную задачу, встающую перед разработчиком полноценного приложения.

В настоящее время остается еще достаточно много программистов, предпочитающих работать в среде DOS. Работа в этой среде позволяет более эффективно использовать системные ресурсы и создавать компактные и быстродействующие приложения. Доказательством этому служит активное использование подобных приложений самой корпорацией Microsoft. Приложения, работающие в среде DOS, называются в среде Windows консольными приложениями. В Visual Studio.NET имеются средства для разработки этого типа приложений, правда, не такие удобные, как в предыдущей версии данного продукта.

Несмотря на то, что Microsoft позиционирует среду Visual Studio.NET как конкурента среды программирования C++ Builder фирмы Inprise, успешно используемого для разработки приложений, работающих с базами данных, в ней еще осталось много свойств от ее предыдущей версии, которая была хорошо приспособлена для создания небольших приложений Windows. Данная книга посвящена описанию возможностей, предоставляемых данной средой в указанной области. Другими потенциальными читателями этой книги являются программисты, имеющие опыт программирования в среде Windows 3.*x* и продолжающие его использовать при работе в Visual C++, не желая изучать особенности данной среды программирования. Конкуренция и требования рынка уже заставляют их не просто использовать некоторые заготовки, предоставляемые им мастером AppWizard, но и более полно применять описанные в данной книге возможности, предоставляемые им библиотекой MFC.

У каждого пользователя среды программирования Visual C++, пытавшегося разобраться в ней самостоятельно, накопилось множество вопросов, и он надеется получить в одной книге ответы на большинство из них. Как показывает опыт, уровень подготовки, даже у достаточно опытных программистов, существенно отличается: некоторые до сих пор писали исключительно в среде DOS и теперь, перейдя к работе в среде Windows, нуждаются в простейшем компиляторе для своих задач, не использующих и не требующих использования сложного графического интерфейса, другие работали в среде Windows 3.*x* и не привыкли следовать при написании программ требованиям, предъявляемым 32-разрядными приложениями, но хотели бы это делать, и, наконец, любому программисту нужен достаточно подробный справочник, в котором он мог бы найти решение стандартных проблем, возникающих при программировании.

Поскольку нельзя объять необъятное, данная книга не является учебником по алгоритмическому языку C++ и операционной системе Windows. Предполагается, что читатель уже написал, по крайней мере, пару программ на этом языке. Однако, как выяснилось, многие программисты, работавшие под DOS и, даже, использовавшие библиотеку классов Turbo Vision фирмы Borland, имеют о классах достаточно смутное представление. Поэтому в данную книгу включена глава, посвященная концепции объектного программирования и классам. Те, кто считает, что он достаточно хорошо знаком с этим вопросом, может пропустить эту главу.

Структура данной книги

Поскольку предполагается, что читатель данной книги может не иметь опыта программирования в среде Windows, то в данной книге содержится весь необходимый материал, позволяющий начать программирование приложений практически с нуля. В приведенном ниже списке разделов и глав дано краткое их описание, позволяющее пользователю лучше ориентироваться в структуре книги.

- Часть I. Создание простейших приложений.

  В данной части рассмотрено создание простейших приложений, использующих основные элементы интерфейса Windows. Здесь же рассмотрена кон-

цепция Документ/Представление, лежащая в основе программирования с использованием библиотеки MFC.

- Глава 1. Использование мастера создания приложений.

  В данной главе рассмотрен мастер создания приложений, позволяющий автоматически создать заготовку приложения, определив в нем некоторые ключевые его особенности. Здесь же рассмотрено создание консольного приложения, работающего в окне MS DOS.

- Глава 2. Классы приложений, документов и представлений.

  В данной главе рассмотрена концепция Документ/Представление. Кроме того, в ней описаны классы шаблонов документов.

- Глава 3. Диалоговые окна и простейшие элементы управления.

  В данной главе рассмотрено создание диалоговых окон с использованием редактора ресурсов и описано включение в них простейших элементов управления.

❒ Часть II. Программирование интерфейса пользователя.

В данной части содержится описание того, каким образом пользователь может выводить на экран необходимую информацию и управлять работой приложения.

- Глава 4. Классы элементов управления.

  В данной главе рассмотрены достаточно сложные элементы управления, работа с которыми требует создания объектов классов данных элементов управления и вызова содержащихся в них методов обработки информации.

- Глава 5. Сообщения и команды.

  В данной главе рассмотрен механизм обработки сообщений, используемый в операционной системе Windows для обеспечения взаимодействия различных процессов и потоков в приложении.

- Глава 6. Вывод информации на экран.

  В данной главе рассмотрены принципы работы интерфейса графических устройств Windows (GDI).

- Глава 7. Работа с файлами документов.

  В данной главе содержится описание работы с файловой системой Windows.

- Глава 8. Работа с текстовыми документами.

  Операционная система Windows создавалась преимущественно для работы с текстовыми документами, и оптимизирована исключительно для выполнения данной задачи. Поэтому, вопросы создания приложения, работающего с текстовой информацией, вынесены в отдельную главу.

- Глава 9. Панели инструментов и строка состояния.

  Панели инструментов и строка состояния являются непременным атрибутом любого законченного приложения Windows. В данной главе описаны

способы создания и программирования работы с панелями инструментов и строкой состояния.

- Глава 10. Печать документов и организация прокрутки в окне.

  Печать документов и прокрутка изображения в окне большей частью реализуются с использованием стандартных средств среды программирования Visual C++, однако, для обеспечения работы данных функций в приложении необходимо произвести его настройку. Этому вопросу и посвящена данная глава.

❒ Часть III. Особенности программирования в среде Visual C++.

Средств, описанных в первых двух частях данной книги, достаточно для создания работоспособного приложения. Однако среда программирования Visual C++ предоставляет пользователю возможности, позволяющие уменьшить размеры исходных текстов программ, повысить их надежность и рационально распределить предоставляемые приложению вычислительные ресурсы. Описание этих возможностей содержится в данной части.

- Глава 11. Исключения, шаблоны и новые возможности Visual C++.

  В данной главе рассматриваются новые возможности, предоставляемые средой программирования Visual C++. Исключения представляют собой механизм, существенно упрощающий процесс обработки ошибок. Без использования этого механизма практически невозможно создать надежно работающее приложение. Шаблоны представляют собой метод задания функций обработки для переменных абстрактных классов. Это позволяет существенно сократить количество практически идентичных по тексту функций, выполняющих одну и ту же операцию для переменных разных типов. Использование шаблонов позволяет сократить размер исходного текста программ и обеспечивает гарантированное внесение изменений во все функции обработки различных типов данных, выполняющих над ними одну и ту же операцию.

- Глава 12. Многозадачность на основе потоков Windows.

  Поток Windows представляет собой основную программную единицу, которой операционная система Windows предоставляет квант процессорного времени. Создание нескольких потоков в одном приложении позволяет более рационально использовать выделяемое данному приложению процессорное время. Создание фоновых потоков и правильная расстановка приоритетов позволяет обеспечить быструю реакцию приложения на запрос пользователя при максимально возможной загрузке процессора.

- Глава 13. Справка в приложении.

  Любое серьезное приложение, независимо от того, создается ли оно для внутреннего использования или для продажи, должно содержать обширную справочную систему, позволяющую получить справку по любому вопросу, связанному с данным приложением. Для коммерческих приложений, а также для приложений, рассчитанных на пользователя со слабым

знанием английского языка, встает вопрос русификации приложения и содержащейся в нем справочной системы. Поэтому в данной главе рассмотрены вопросы не только связанные с созданием справочной системы, но и с ее русификацией. Параллельно рассмотрен вопрос русификации всего приложения.

- Глава 14. Отладка приложения.

  Отладка приложения является непременным этапом его создания. Практически ни одно достаточно серьезное приложение не заработало безошибочно с первого раза. В данной главе рассмотрены возможности, предоставляемые средой программирования Visual C++ для отладки приложения.

- Глава 15. Создание пользовательской библиотеки.

  Любой программист, написавший хотя бы одно полноценное приложение, хочет использовать его отдельные компоненты в других своих приложениях. То же самое относится и к любому специалисту в некоторой области, которому хотелось бы применять в своих приложениях некоторый инструментарий, содержащий стандартные методы данной области. Все эти вопросы могут быть решены с использованием пользовательских библиотек, принципы создания которых описаны в данной главе.

- Глава 16. Создание простейшего приложения Internet.

  Microsoft позиционирует Visual Studio.NET, прежде всего, как средство создания многоуровневых систем в Internet. Поэтому в данной главе приведено краткое описание классов WinInet и продемонстрировано их использование для создания простейшего приложения для чтения Web-страниц.

❒ Приложения

В приложениях изложены вопросы, не связанные напрямую с программированием приложений, но изучение которых может способствовать успешной работе в Visual C++.

- Приложение 1. Объектно-ориентированное программирование и классы.

  Как уже отмечалось выше, опыт показывает, что не все программисты, использующие классы, хорошо понимают их назначение. Поскольку использование библиотеки MFC построено на использовании классов и для эффективного ее использования пользователь должен хорошо разбираться в основах ООП, то в данном приложении дан краткий обзор данной концепции и связанного с ней понятия класса.

- Приложение 2. Интерфейс пользователя Visual C++.

  Visual C++ представляет собой достаточно сложную среду программирования. Хотя ее интерфейс и задумывался как "интуитивно ясный", но многие функции данной среды таковыми не являются. Кроме того, пользователь может и не догадываться насколько ему необходима данная функция, пока он не прочтет о ней и не попробует ею воспользоваться. Данное приложе-

ние содержит описание интерфейса пользователя, достаточное для начала работы с ним абсолютно неподготовленного пользователя.

- Приложение 3. Описание дискеты.

  Помимо самораспаковывающегося архива SAMPLES.EXE, на дискете находится файл с описанием классов и функций Visual C++. Содержимое этого файла представляет собой выборочный перевод информации, находящейся в библиотеке MSDN. Необходимость размещения данного файла на дискете обусловлена тем, что далеко не все пользователи настолько хорошо знают английский язык, чтобы быть уверенными в том, что они поймут все тонкости использования функций, описанных в данной библиотеке. В этом файле пользователь может найти полное описание большинства функций, используемых в программах, приведенных в качестве примеров.

  Все функции, классы и структуры в данном файле расположены в алфавитном порядке. Сначала идет описание глобальных функций, а затем описание классов и структур.

Соглашения, принятые в данной книге

В данной книге использовалось специальное форматирование текста для выделения некоторых текстов или фрагментов текста. Ниже приведены основные принципы выделения текста.

- Исходные тексты фрагментов программ, представляющие собой одну или более строк текста, выделяются специальным шрифтом, как это показано ниже:

```
void CMainFrame::Dump(CDumpContext& dc) const
{
  CMDIFrameWnd::Dump(dc);
}
```

- Когда на эти фрагменты программ имеются ссылки из различных фрагментов текста, они оформляются в виде листинга, как показано ниже.

Листинг 1. Заголовок листинга

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
  if (CMDIFrameWnd::OnCreate(lpCreateStruct) == -1)
    return -1;

  if (!m_wndToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD | WS_VISIBLE |
      CBRS_TOP | CBRS_GRIPPER | CBRS_TOOLTIPS | CBRS_FLYBY |
      CBRS_SIZE_DYNAMIC) || !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
```

```
  {
    TRACE0("Failed to create toolbar\n");
    return -1;  // fail to create
  }

  if (!m_wndStatusBar.Create(this) ||
    !m_wndStatusBar.SetIndicators(indicators,
      sizeof(indicators)/sizeof(UINT)))
  {
    TRACE0("Failed to create status bar\n");
    return -1;  // fail to create
  }

  // TODO: Delete these three lines if you don't want the toolbar to
  //  be dockable
  m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
  EnableDocking(CBRS_ALIGN_ANY);
  DockControlBar(&m_wndToolBar);

  return 0;
}
```

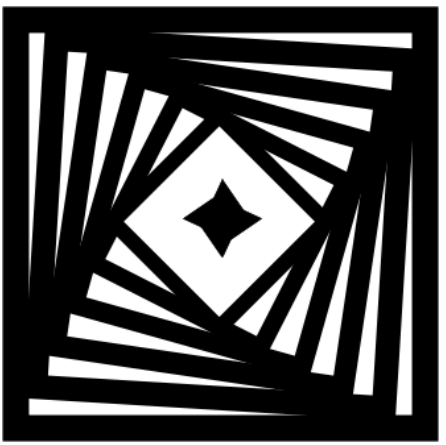
- Если в тексте встречается имя класса, функции, имя типа переменной или фрагмент текста длиной менее строки, он выделяется специальным шрифтом. Например: `int`, `CMainFrame`, `DockControlBar` и т. д.
- Имена функции, обычно, даются в полной форме. Это значит, что если функция является членом класса, то перед ней указывается имя класса, членом которого она является. При этом указывается имя того класса в иерархии, в котором впервые появилась данная функция. Например: `CObject::Serialize`. Если перед именем функции не стоит имя класса, то это означает, что это либо глобальная функция, либо рассматриваемая в настоящее время функция пользовательского класса, либо часто встречающаяся функция, полное описание которой было только что приведено. Это позволяет сравнительно просто отыскать описание данной функции на дискете, прилагаемой к этой книге.
- Заголовок диалогового окна, имя кнопки или команды меню выделяются полужирным шрифтом. Например: **Open**, команда меню **File | New** и т. д.
- Имя клавиши заключается в угловые скобки (<>). Например: <Ctrl>, <F5> и т. д.
- Если требуется одновременно нажать несколько клавиш, они объединяются знаком (+). Например: <Ctrl>+<Alt>+<Del>.
- При первом появлении нового термина он выделяется курсивом. Например: *новый термин*.
- Ссылка на другую главу содержит ее название и номер. Например: *см. главу 5*.

Требования к аппаратным средствам и к программному обеспечению

Для работы в среде программирования Visual Studio.NET требуется следующее аппаратное и программное обеспечение:

- IBM совместимый PC с процессором Intel Pentium II 450 МГц или лучшим;
- не менее 256 Мбайт оперативной памяти;
- жесткий диск объемом не менее 3 Гбайт;
- дисплей SVGA и соответствующий видеоадаптер, обеспечивающие разрешение не менее 800×600 точек; дисковод CD-ROM;
- совместимая с Microsoft мышь; операционная система Windows 98, Windows ME, Windows NT или Windows 2000.

Естественно, эти требования являются минимальными и работа в такой конфигурации не доставит вам большого удовольствия.



ЧАСТЬ I

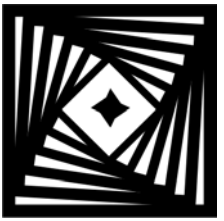
Создание простейших приложений

Глава 1. Использование мастера создания приложений

Глава 2. Классы приложений, документов и представлений

Глава 3. Диалоговые окна и простейшие элементы управления

Глава 1



Использование мастера создания приложений

Среда программирования Visual Studio.NET используется не только для редактирования, компиляции и отладки программ, но и для генерации их заготовок. Эти заготовки представляют собой работоспособные программы, реализующие основные элементы интерфейса Windows, необходимые в любой пользовательской программе, и, естественно, не выполняющие никакой обработки. Мастер приложений позволяет создать такую заготовку за несколько минут, однако, чтобы получить на ее основе полноценное коммерческое приложение, уйдут недели и месяцы работы.

Программисту, начинающему работу в среде Visual Studio.NET, необходимо, прежде всего, ознакомиться с простейшими типами приложений, генерируемых мастерами. К ним следует отнести приложения, генерируемые мастером MFC (Microsoft Foundation Classes) Application Wizard. Эти типы приложений являются базовыми для языка Visual C++ и позволяют ознакомиться с его основными особенностями.

Библиотека MFC является одной из основных библиотек языка Visual C++. Она используется для работы с элементами управления окон Windows и обработки поступающих от них сообщений. Применение данной библиотеки существенно облегчает написание программ на языке Visual C++.

Мастер создания приложений MFC позволяет создавать четыре вида приложений:

- диалоговые приложения;
- однооконные приложения;
- многооконные приложения;
- приложения с несколькими документами верхнего уровня.

Первые три типа приложений хорошо знакомы программистам, использовавшим предыдущие версии языка Visual C++.

Диалоговое приложение

Диалоговые приложения рассматриваются корпорацией Microsoft как простейший тип приложения, которое не должно иметь никаких меню, кроме системного, а также не может открывать или сохранять информацию в файле. Оба эти ограничения могут быть обойдены, однако при этом возникают определенные сложности. Другое неудобство, возникающее при работе с диалоговыми прило-

жениями, связано с тем, что, по мнению разработчиков Visual C++, считывание из него информации должно производиться только при его закрытии, что исключает взаимодействие его элементов управления. Поэтому очень часто возникают сложности при необходимости вывести в одном элементе управления изменения, произведенные в другом элементе управления. Особые сложности возникают в том случае, если изменения вносятся в текстовое поле.

Диалоговое приложение является полноценным приложением Windows, в котором может быть использована полноценная поддержка технологии ActiveX, что позволяет решить вопрос с передачей данных в приложение и из приложения. С другой стороны, это приложение может быть оформлено в виде самостоятельного модуля, что позволит задать ему при выполнении уровень приоритета, недоступный для потока основного приложения. Поэтому диалоговое приложение может использоваться для реализации критичных ко времени исполнения фрагментов программы.

Текст диалогового приложения можно найти в каталоге Dialog на прилагаемой к данной книге дискете.

Чтобы самостоятельно создать заготовку диалогового приложения с использованием мастера создания приложений MFC:

1. Закройте все открытые проекты и файлы и выберите команду **File | New | Project** (Файл | Создать | Проект) или нажмите кнопку **New Project** (Новый проект) на панели инструментов **Standard** (Стандартная). Появится диалоговое окно **New Project** (Новый проект), изображенное на рис. 1.1.

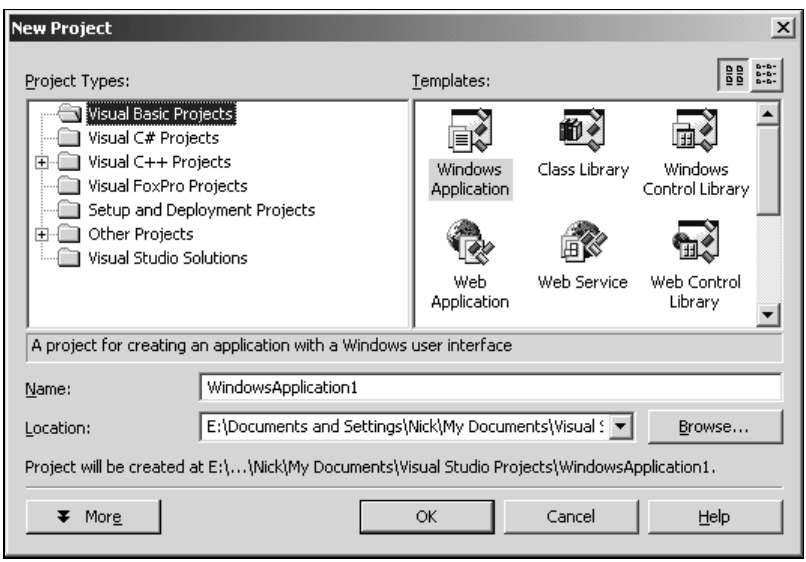


Рис. 1.1. Диалоговое окно **New Project**

2. В окне иерархического списка **Project Types** (Типы проектов) раскройте папку **Visual C++ Projects** (Проекты Visual C++). Диалоговое окно **New Project** (Новый проект) примет вид, изображенный на рис. 1.2.

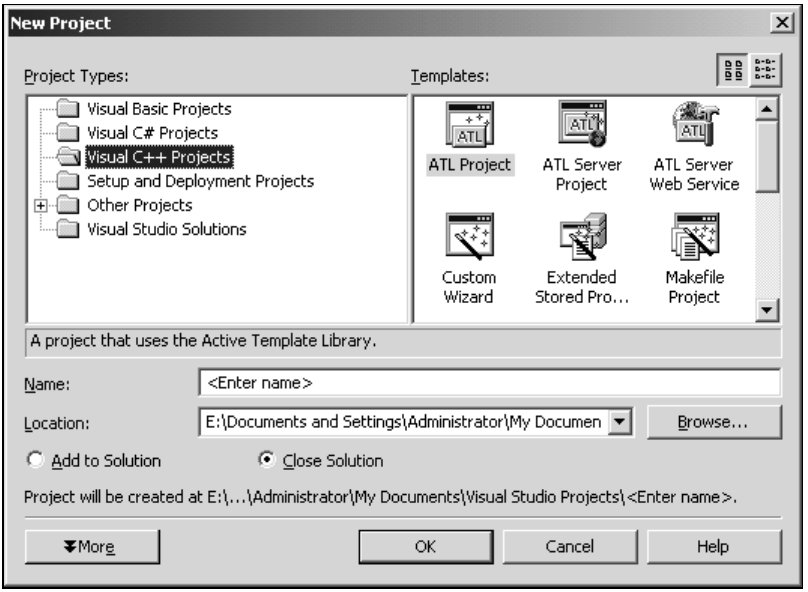


Рис. 1.2. Раскрытая папка **Visual C++ Projects**

3. В окне списка **Templates** (Шаблоны) выделите значок **MFC Application** (Приложение MFC), в текстовое поле **Name** (Имя) введите имя приложения "Dialog" и нажмите кнопку **OK**. Появится диалоговое окно **MFC Application Wizard - Dialog** (Мастер создания приложений MFC), изображенное на рис. 1.3.
4. Раскройте вкладку **Application Type** (Тип приложения). Диалоговое окно **MFC Application Wizard** (Мастер создания приложений MFC) примет вид, изображенный на рис. 1.4.
5. Установите переключатель **Application type** (Выбор типа приложения) в положение **Dialog based** (Диалоговое) и нажмите кнопку **Finish** (Готово). Мастер MFC Application Wizard создаст заготовку диалогового приложения, как это показано на рис. 1.5.

В правой части окна Visual Studio.NET расположен ярлык вкладки **Toolbox** (Инструментарий), позволяющей автоматизировать процесс создания диалогового окна. Элементы управления диалогового окна и использование вкладки **Toolbox** (Инструментарий) будут более подробно описаны в *главе 3*, а попытка написания примера программы на базе этого приложения наталкивается на специфичность области его использования.

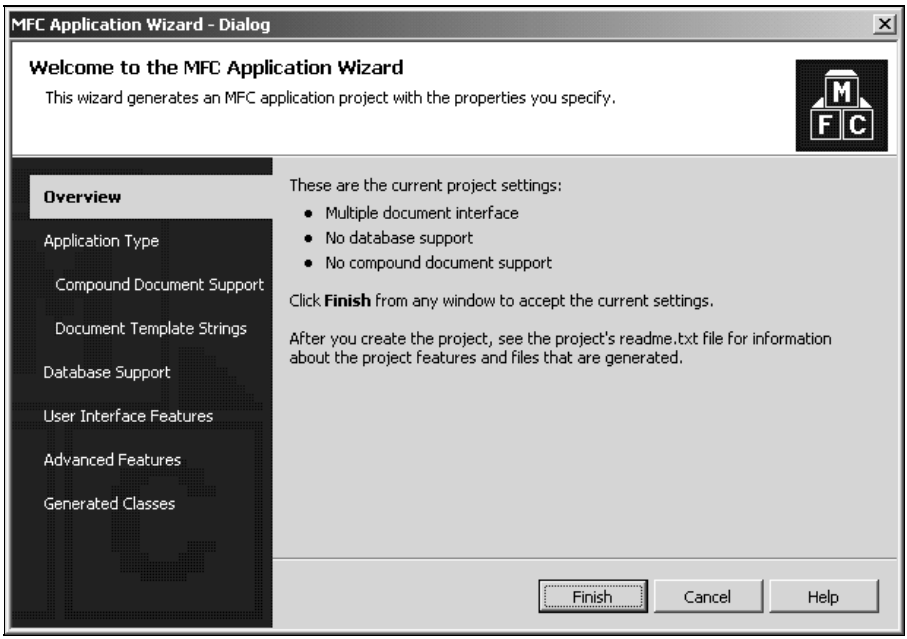


Рис. 1.3. Диалоговое окно **MFC Application Wizard - Dialog**

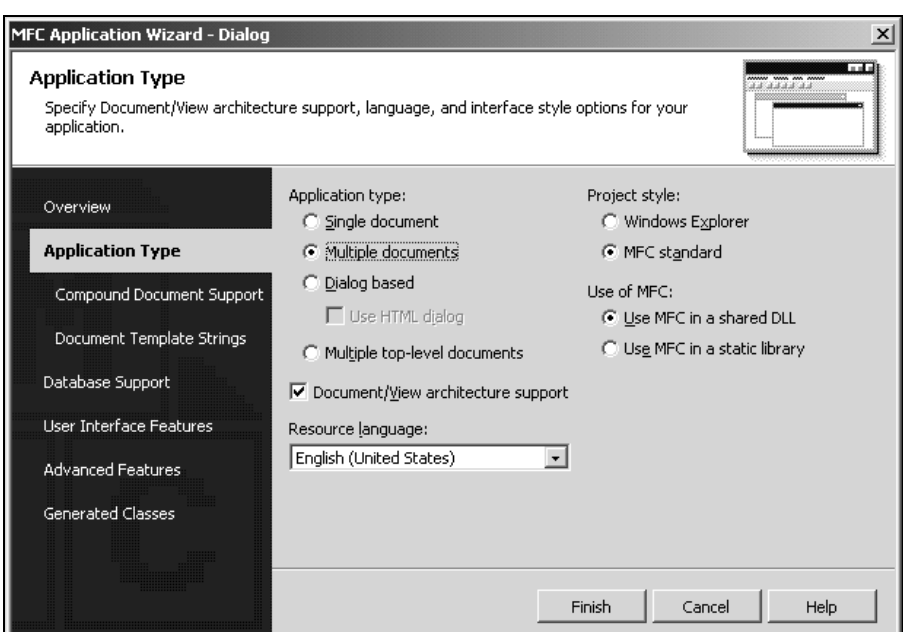


Рис. 1.4. Вкладка **Application Type**

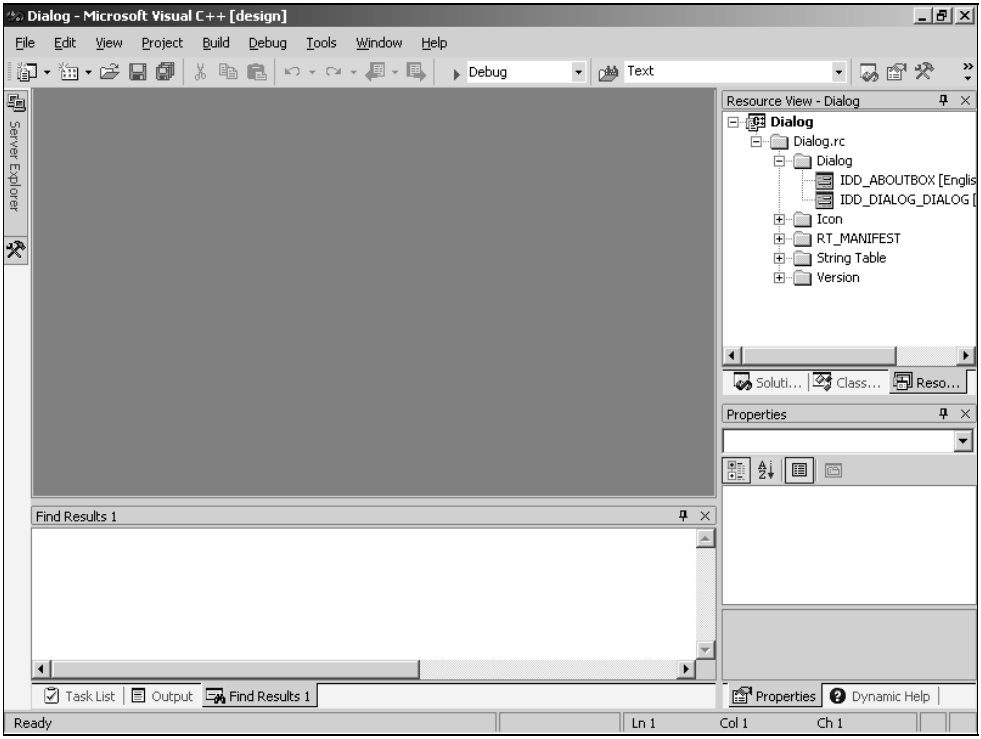


Рис. 1.5. Заготовка диалогового окна

Многооконное приложение Windows

Под многооконным приложением Windows мы будем понимать приложение, использующее MDI (Multiple Document Interface, Многооконный интерфейс приложения). Подобные приложения являются основным типом приложений в Windows, поэтому в дальнейшем мы будем в основном рассматривать приложения этого вида.

Демонстрационное многооконное приложение можно найти в папке MDI на прилагаемой к данной книге дискете.

Чтобы самостоятельно создать заготовку многооконного приложения Windows с использованием мастера MFC Application Wizard:

1. Закройте все открытые проекты и файлы и выберите команду **File | New | Project** (Файл | Новый | Проект) или нажмите кнопку **New Project** (Новый проект) на панели инструментов **Standard** (Стандартная). Появится диалоговое окно **New Project** (Новый проект).
2. В окне иерархического списка **Project Types** (Типы проектов) раскройте папку **Visual C++ Projects** (Проекты Visual C++).

3. В окне списка **Templates** (Шаблоны) выделите значок **MFC Application** (Приложение MFC), в текстовое поле **Name** (Имя) введите имя приложения "MDI" и нажмите кнопку **OK**. Появится диалоговое окно **MFC Application Wizard** (Мастер создания приложений MFC).
4. Раскройте вкладку **Application Type** (Тип приложения).
5. Оставив переключатель в положении **Multiple documents** (Многооконные приложения), нажмите кнопку **Finish** (Готово). Мастер MFC Application Wizard создаст заготовку многооконного приложения, но при этом не будет открыто ни одного окна редактирования файлов.
6. Запустите приложение на исполнение. Для чего выберите команду меню **Debug | Start** (Отладка | Запуск) или нажмите клавишу <F5>. Появится окно **Microsoft Development Environment** (рис. 1.6), сообщающее, что текущая конфигурация проекта устарела и предлагающее перекомпилировать проект.

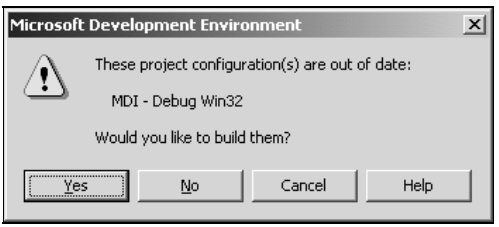


Рис. 1.6. Диалоговое окно **Microsoft Development Environment**

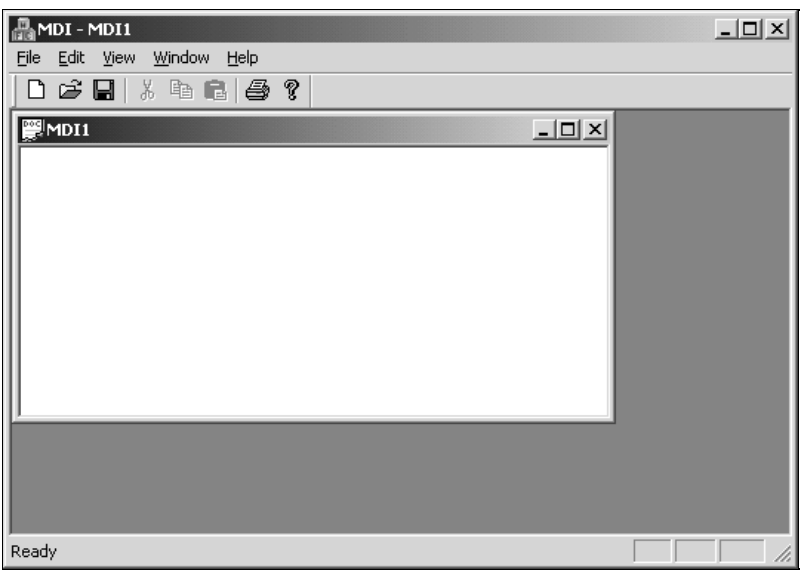


Рис. 1.7. Многооконное приложение

7. Нажмите кнопку **Yes** (Да). Visual C++ произведет трансляцию приложения и запустит его на исполнение. При этом на экране появится окно, изображенное на рис. 1.7.

Это приложение позволяет создавать новые окна, изменять их размер и уничтожать их. Оно может работать с меню и панелью инструментов, но не может выполнять никаких полезных действий. Структура этого проекта будет подробно рассмотрена в следующей главе.

Другие виды приложений MFC

На вкладке **Application Type** (Тип приложения) диалогового окна **MFC Application Wizard** (Мастер создания приложений MFC) остались нерассмотренными два положения переключателя: **Single document** (Однооконное приложение) и **Multiple top-level documents** (Несколько окон верхнего уровня).

Одноконное приложение является одним из базовых типов приложений Windows. Оно использует однооконный интерфейс приложения. Это приложение может работать только с одним документом и, по сути своей, занимает промежуточное положение между диалоговым приложением и многооконным приложением. Поскольку процесс его создания отличается от процесса создания многооконного приложения только установкой переключателя в соответствующее положение, то здесь будет приведен только внешний вид его окна (рис. 1.8).

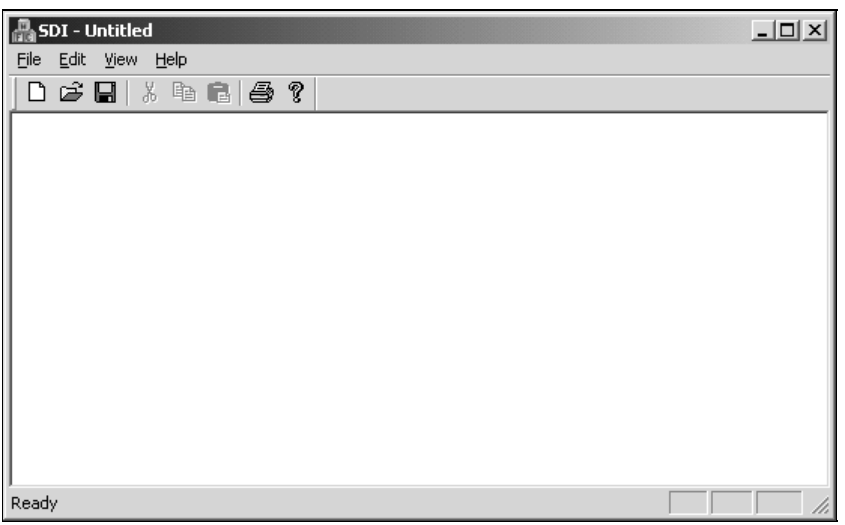


Рис. 1.8. Однооконное приложение

Данный тип приложения может использоваться для обработки конкретного типа документа, при этом предполагается, что не требуется использования других

документов. Подобные задачи, но без использования документов, решает диалоговое приложение.

Обычно обработка одного документа предполагает возможность одновременной работы со связанными с ним документами, например, для внесения в них соответствующих изменений по ходу работы или получения из них необходимой информации. Поэтому в Visual Studio.NET введен новый тип приложения: приложение с несколькими окнами верхнего уровня. Этот тип приложения, по сути, представляет собой несколько взаимосвязанных однооконных приложений. Первое из созданных окон является главным и его закрытие влечет за собой закрытие всех остальных окон. Остальные окна являются подчиненными и их закрытие не влечет за собой закрытие других окон. Окно, создаваемое из подчиненного окна, становится подчиненным окном главного окна и никак не зависит от окна, в котором было создано. При открытии файлов для них открываются свои подчиненные окна.

Единственным видимым отличием главного окна является то, что в его меню **File** (Файл) присутствует команда **Exit** (Выход). В то время как во всех остальных окнах она заменена командой **Close** (Закрыть).

Консольное приложение

Помимо приложений MFC, особое внимание следует обратить на консольное приложение. Раньше использование этого типа приложений можно было рассматривать как попытку завоевать симпатии приверженцев программирования в среде DOS, оставшихся без средств программирования. По мере перехода от версии к версии возможности заготовок консольных приложений, создаваемых средой программирования Visual C++, постоянно росли. Так было по версию 6.0. С появлением пакета Visual Studio.NET, который иногда называют Visual Studio 7.0, все изменилось. Хотя в нем и остались средства создания консольных приложений, средства их создания не обеспечивают того удобства работы, которое было присуще версии 6.0.

Консольные приложения создаются, как правило, в тех случаях, когда для приложения требуется создать простейшую оболочку, обладающую самым простым интерфейсом. Информация консольному приложению передается в аргументах командной строки и через стандартный поток ввода. Для вывода информации консольное приложение использует стандартный поток вывода. Поэтому одной из основных областей использования консольных приложений является создание внепроцессных серверов, приложение которых должно иметь возможность работать с аргументами командной строки, что несколько затруднено при использовании других типов приложений.

С точки зрения программиста, консольное приложение является обычной программой DOS, использующей все стандартные операторы, но позволяющей, однако, использовать всю доступную оперативную память. Этот тип приложения запускается в особом типе окна, называемом "Окно MS DOS", в такое окно

в среде Windows 95 можно выйти, выбрав последовательно **Пуск | Программы | Сеанс MS-DOS**.

При преобразовании программ, написанных под DOS, в консольное приложение, необходимо помнить, что MS-DOS является 16-разрядной операционной системой, а Windows 95, Windows 98 и Windows NT — 32-разрядные операционные системы. С точки зрения программы это означает, что в MS-DOS и Windows 3.*x* используются 16-разрядные указатели, а в Win32 — 32-разрядные. Для упрощения переноса программ, написанных для 16-разрядных операционных систем, в 32-разрядные операционные системы размер переменной типа `int` устанавливается равным размеру указателя. Поэтому в консольном приложении переменная типа `int` занимает 4 байта, а не 2 байта, как это имеет место в программах, написанных под DOS. В большинстве случаев это приводит исключительно к незначительному росту объема памяти, занимаемого программой, но в некоторых случаях, например при работе с файлами, необходимо помнить о различии форматов целых чисел в различных операционных системах.

Демонстрационное консольное приложение можно найти в папке Console на прилагаемой к данной книге дискете.

Чтобы самостоятельно создать заготовку консольного приложения с использованием мастера:

1. Закройте все открытые проекты и файлы и выберите команду **File | New | Project** (Файл | Новый | Проект) или нажмите кнопку **New Project** (Новый проект) на панели инструментов **Standard** (Стандартная). Появится диалоговое окно **New Project** (Новый проект).
2. В окне иерархического списка **Project Types** (Типы проектов) раскройте папку **Visual C++ Projects** (Проекты Visual C++).
3. В окне списка **Templates** (Шаблоны) выделите значок **Win32 Project** (Проект Win32), в текстовое поле **Name** (Имя) введите имя приложения "Console" и нажмите кнопку **OK**. Появится диалоговое окно **Win32 Application Wizard - Console** (Мастер создания приложения Win32), изображенное на рис. 1.9.
4. Раскройте вкладку **Application Settings** (Настройки приложения). Диалоговое окно **Win32 Application Wizard** (Мастер создания приложения Win32) примет вид, изображенный на рис. 1.10.
5. Установите переключатель **Application type** (Тип приложения) в положение **Console application** (Консольное приложение) и нажмите кнопку **Finish** (Готово). Мастер Win32 Application Wizard создаст заготовку консольного приложения.
6. Откройте окно **Solution Explorer** (Проводник решения) и дважды щелкните левой кнопкой мыши на имени файла Console.cpp. Откроется окно его редактирования, содержащее заготовку, созданную мастером Win32 Application Wizard.
7. Измените файл Console.cpp в соответствии с текстом листинга 1.1.

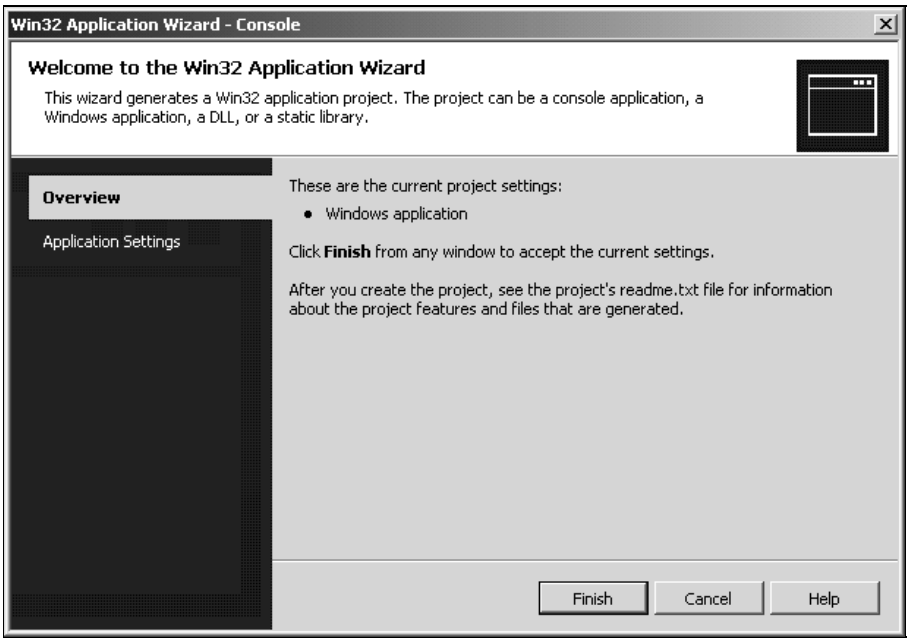


Рис. 1.9. Диалоговое окно **Win32 Application Wizard - Console**

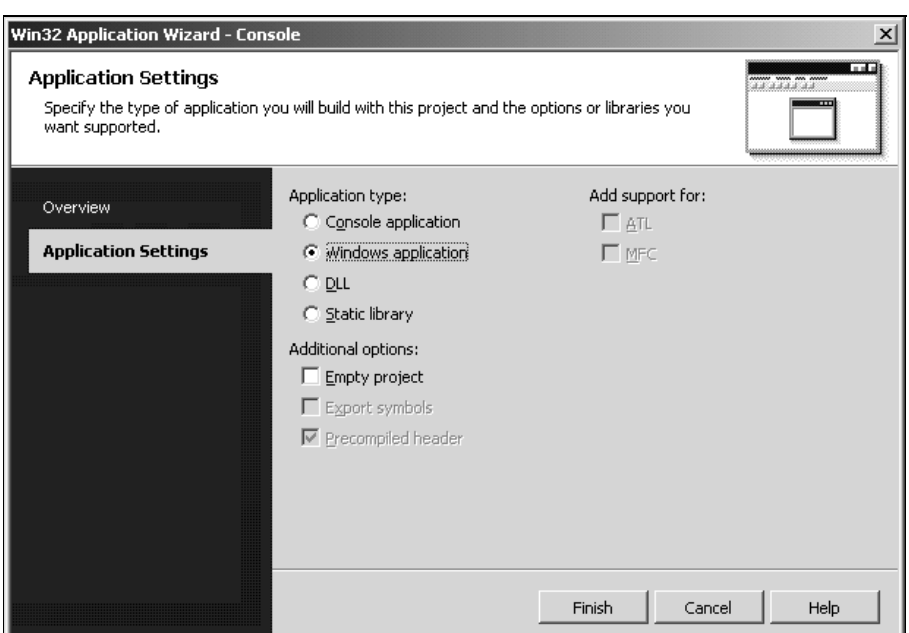


Рис. 1.10. Вкладка **Application Settings**

Листинг 1.1. Файл Console.cpp

```
// Файл Console.cpp содержит входную точку консольного приложения
//
/////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include <iostream.h>

int _tmain(int argc, _TCHAR* argv[])
{
  cout << "Are you ready to terminate?" << endl;
  char  szText[256];
  cin >> szText;
  return 0;
}
```

8. Нажмите клавишу <F5> и запустите приложение на исполнение. Появится окно приложения, изображенное на рис. 1.11.

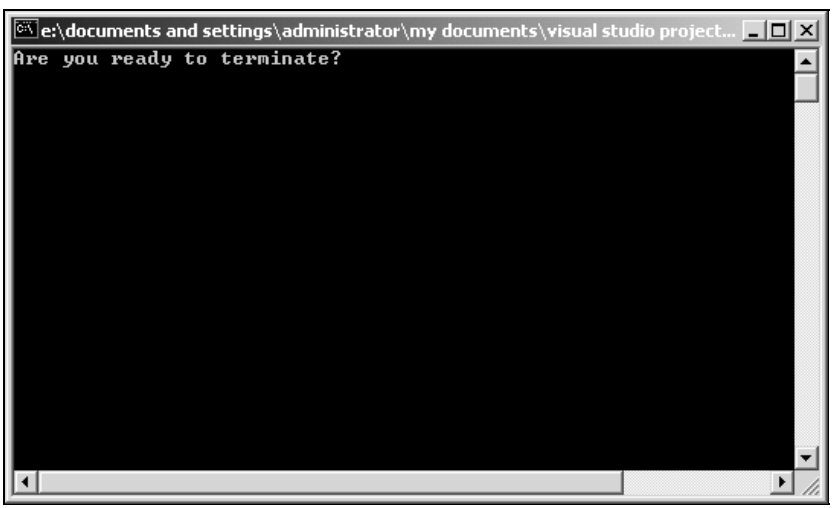


Рис. 1.11. Окно консольного приложения

9. Введите строку "Yes" и нажмите клавишу <Enter>. Консольное приложение завершит свою работу.

Это приложение имеет простейшую структуру: в нем с использованием стандартного потока вывода выводится текстовая строка. Чтобы приложение после вывода строки не прекратило свою работу, строка содержит вопрос и предполагает ввод пользователем строки текста. После чтения этой строки, без проверки ее содержимого, приложение завершает свою работу.

Обратите внимание на то, что при компиляции приложения появляется предупреждение, настоятельно не рекомендующее использование стандартных потоков для ввода и вывода информации.

Для того чтобы консольное приложение могло воспользоваться услугами библиотеки MFC, при его создании необходимо установить флажок **MFC** в группе **Add support for** (Добавить поддержку). Создаваемая при этом заготовка получается сложнее заготовки обычного консольного приложения, но не настолько сложной, чтобы в ней нельзя было разобраться.

Глава 2

Классы приложений, документов и представлений

В основе концепции программирования в среде Windows с использованием библиотеки классов MFC лежит концепция Документ/Представление. Эта концепция подразумевает, что в каждом приложении должны быть четко разделены функции манипулирования информацией и функции ее отображения.

Поэтому каждое приложение, построенное в соответствии с данной концепцией, должно включать в себя описание класса, производного от класса `CDocument`, на который возлагаются задачи хранения обрабатываемой информации в оперативной памяти, предоставление этой информации по внешним запросам, запись и чтение этой информации с диска. Это же приложение должно включать в себя описание класса, производного от класса `CView`, на который возлагаются задачи отображения этой информации на экране и реализация интерфейса пользователя. Кроме этих двух классов, приложение может содержать и другие полезные классы, но именно взаимодействие этих двух классов в рамках *шаблона документа* позволяет существенно упростить программирование многих стандартных операций по взаимодействию создаваемого приложения с операционной системой Windows.

Другим неотъемлемым компонентом любой программы, использующей библиотеку MFC, является класс приложения, производный от класса `CWinApp`. Методы данного класса отвечают за инициализацию, работу и правильное завершение всего приложения.

Класс документа

Рассмотрим класс документа в многооконном приложении, созданном в *главе 1*. Полный текст данного приложения содержится в папке MDI на прилагаемой к данной книге дискете. Если вы еще не создали это приложение, обратитесь к предыдущей главе и выполните все описанные в ней действия по созданию многооконного приложения.

Рассмотрим файлы, помещенные мастером MFC Class Wizard в заготовку многооконного приложения. В листинге 2.1 представлен файл MDIDoc.h, содержащий заголовок класса документа данного приложения. В этом и последующих листингах данной книги все комментарии переведены на русский язык.

Листинг 2.1. Файл MDIDoc.h

```
// Файл MDIDoc.h содержит интерфейс класса CMDIDoc
//

#pragma once

class CMDIDoc : public CDocument
{
protected: // Создается только при обеспечении живучести
  CMDIDoc();
  DECLARE_DYNCREATE(CMDIDoc)

// Атрибуты
public:

// Операции
public:

// Перегружаемые функции
  public:
  virtual BOOL OnNewDocument();
  virtual void Serialize(CArchive& ar);

// Реализация
public:
  virtual ~CMDIDoc();
#ifdef _DEBUG
  virtual void AssertValid() const;
  virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// Функции карты сообщений
protected:
  DECLARE_MESSAGE_MAP()
};
```

Данный класс является непосредственным потомком класса `CDocument`, динамически создаваемым в процессе исполнения программы. Он имеет единственный

защищенный конструктор. Это говорит о том, что непосредственное создание объекта данного класса программистом не предусматривается. Мастер создания приложений MFC включает в данный класс всего два метода для обработки информации и два метода для отладки. Кроме того, созданный класс имеет пустую карту сообщений. Обычно эта карта так и остается незаполненной, поскольку в соответствии с концепцией Документ/Представление все сообщения должен принимать и обрабатывать объект класса представления, а объект класса документа должен предоставлять ему для этого необходимые методы по прямым запросам. Обработка сообщений будет подробно рассмотрена в *главе 5*.

Для того чтобы подробнее ознакомиться с этими методами, рассмотрим файл реализации данного класса MDIDoc.cpp, приведенный в листинге 2.2.

Листинг 2.2. Файл MDIDoc.cpp

```
// Файл MDIDoc.cpp содержит реализацию класса CMDIDoc
//

#include "stdafx.h"
#include "MDI.h"

#include "MDIDoc.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif

// Класс CMDIDoc

IMPLEMENT_DYNCREATE(CMDIDoc, CDocument)

BEGIN_MESSAGE_MAP(CMDIDoc, CDocument)
END_MESSAGE_MAP()

// Конструктор и деструктор класса CMDIDoc

CMDIDoc::CMDIDoc()
{
  // ЧТО ДЕЛАТЬ: добавить сюда текст конструктора
}

CMDIDoc::~CMDIDoc()
{
}
```

```
BOOL CMDIDoc::OnNewDocument()
{
  if (!CDocument::OnNewDocument())
    return FALSE;
  // ЧТО ДЕЛАТЬ: добавить код повторной инициализации
  // (однооконное приложение будет повторно использовать
  // этот документ)
  return TRUE;
}

// Обеспечение живучести объекта класса CMDIDoc
void CMDIDoc::Serialize(CArchive& ar)
{
  if (ar.IsStoring())
  {
    // ЧТО ДЕЛАТЬ: добавить код для сохранения объекта
  }
  else
  {
    // ЧТО ДЕЛАТЬ: добавить код для загрузки объекта
  }
}

// Диагностические функции класса CMDIDoc

#ifdef _DEBUG
void CMDIDoc::AssertValid() const
{
  CDocument::AssertValid();
}

void CMDIDoc::Dump(CDumpContext& dc) const
{
  CDocument::Dump(dc);
}
#endif //_DEBUG

// Функции обработки сообщений класса CMDIDoc
```

Как видно из листинга 2.2, созданный класс имеет пустые конструктор и деструктор. Это связано с тем, что стандартные операции по созданию и уничтоже-

нию элементов данного класса производятся в конструкторе и деструкторе базового класса, а данных, которые необходимо инициализировать или уничтожать в конструкторе и деструкторе, этот класс пока не содержит. Отсутствие данных в классе объясняется тем, что приведенный листинг представляет собой только заготовку пользовательского класса, и, поэтому, не содержит каких-либо пользовательских данных.

Функция `CDocument::OnNewDocument` будет вызываться всякий раз, когда понадобится создать пустой документ, например, в результате выбора команды **File | New** (или **Файл | Создать**, если вы русифицировали меню вашего приложения). По умолчанию эта функция просто вызывает соответствующий метод базового класса. Поскольку созданное нами приложение использует только один тип документов, то в результате выполнения данной функции в приложении появится пустое окно документа.

Окно списка **New** (Создать) данного диалогового окна содержит краткие имена используемых в данном приложении документов. Чтобы выбрать нужный тип документа, выделите его имя в окне списка **New** (Создать) и нажмите кнопку **OK**. Появится пустое окно документа выбранного типа.

Функция `CObject::Serialize` осуществляет чтение объекта из архива и запись его в архив. Единственным аргументом данной функции является объект класса `CArchive`, осуществляющий запись информации, содержащейся в объекте, в файл архива и последующее чтение этой информации из файла архива в создаваемый объект. Данная функция и передаваемый ей в качестве аргумента объект класса `CArchive` используются как для записи информации в файл, так и для чтения ее из файла, поэтому, для того, чтобы определить, какая операция будет производиться в настоящий момент, используется функция `CArchive::IsStoring`, возвращающая значение `TRUE`, если производится сохранение объекта, и `FALSE`, если производится его инициализация. Поскольку данный класс документа не содержит данных, то в нем отсутствуют операции по их сохранению в архиве и извлечению из него, располагаемые в различных ветвях условного оператора.

Функция `CObject::AssertValid` служит для целей отладки и производит проверку данного объекта на допустимость значений хранимых в нем величин.

Функция `CObject::Dump` также служит для целей отладки и передает содержимое вашего объекта в объект класса `CDumpContext`.

Поскольку функции `AssertValid` и `Dump` используются только в процессе отладки, их объявление в файле заголовка и их текст в файле реализации помещаются между операторами условной трансляции `#ifdef _DEBUG` и `#endif`. Завершает данную программу строка комментариев, указывающая на то, что за ней должны располагаться методы, обрабатывающие данные, хранящиеся в данном классе. Отладочные функции приложения будут более подробно рассмотрены в *главе 14*.

Класс представления

Теперь перейдем к рассмотрению класса представления заготовки многооконного приложения. В листинге 2.3 представлен файл MDIView.h, содержащий заголовок данного класса.

Листинг 2.3. Файл MDIView.h

```
// Файл MDIView.h содержит интерфейс класса CMDIView
//
#pragma once

class CMDIView : public CView
{
protected: // Создается только при обеспечении живучести
  CMDIView();
  DECLARE_DYNCREATE(CMDIView)

// Атрибуты
public:
  CMDIDoc* GetDocument() const;

// Операции
public:

// Перегружаемые функции
  public:
  virtual void OnDraw(CDC* pDC);  // Перегружается для
                                  // вывода на экран объекта
virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
protected:
  virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
  virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);
  virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);

// Реализация
public:
  virtual ~CMDIView();
#ifdef _DEBUG
  virtual void AssertValid() const;
  virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// Функции карты сообщений
protected:
```

```
  DECLARE_MESSAGE_MAP()
};

#ifndef _DEBUG  // Отладочная версия расположена в файле MDIView.cpp
inline CMDIDoc* CMDIView::GetDocument() const
  { return reinterpret_cast<CMDIDoc*>(m_pDocument); }
#endif
```

Класс `CMDIView` является непосредственным потомком класса `CView`, динамически создаваемым в процессе исполнения программы. Так же, как и класс `CMDIDoc`, он имеет единственный защищенный конструктор, что говорит о том, что непосредственное создание объекта данного класса программистом не предусматривается. Заголовок данного класса, так же как и заголовок рассмотренного выше класса документа, имеет пустую карту сообщений. Однако в реальном приложении эта карта практически никогда не остается пустой.

Для того чтобы подробнее ознакомиться с методами данного класса, рассмотрим файл его реализации MDIView.cpp, приведенный в листинге 2.4.

Листинг 2.4. Файл MDIView.cpp

```
// Файл MDIView.cpp содержит реализацию класса CMDIView
//

#include "stdafx.h"
#include "MDI.h"

#include "MDIDoc.h"
#include "MDIView.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif

// Класс CMDIView

IMPLEMENT_DYNCREATE(CMDIView, CView)

BEGIN_MESSAGE_MAP(CMDIView, CView)
  // 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()

// Конструктор и деструктор класса CMDIView

CMDIView::CMDIView()
{
  // ЧТО ДЕЛАТЬ: добавить сюда текст конструктора
}

CMDIView::~CMDIView()
{
}

BOOL CMDIView::PreCreateWindow(CREATESTRUCT& cs)
{
  // ЧТО ДЕЛАТЬ: Изменить класс окна или его стили в переменной
  // CREATESTRUCT cs
  return CView::PreCreateWindow(cs);
}

// Вывод на экран объекта класса CMDIView

void CMDIView::OnDraw(CDC* /*pDC*/)
{
  CMDIDoc* pDoc = GetDocument();
  ASSERT_VALID(pDoc);

  // ЧТО ДЕЛАТЬ: добавить код для вывода объекта на экран
}

// Печать объекта класса CMDIView
BOOL CMDIView::OnPreparePrinting(CPrintInfo* pInfo)
{
  // Подготовка по умолчанию
  return DoPreparePrinting(pInfo);
}

void CMDIView::OnBeginPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
```

```
  // ЧТО ДЕЛАТЬ: добавить дополнительную инициализацию перед печатью
}

void CMDIView::OnEndPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
  // ЧТО ДЕЛАТЬ: уничтожить объекты после печати
}

// Диагностические функции класса CMDIView
#ifdef _DEBUG
void CMDIView::AssertValid() const
{
  CView::AssertValid();
}

void CMDIView::Dump(CDumpContext& dc) const
{
  CView::Dump(dc);
}

CMDIDoc* CMDIView::GetDocument() const  // Распространяемая версия
                              // данной функции является встроенной
{
  ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CMDIDoc)));
  return (CMDIDoc*)m_pDocument;
}
#endif //_DEBUG

// Функции обработки сообщений класса CMDIView
```

Как видно из листинга 2.4, созданный класс имеет пустые конструктор и деструктор. Это связано с тем, что стандартные операции по созданию и уничтожению элементов данного класса производятся в конструкторе и деструкторе базового класса, а никаких данных, характерных только для данного окна представления заготовка пользовательского класса, не содержит.

Карта сообщений файла реализации класса `CMDIView` содержит макросы обработки стандартных сообщений печати `ID_FILE_PRINT`, `ID_FILE_PRINT_DIRECT` и `ID_FILE_PRINT_PREVIEW`. Для их обработки используются методы базового класса `CView::OnFilePrint` и `CView::OnFilePrintPreview`. Эти функции являются защищенными членами класса, поэтому информация по ним отсутствует в справочной системе.

Функция `CWnd::PreCreateWindow` вызывается приложением перед созданием окна, связанного с данным объектом класса `CWnd`. В этом случае единственным оператором перегруженной функции `PreCreateWindow` является вызов метода базового класса.

Функция `CView::OnDraw` вызывается приложением для отображения документа на экране при печати документа и при его предварительном просмотре перед печатью. Для доступа к *интерфейсу графических устройств* (GDI) функция `OnDraw` использует указатель на объект класса `CDC`. Работа с данной функцией при простом выводе информации на экран описана в *главе 6*, а ее работа при прокрутке изображения в окне и печати документов — в *главе 10*.

Текст функции `OnDraw`, содержащийся в листинге 2.4, состоит из двух операторов. В первом операторе создается указатель на объект класса `CMDIDoc`, для чего используется функция `CView::GetDocument`. Эта функция занимает центральное место в концепции Документ/Представление, поскольку именно она связывает эти два понятия в единое целое. Эта функция возвращает указатель на объект класса `CDocument`, ассоциированный с данным представлением. Если с данным представлением не ассоциировано никакого документа, данная функция возвращает ноль. Полученный указатель позволяет объекту представления получить доступ к данным и методам объекта документа.

Второй оператор функции `OnDraw` является макросом `ASSERT_VALID`, осуществляющим проверку допустимости значений параметров в полученном объекте документа. При компиляции окончательной версии приложения, использующего библиотеку MFC, данный макрос не производит никаких операций. Однако в отладочной версии этой библиотеки данный макрос проверяет свой аргумент на ненулевое значение и вызывает функцию `AssertValid` передаваемого объекта. Если хотя бы одна из этих проверок дает отрицательный результат, то на экране появляется сообщение, аналогичное сообщению, выдаваемому макросом `ASSERT` и функцией `AssertValid` класса документа. Более подробно этот макрос описан в *главе 14*.

Функции `CView::OnPreparePrinting`, `CView::OnBeginPrinting` и `CView::OnEndPrinting` используются при печати документа и не представляют особого интереса на данном этапе рассмотрения. Работа с этими функциями подробно описана в *главе 10*.

Класс `CMDIView`, так же, как и класс `CMDIDoc` имеет отладочные функции `CObject::AssertValid` и `CObject::Dump`, подробное описание которых дано на дискете. Кроме них, отладочная секция программы содержит реализацию функции `GetDocument`. Эта функция не является только отладочной, просто для нее предусмотрено две версии: отладочная, содержащая необходимые проверки, и окончательная, которая с целью повышения быстродействия программы сделана встраиваемой (inline).

Первый оператор отладочной версии функции `GetDocument` проверяет, является ли объект, на который указывает хранящийся в данном классе указатель, объектом класса документа, ассоциированного с данным классом представления.

В случае положительного исхода проверки второй оператор возвращает этот указатель на объект документа.

Завершает данную программу строка комментариев, указывающая на то, что за ней должны располагаться функции обработки пользовательских сообщений, поступающих в объект данного класса. Поскольку данный класс является заготовкой, он не содержит функций обработки пользовательских сообщений.

Класс приложения

При описании класса представления несколько раз упоминалось об ассоциации класса представления и класса документа. Где и каким образом происходит эта ассоциация? Для того чтобы ответить на этот и ряд других вопросов, необходимо рассмотреть класс приложения.

Поскольку файлы заголовков классов демонстрационных приложений, как правило, создаются мастером создания приложений соответствующего типа и заполняются автоматически с использованием средств, предоставляемых средой программирования, в дальнейшем их тексты не будут приводиться в данной книге без особой необходимости. Полные тексты приложений, включающие тексты файлов заголовков, содержатся на прилагаемой к данной книге дискете. Поэтому в дальнейшем при рассмотрении приложений будут даны только тексты файлов реализации.

Файл MDI.cpp, являющийся файлом реализации класса приложения `CMDIApp`, приведен в листинге 2.5.

Листинг 2.5. Файл MDI.cpp

```
// Файл MDI.cpp определяет поведение класса приложения.
//

#include "stdafx.h"
#include "MDI.h"
#include "MainFrm.h"

#include "ChildFrm.h"
#include "MDIDoc.h"
#include "MDIView.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif

// Класс CMDIApp
```

```
BEGIN_MESSAGE_MAP(CMDIApp, CWinApp)
  ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
  // Стандартные команды для работы с файлами документов
  ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
  ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
  // Стандартные команды установки режима печати
  ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()

// Конструктор класса CMDIApp

CMDIApp::CMDIApp()
{
  // ЧТО ДЕЛАТЬ: добавить сюда текст конструктора.
  // Все процедуры инициализации должны располагаться в функции
  // InitInstance
}

// Единственный и неповторимый объект класса CMDIApp
CMDIApp theApp;

// Инициализация класса CMDIApp
BOOL CMDIApp::InitInstance()
{
  CWinApp::InitInstance();

  // Инициализация библиотек OLE
  if (!AfxOleInit())
  {
    AfxMessageBox(IDP_OLE_INIT_FAILED);
    return FALSE;
  }
  AfxEnableControlContainer();
  // Стандартная инициализация/
  // Если вы не используете этих свойств и хотите уменьшить размер
  // исполняемого файла приложения, удалите ненужные вам процедуры
  // инициализации.
  // Измените ключ системного реестра, по которому будут храниться
  // параметры приложения.
  // ЧТО ДЕЛАТЬ: необходимо поместить в эту строку что-нибудь
  // соответствующее, например, название вашей компании или организации
```

```
  SetRegistryKey(_T("Local AppWizard-Generated Applications"));
  LoadStdProfileSettings(4);  // Загрузка стандартного файла INI
                              // (включая список последних открывавшихся файлов)
  // Регистрация шаблона документов приложения. Шаблоны документов
  // обеспечивают взаимодействие документов, рамок окна и представлений
  CMultiDocTemplate* pDocTemplate;
  pDocTemplate = new CMultiDocTemplate(IDR_MDITYPE,
    RUNTIME_CLASS(CMDIDoc),
    RUNTIME_CLASS(CChildFrame),  // Пользовательская рамка окна
                                 // многооконного приложения
    RUNTIME_CLASS(CMDIView));
  AddDocTemplate(pDocTemplate);
  // Создание рамки окна многооконного приложения
  CMainFrame* pMainFrame = new CMainFrame;
  if (!pMainFrame->LoadFrame(IDR_MAINFRAME))
    return FALSE;
  m_pMainWnd = pMainFrame;
  // Если приложение может работать с файлами, перетащенными в ее окно,
  // в приложениях MDI сразу же после установки значения m_pMainWnd
  // следует вызвать функцию DragAcceptFiles.
  // Анализ аргументов командной строки, динамический обмен данными
  // открытие файла
  CCommandLineInfo cmdInfo;
  ParseCommandLine(cmdInfo);
  // Диспетчеризация команд в командной строке. Если приложение было
  // запущено с ключами /RegServer, /Register, /Unregserver или
  // /Unregister, функция должна возвращать значение FALSE.
  if (!ProcessShellCommand(cmdInfo))
    return FALSE;
  // Вывод на экран главного окна приложения после его инициализации

  pMainFrame->ShowWindow(m_nCmdShow);
  pMainFrame->UpdateWindow();
  return TRUE;
}

// Класс диалогового окна CAboutDlg

class CAboutDlg : public CDialog
{
```

```
public:
  CAboutDlg();

// Данные диалогового окна
  enum { IDD = IDD_ABOUTBOX };

protected:
  virtual void DoDataExchange(CDataExchange* pDX); // Поддержка DDX/DDV

// Реализация
protected:
  DECLARE_MESSAGE_MAP()
};

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

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

BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
END_MESSAGE_MAP()

// Вывод диалогового окна

void CMDIApp::OnAppAbout()
{
  CAboutDlg aboutDlg;
  aboutDlg.DoModal();
}

// Функции обработки сообщений класса CMDIApp
```

Класс `CMDIApp` является потомком класса `CWinApp`, который, в свою очередь, является потомком класса `CWinThread`. Поэтому можно сказать, что данный класс является классом главного потока программы. Наличие потоков является главной отличительной особенностью 32-разрядных систем Windows от их 16-разрядных предшественников, которые являлись "согласующей" многозадачной

средой без квантования времени. Это означает, что управление у задачи не отнимается по истечении определенного кванта времени. Для обеспечения возможности параллельной работы нескольких приложений в 16-разрядных системах Windows приходилось генерировать особое прерывание от таймера, которое приостанавливало выполнение одного приложения и передавало управление другому. Работа с таймером полностью возлагалась на пользователя.

В "вытесняющих" средах, к которым относятся и 32-разрядные системы Windows, операционная система выделяет каждому потоку квант времени, по истечении которого его выполнение прерывается и операционная система решает, какому потоку следует передать управление. Поток представляет собой некоторый неделимый процесс, выполняемый в оперативной памяти компьютера, и имеющий определенный уровень приоритета, определяющий порядок предоставления ему квантов времени. Поток может взаимодействовать с другими потоками и порождать их. Работа с потоками подробно описана в *главе 12*.

Каждое приложение, использующее MFC, может содержать только один объект класса, производного от `CWinApp`. Этот объект создается на этапе создания глобальных объектов и уже существует к моменту вызова функции `WinMain`, содержащейся в библиотеке MFC.

Функция `InitInstance`, принадлежащая данному классу, служит для создания объекта класса главного окна приложения.

Дизайн элементов управления и системный реестр

Первым оператором функции `InitInstance` является вызов метода базового класса. После чего производится вызов функции `AfxOleInit`, инициализирующей систему OLE в данном приложении. Далее следует вызов функции `AfxEnableControlContainer`, обеспечивающей данному приложению поддержку элементов управления OLE.

Вызов функции `SetRegistryKey` позволяет сохранять параметры начальной установки вашего приложения в системном реестре, а не в файлах с расширением ini, как это имело место в старых версиях Windows. Кроме того, вызов данной функции приводит к сохранению в системном реестре списка недавно использованных файлов. Окно приложения с раскрытым меню **File** (Файл), в котором содержится список недавно использованных файлов, приведено на рис. 2.1.

Функция `LoadStdProfileSettings` служит для чтения из системного реестра параметров начальной установки данного приложения. Работа с системным реестром описана в *главе 7*.

В следующем разделе рассматриваются операции, ради которых здесь и приведено столь подробное описание работы функции `InitInstance` — создание шаблона документа.

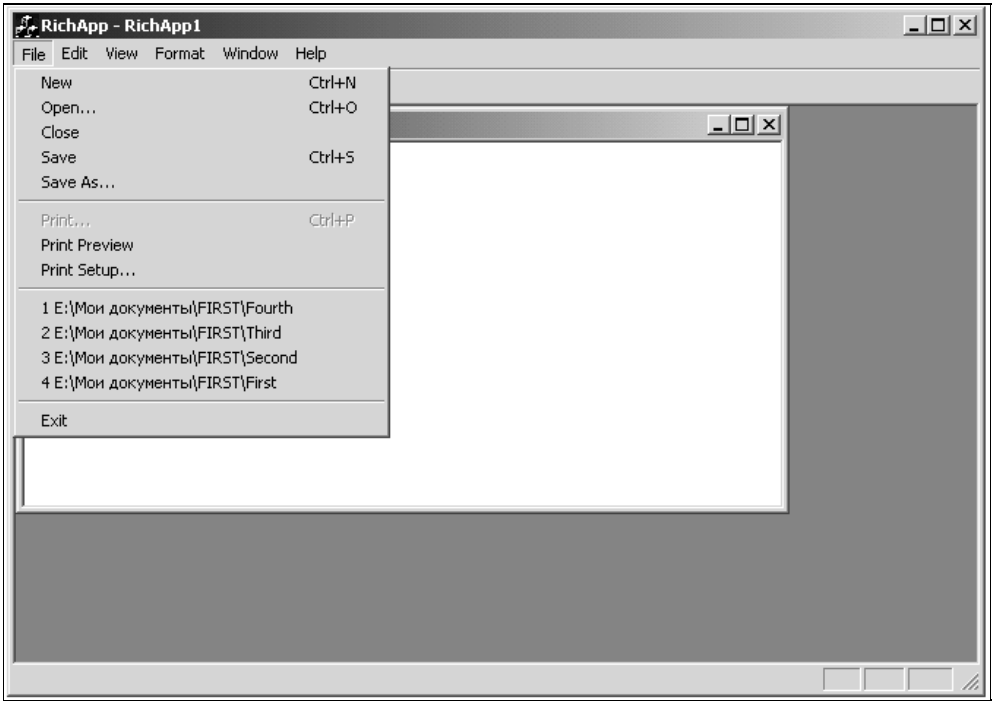


Рис. 2.1. Список недавно использованных файлов

Шаблон документа

В начале этого раздела был поставлен вопрос о том, каким образом производится связывание класса представления и класса документа. Эта процедура осуществляется в шаблоне документа. В этом шаблоне объединены следующие объекты:

- идентификатор ресурса меню — в данном случае `IDR_MDITYPE`;
- класс документа — в данном случае `CMDIDoc`;
- класс окна — в случае многооконного приложения это `CChildFrame`, а в случае однооконного приложения или приложения с несколькими окнами верхнего уровня — `CMainFraim`;
- класс представления — в данном случае `CMDIView`.

Описание шаблона документа начинается с получения указателя на объект класса `CMultiDocTemplate` (в случае однооконного приложения это будет объект класса `CSingleDocTemplate`). По адресу, содержащемуся в полученном указателе, записывается создаваемый объект шаблона документа. Обратите внимание на то, что в качестве параметра в макросе `RUNTIME_CLASS` выступает не объект класса или указатель на него, а чистое имя класса. Это позволяет создавать объекты данных классов в качестве внутренних переменных класса шаблона документа. Именно

объединение классов документа и представления в одном объекте класса шаблона и позволяет установить между ними однозначное соответствие.

Кроме имен классов документа и представления шаблон документа содержит идентификатор ресурса меню и класс окна. Включение информации о меню в класс шаблона документа представляется абсолютно логичным, поскольку каждый документ имеет характерную только для него структуру данных и методов их обработки. Поэтому ясно, что меню, оптимизированное для целей обработки текстовой информации, будет достаточно сложно использовать для редактирования изображений. Создание же универсального меню приведет к неизбежной путанице и существенно усложнит работу с приложением. Поэтому при работе с каждым типом документов в главное окно загружается соответствующее меню, оптимизированное для работы именно с этим типом документов.

Необходимость внесения в шаблон документа информации об окне данного приложения не казалась столь очевидной для Visual C++ версии 6.0, поскольку в ней каждому из классов шаблонов (`CMultiDocTemplate` или `CSingleDocTemplate`) четко соответствовал свой тип окна (`CChildFrame` или `CMainFrame`). Оказалось, что у Microsoft были далеко идущие планы: в версии 7.0 введен новый тип приложения (с несколькими окнами верхнего уровня). Теперь классу шаблона `CMultiDocTemplate` может соответствовать как класс дочернего окна `CChildFrame` для многооконного приложения, так и класс главного окна `CMainFraim` для приложения с несколькими окнами верхнего уровня.

Для того чтобы макрос `RUNTIME_CLASS` нормально работал, нужно соблюдать некоторые соглашения относительно объявления включенных в него классов. В заголовке класса один из его конструкторов (обычно закрытый) объявляется с использованием макроса `DECLARE_DYNACREATE`, а в файле реализации данного класса должен присутствовать макрос `IMPLEMENT_DYNACREATE` для данного конструктора.

Завершает процедуру создания шаблона документа вызов функции `CWinApp::AddDocTemplate`, единственным параметром которой является указатель на только что созданный объект класса шаблона документа.

В данном приложении был создан всего один шаблон документа. При необходимости работать с несколькими типами документов программист должен создать соответствующие классы документа и представления, разработать соответствующее меню, имеющее свой идентификатор, и самостоятельно написать процедуру создания шаблона документа, аналогичную описанной выше. Естественно, каждый объект класса шаблона документа должен иметь уникальный идентификатор.

Создание окон

Когда шаблон документа готов, создается динамический объект класса главного окна приложения `CMainFrame`. Объект класса, производный от `CFrameWnd`, создается в два этапа. Сначала вызывается конструктор, создающий объект класса `CFrameWnd`, а затем вызывается функция `CFrameWnd::LoadFrame`, загружающая

главное окно приложения и связанные с ним ресурсы, а также связывающая загруженное окно с объектом класса, производного от класса `CMainFrame`.

В ресурсе, идентификатор которого передается функции в качестве параметра, определены меню, таблица назначения клавиш, значки и строковый ресурс заголовка окна. Для самостоятельного задания всех параметров создания окна вместо функции `LoadFrame` следует использовать функцию `CFrameWnd::Create`. Приложение вызывает функцию `LoadFrame` при создании главного окна программы с использованием объекта класса шаблона документа.

Полученный указатель на объект главного окна приложения присваивается идентификатору `m_pMainWnd`, являющемуся членом класса приложения. Значение этой переменной может быть использовано для непосредственного доступа к объекту главного окна приложения.

Следующим шагом по инициализации приложения является создание объекта класса `CCommandLineInfo`, используемого для выделения параметров инициализации приложения из командной строки. Созданный объект передается в качестве аргумента функции `ParseCommandLine`. Для каждого элемента командной строки функция `ParseCommandLine` вызывает функцию `CCommandLineInfo::ParseParam`, запоминающую его значение в объекте `CCommandLineInfo`. Этот объект затем передается функции `ProcessShellCommand`, производящей действия, определяемые значениями аргументов и флагов командной строки. Функция `ProcessShellCommand` возвращает ненулевое значение, если процесс завершился нормально, и ноль — в противном случае.

На вызове этой функции следует остановиться особо. Разработчики из Microsoft не предполагали, что с помощью Visual C++ будут разрабатываться какие-либо иные программы, кроме текстовых редакторов, поэтому предусмотрели при запуске многооконного приложения вывод на экран пустого окна. Целесообразность данного подхода сомнительна даже в текстовом редакторе, поскольку в большинстве случаев пользователь не начинает создание нового документа, а занимается редактированием старого. Однако в данном случае еще можно понять, зачем на экране появилось пустое окно. Но, представьте себе, что пустое окно появляется в приложении, созданном для обработки и анализа различных типов сигналов, каждый из которых представлен своим типом документа. Окно какого типа документа должно выводиться при запуске приложения? И что должен делать пользователь с пустым окном? Не рисовать же в нем вручную спектр сигнала. Поэтому хорошим тоном в серьезных приложениях считается отсутствие каких-либо окон при запуске приложения на исполнение.

Но мастер создания приложений MFC создает приложение, выводящее пустое окно на экран. Что же делать? Наиболее простым решением является уничтожение условного оператора, содержащего вызов функции `ProcessShellCommand`, или ремаркирование его, если вам не хочется удалять операторы из заготовки приложения. При этом не следует забывать, что условный оператор включает в себя не только сам оператор, но и оператор `return`, выполнение которого зависит от величины, возвращаемой функцией `ProcessShellCommand`. В этом случае

программа не будет обрабатывать командную строку, но большинство приложений Windows ее и не используют.

Функция `CWnd::ShowWindow` устанавливает режим отображения окна, например, стандартный размер отображаемого окна, окно, развернутое на весь экран или отображаемое как пиктограмма. При вызове этой функции в процедуре `InitInstance` ее аргументом является `CWinApp::m_nCmdShow`. При последующих ее вызовах в качестве аргумента необходимо непосредственно указывать режимы отображения окна, для обозначения которых существуют предопределенные величины.

Последним исполнительным оператором процедуры `InitInstance` является вызов функции `CWnd::UpdateWindow`, производящей перерисовку рабочей области окна путем посылки сообщения `WM_PAINT` в том случае, если эта область не пуста. Данная функция осуществляет непосредственную посылку сообщения `WM_PAINT`, не помещая его в очередь приложений.

Если программа дошла до этой точки, то инициализация приложения считается успешно завершенной, поэтому функция возвращает значение `TRUE`.

Кроме описания класса приложения данный файл содержит также описание класса диалогового окна справки о программе. Поскольку это диалоговое окно будет рассмотрено в *главе 13*, здесь мы оставим данный фрагмент файла без комментариев.

Работа с несколькими типами документов

Мастер MFC Application Wizard создает заготовку приложения, обрабатывающего документы одного типа. При создании реальных приложений может потребоваться работать с различными типами документов в одном приложении. Рассмотрим процедуру включения второго типа документа в приложение MDI.

Чтобы включить в приложение MDI обработку второго типа документа:

1. Выберите команду меню **File | Open Solution** (Файл | Открыть решение). Появится диалоговое окно **Open Solution** (Открыть решение), изображенное на рис. 2.2.
2. В окне списка раскройте папку **MDI** и дважды щелкните левой кнопкой мыши на значке **MDI**. В среду программирования будет загружено соответствующее приложение.
3. Выберите команду меню **View | Class View** (Вид | Просмотр классов) или нажмите комбинацию клавиш <Ctrl>+<Shift>+<C>. Откроется окно **Class View - MDI** (Просмотр классов), изображенное на рис. 2.3.
4. В окне **Class View** (Просмотр классов) щелкните правой кнопкой мыши на папке **MDI** и выберите в появившемся контекстном меню команду **Add | Add Class** (Добавить | Добавить класс). Появится диалоговое окно **Add Class - MDI** (Добавить класс), изображенное на рис. 2.4.

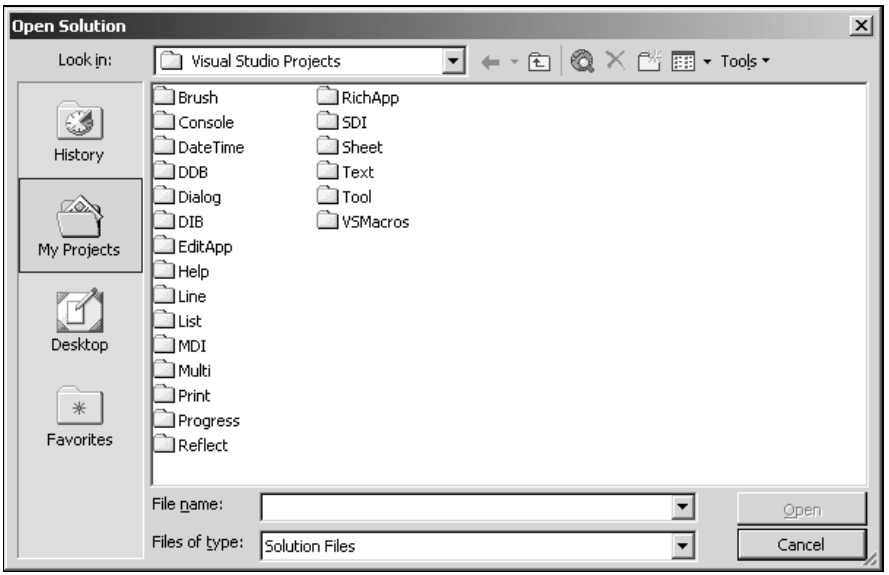


Рис. 2.2. Диалоговое окно **Open Solution**

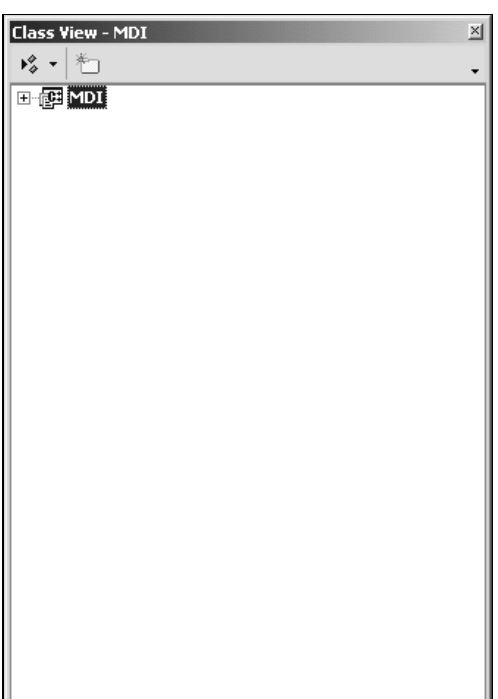


Рис. 2.3. Окно **Class View - MDI**

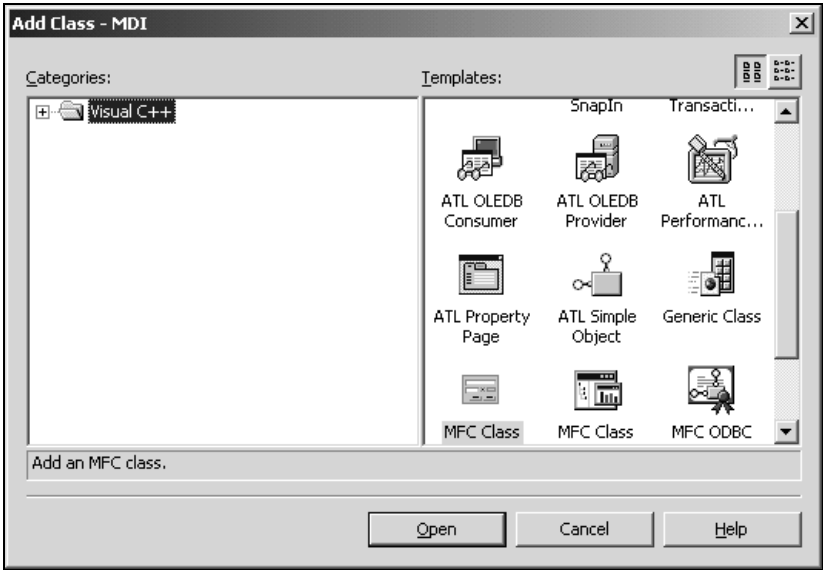


Рис. 2.4. Диалоговое окно **Add Class - MDI**

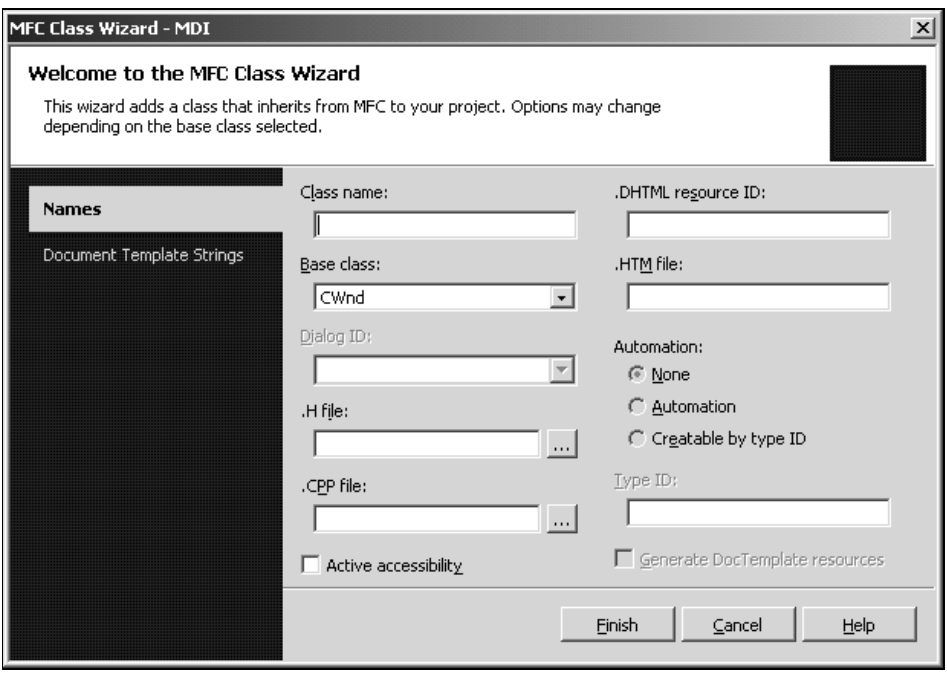


Рис. 2.5. Диалоговое окно **MFC Class Wizard - MDI**

5. В окне **Templates** (Шаблоны) выделите значок **MFC Class** (Класс MFC) и нажмите кнопку **Open** (Открыть). Появится диалоговое окно **MFC Class Wizard - MDI** (Мастер создания классов MFC), изображенное на рис. 2.5.
6. Введите в текстовое поле **Class name** (Имя класса) имя класса `CMyDoc`, выберите в раскрывающемся списке **Base class** (Базовый класс) имя базового класса `CDocument` и нажмите кнопку **Finish** (Готово). В приложение будет включен новый класс документа.
7. Повторите п.п. 4—6 для создания класса `CMyView`, производного от класса `CView`.
8. В окне **Class View** (Просмотр класса) раскройте папку **MDI**, а в ней раскройте папку `CMDIApp`, как это показано на рис. 2.6, и дважды щелкните левой кнопкой мыши на имени функции `InitInstance`. Откроется окно редактирования файла MDI.cpp, а текстовый курсор будет установлен в заготовке данной функции.

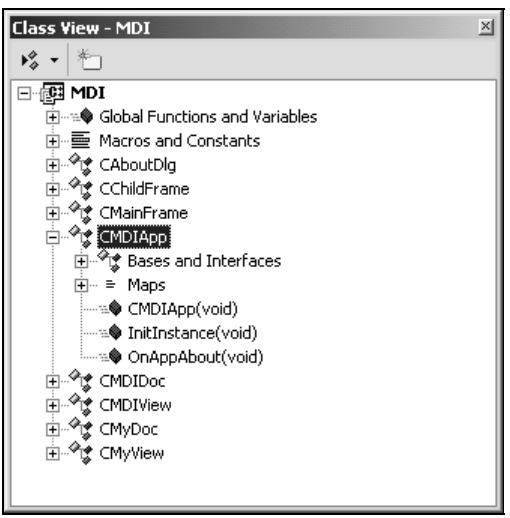


Рис. 2.6. Раскрытая папка **MDI**

9. Измените функцию `InitInstance` в соответствии с текстом листинга 2.6.

Листинг 2.6. Функция `InitInstance`

```
// Инициализация класса CMDIApp

BOOL CMDIApp::InitInstance()
{
  CWinApp::InitInstance();
```

```
// Инициализация библиотек OLE
if (!AfxOleInit())
{
  AfxMessageBox(IDP_OLE_INIT_FAILED);
  return FALSE;
}
AfxEnableControlContainer();
// Стандартная инициализация
// Если вы не используете этих свойств и хотите уменьшить размер
// исполняемого файла приложения, удалите ненужные вам процедуры
// инициализации.
// Измените ключ системного реестра, по которому будут храниться
// параметры приложения.
// ЧТО ДЕЛАТЬ: необходимо поместить в эту строку что-нибудь
// соответствующее, например, название вашей компании
// или организации
SetRegistryKey(_T("Local AppWizard-Generated Applications"));
LoadStdProfileSettings(4);  // Загрузка стандартного файла INI
             // (включая список последних открывавшихся файлов)
// Регистрация шаблона документов приложения. Шаблоны документов
// обеспечивают взаимодействие документов, рамок окна и представлений
CMultiDocTemplate* pDocTemplate;
pDocTemplate = new CMultiDocTemplate(IDR_MDITYPE,
  RUNTIME_CLASS(CMDIDoc),
  RUNTIME_CLASS(CChildFrame),    // Пользовательская рамка окна
                                 // многооконного приложения
  RUNTIME_CLASS(CMDIView));
AddDocTemplate(pDocTemplate);


CMultiDocTemplate* pMyDocTemplate;
pMyDocTemplate = new CMultiDocTemplate(IDR_MYTYPE,
  RUNTIME_CLASS(CMyDoc),
  RUNTIME_CLASS(CChildFrame),    // Пользовательская рамка окна
                                 // многооконного приложения
  RUNTIME_CLASS(CMyView));
AddDocTemplate(pMyDocTemplate);
// Создание рамки окна многооконного приложения
CMainFrame* pMainFrame = new CMainFrame;
```

```
  if (!pMainFrame->LoadFrame(IDR_MAINFRAME))
    return FALSE;
  m_pMainWnd = pMainFrame;
  // Если приложение может работать с файлами, перетащенными в ее окно,
  // в приложениях MDI сразу же после установки значения m_pMainWnd
  // следует вызвать функцию DragAcceptFiles.
  // Анализ аргументов командной строки, динамический обмен данными
  // открытие файла
  CCommandLineInfo cmdInfo;
  ParseCommandLine(cmdInfo);
  // Диспетчеризация команд в командной строке. Если приложение было
  // запущено с ключами /RegServer, /Register, /Unregserver или
  // /Unregister, функция должна возвращать значение FALSE.
  if (!ProcessShellCommand(cmdInfo))
    return FALSE;
  // Вывод на экран главного окна приложения после его инициализации
  pMainFrame->ShowWindow(m_nCmdShow);
  pMainFrame->UpdateWindow();
  return TRUE;
}
```

10. В начале файла MDI.cpp после строки `#include "MDIView.h"` вставьте следующий текст:

    ```
    #include "MyDoc.h"
    #include "MyView.h"
    ```

11. В окне **Class View** (Просмотр класса) дважды щелкните левой кнопкой мыши на папке `CMyView`. Откроется окно редактирования файла MyView.h.
12. Поместите в этот файл текст листинга 2.7.

Листинг 2.7 Файл MyView.h

```
#pragma once

#include "MyDoc.h"

// Класс представления CMyView
class CMyView : public CView
{
  DECLARE_DYNCREATE(CMyView)
```

```
protected:
  CMyView();  // Защищенный конструктор,
              // используемый при динамическом создании класса
  virtual ~CMyView();

// Атрибуты
public:
  CMyDoc* GetDocument() const;

public:
  virtual void OnDraw(CDC* pDC);  // Перегружается для вывода информации
#ifdef _DEBUG
  virtual void AssertValid() const;
  virtual void Dump(CDumpContext& dc) const;
#endif

protected:
  DECLARE_MESSAGE_MAP()
};

#ifndef _DEBUG     // Отладочная версия в файле MyView.cpp
inline CMyDoc* CMyView::GetDocument() const
   { return (CMyDoc*)m_pDocument; }
#endif
```

13. В окне **Class View** (Просмотр класса) раскройте папку `CMyView` и дважды щелкните левой кнопкой мыши на имени функции `OnDraw`. Откроется окно редактирования файла MyView.cpp, а текстовый курсор будет расположен в тексте данной функции.
14. Измените функцию `OnDraw` в соответствии с текстом листинга 2.8.

Листинг 2.8 Функция `OnDraw`

```
// Вывод информации в CMyView

void CMyView::OnDraw(CDC* pDC)
{
  CMyDoc* pDoc = GetDocument();
  ASSERT_VALID(pDoc);
}
```

15. После реализации функции `Dump` поместите текст, приведенный в листинге 2.9.

Листинг 2.9 Функция `GetDocument`

```
CMyDoc* CMyView::GetDocument() const  // Распространяемая версия
                                      // является встраиваемой
{
  ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CMyDoc)));
  return (CMyDoc*)m_pDocument;
}
```

16. Выберите команду меню **View | Resource View** (Вид | Просмотр ресурсов) или нажмите комбинацию клавиш <Ctrl>+<Shift>+<E>. Откроется окно **Resource View** (Просмотр ресурсов).
17. Раскройте папку **MDI**, раскройте в ней папку, щелкните правой кнопкой мыши на папке **MDI.rc** и выберите в появившемся контекстном меню команду **Add Resource** (Добавить ресурс). Появится диалоговое окно **Add Resource** (Добавить ресурс), изображенное на рис. 2.7.

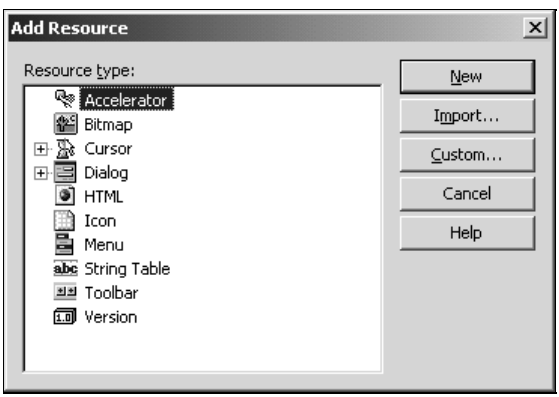


Рис. 2.7. Диалоговое окно **Add Resource**

18. В окне списка **Resource type** (Тип ресурса) выделите строку **Icon** (Значок) и нажмите кнопку **Import** (Импортировать). Появится диалоговое окно **Import**.
19. В окне списка раскройте папку /res и выделите в раскрывающемся списке **Files of type** (Тип файлов) строку **Icon Files** (Файлы значков). Диалоговое окно **Import** (Импортировать) примет вид, изображенный на рис. 2.8
20. Дважды щелкните левой кнопкой мыши на значке **MDIDoc**. Раскроется окно редактирования ресурса значка.

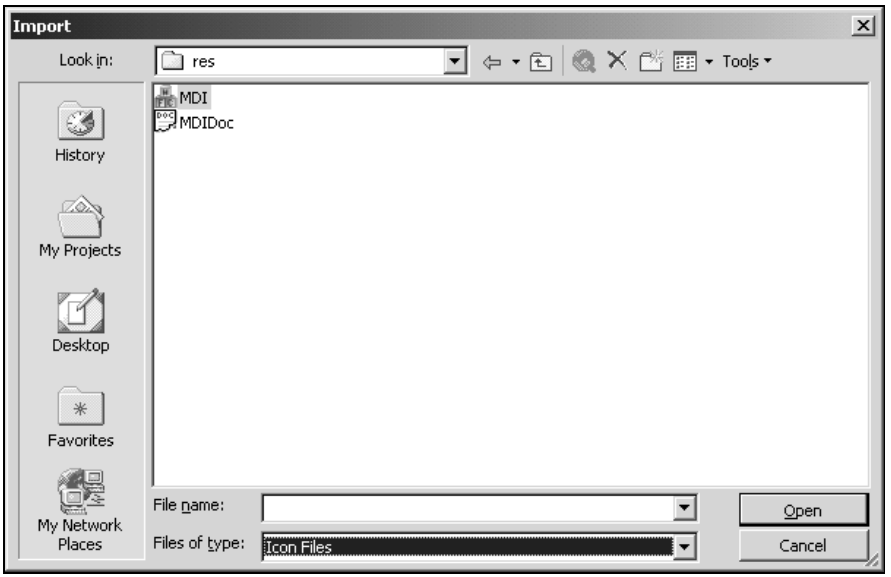


Рис. 2.8. Диалоговое окно **Import**

21. В окне **Resource View** (Просмотр ресурсов) щелкните правой кнопкой мыши на идентификаторе ресурса `IDI_ICON1` и выберите в появившемся контекстном меню команду **Properties** (Свойства). Откроется окно **Properties** (Свойства), изображенное на рис. 2.9.

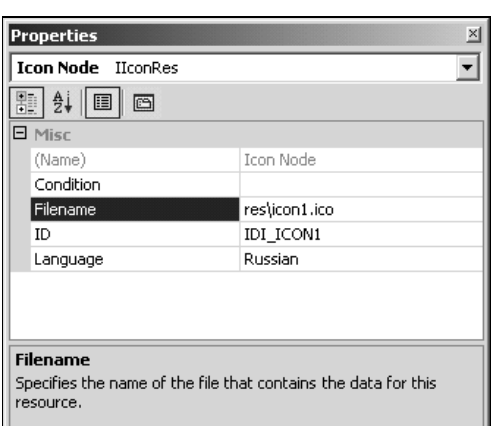


Рис. 2.9. Окно **Properties**

22. Введите в текстовое поле раскрывающегося списка **ID** (Идентификатор ресурса) идентификатор ресурса `IDR_MYTYPE`.

23. В окне **Resource View** (Просмотр ресурсов) раскройте папку **String Table** (Таблица строковых ресурсов) и дважды щелкните левой кнопкой мыши на значке **String Table** (Таблица строковых ресурсов). Откроется окно редактирования строковых ресурсов.
24. Выделите строку `IDR_MDITYPE` и введите в столбец **Caption** (Заголовок) текст "\nMDI\nMDI\nMDI Documents *.mdi\n.mdi\nMDI.Document\nMDI.Document". Таблица строковых ресурсов примет вид, изображенный на рис. 2.10.

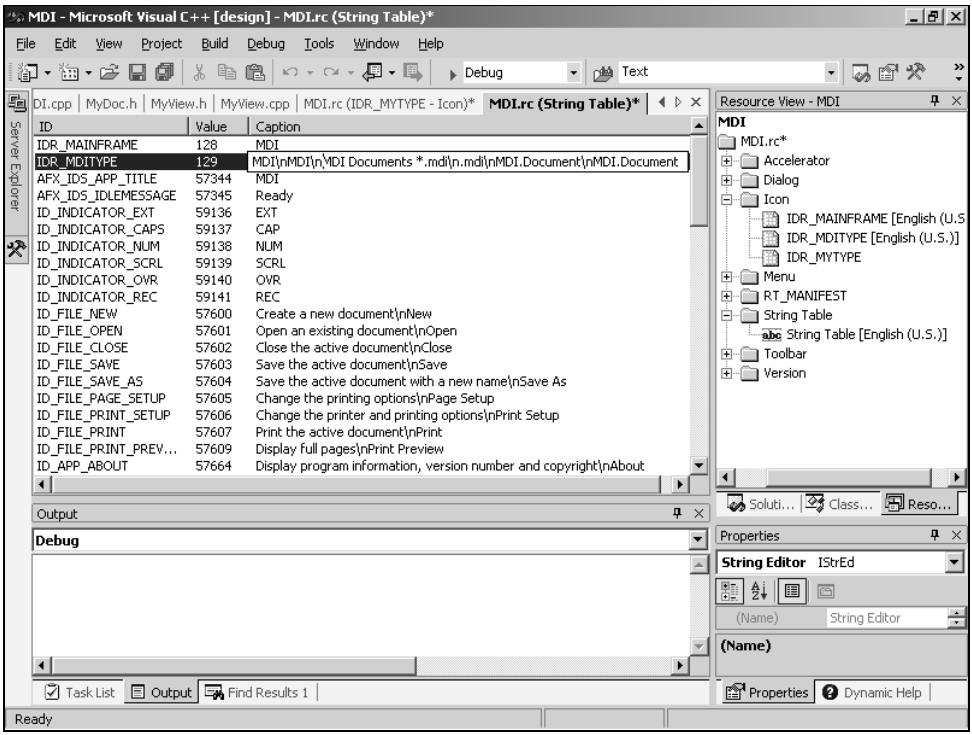


Рис. 2.10. Таблица строковых ресурсов

25. Перейдите в конец таблицы строковых ресурсов и щелкните левой кнопкой мыши в столбце **ID** (Идентификатор ресурса) пустой строки, расположенной в конце этого списка. На месте щелчка появится раскрывающийся список. Выберите в нем идентификатор ресурса `IDR_MYTYPE`.
26. В столбец **Caption** (Заголовок) данной строки введите текст "\nMy\nMy\nMy Documents *.my\n.my\nMy.Document\nMy.Document".
27. В окне **Resource View** (Просмотр ресурсов) раскройте папку **Menu** (Меню), щелкните правой кнопкой мыши на идентификаторе ресурса `IDR_MDITYPE` и выберите в появившемся контекстном меню команду **Copy** (Копировать).

28. Щелкните правой кнопкой мыши на папке **Menu** (Меню) и выберите в появившемся контекстном меню команду **Paste** (Вставить). В данную папку будет добавлен идентификатор ресурса `IDR_MDITYPE1`.
29. Выделите идентификатор ресурса `IDR_MDITYPE1` и выделите в раскрывающемся списке **ID** (Идентификатор ресурса) окна **Properties** (Свойства) идентификатор ресурса `IDR_MYTYPE`.
30. Нажмите клавишу <F5> и запустите приложение на исполнение. Появится диалоговое окно **New** (Создать), изображенное на рис. 2.11.

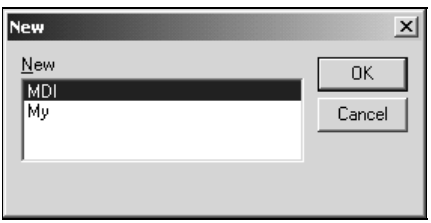


Рис. 2.11. Диалоговое окно **New**

31. Нажмите кнопку **OK**. На экран будет выведено приложение с раскрытым окном выделенного в окне **New** (Создать) типа.
32. Выберите команду меню **File | Open** (Файл | Открыть) и нажмите кнопку **Open** (Открыть) в панели инструментов. Появится диалоговое окно **Open** (Открыть).
33. Раскройте в нем список **Files of type** (Тип файлов) — окно примет вид, изображенный на рис. 2.12.
34. Закройте диалоговое окно **Open** (Открыть) и приложение.

Для добавления в приложение нового типа документов необходимо создать не только новый класс документа, но и новый класс представления, который будет использоваться для отображения документов данного типа. Конечно, никто не запрещает использовать один и тот же класс представления для работы с различными типами документов, однако следует помнить, что базовая функция класса представления `GetDocument` возвращает указатель на объект конкретного класса документа, что существенно затрудняет использование данного класса представления с другими типами документов.

В данном демонстрационном приложении это обстоятельство несущественно, но оно становится важным в реальном приложении. Поскольку среда программирования Visual C++ связывает класс представления с классом документа только один раз при создании приложения, то при создании других классов представления пользователю нужно внести в них некоторые изменения, связанные, прежде всего, с тем, что любой класс представления, создаваемый мастером MFC Class Wizard, связывается с классом `CDocument` и в заготовке класса представления отсутствует функция `GetDocument`.

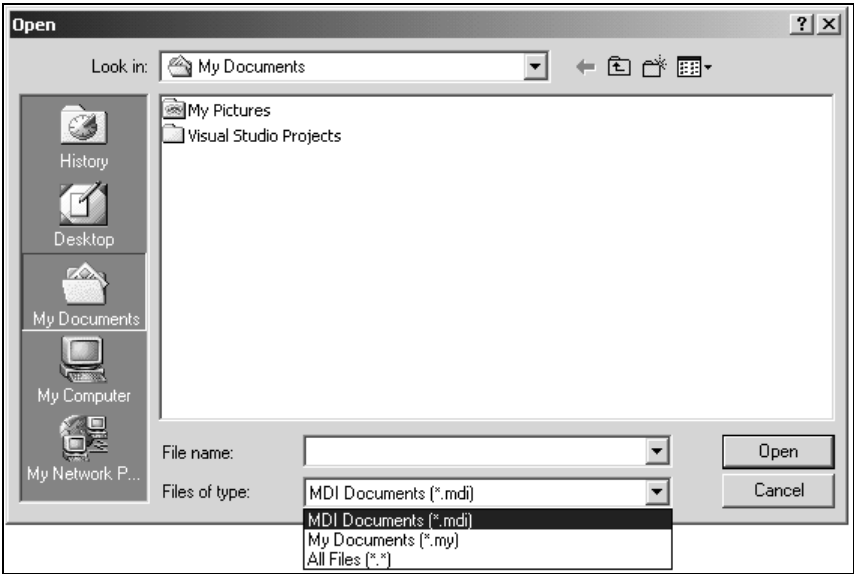


Рис. 2.12. Диалоговое окно **Open**

Изменения, внесенные нами в файл заголовка и в файл реализации пользовательского класса представления, свелись к тому, что в них было включено объявление функции `CDocument`, а также ее реализации для отладочной и окончательной версий приложения. Эти реализации, после внесения в них соответствующих изменений, могут быть переписаны из класса представления, созданного мастером MFC Application Wizard.

Функция `GetDocument` используется в заготовке функции `OnDraw`. При изменении текста этой заготовки в ней не только был изменен тип создаваемой переменной, но и включена проверка корректности этого типа.

Изменения, внесенные в функцию `CMDIApp::InitInstance`, сводятся к добавлению в приложение нового шаблона документа. Этот процесс был подробно описан в предыдущем разделе.

Изменения, внесенные в ресурсы приложения, связаны с тем, что каждому типу шаблона документа должен соответствовать свой значок, свое меню и свой строковый ресурс. Для простоты оба используемых шаблона документа имеют одинаковые значки и меню, хотя каждое из них может быть изменено независимо от соответствующего ресурса другого шаблона документа. Однако строковый ресурс каждого шаблона документа должен отличаться, поскольку в нем содержится идентификатор, определяющий этот ресурс в окне **New** (Новый), текстовая строка, позволяющая выбирать этот тип документа в окнах **Open** (Открыть) и **Save As** (Сохранить как), а также расширение, характеризующее файлы документов данного типа. Более подробно строка текстового ресурса шаблона документа будет рассмотрена в *главе 13* при описании русификации приложения.

Глава 3

Диалоговые окна и простейшие элементы управления

После рассмотрения заготовок приложений, создаваемых мастером создания приложений MFC, и концепции Документ/Представление, нам необходимо сделать отступление и рассмотреть диалоговые окна.

Дело в том, что без диалоговых окон практически невозможно написать какую-либо программу в среде Windows. Каждая программа создается для обработки некоторой информации, поэтому ей необходимо в какой-либо форме передать сведения о том, где взять эту информацию, что с ней делать и куда ее поместить. В случае отладочных программ эту информацию можно, конечно, помещать непосредственно в текст программы и изменять ее путем редактирования файла. Но данный подход нельзя признать изящным, и он абсолютно неприменим к программам, с которыми должен работать кто-нибудь еще, кроме их создателя. Другим вариантом передачи информации в программу является меню, но работа с ним тоже нуждается в описании и, кроме того, при работе с меню, так же как и при работе с диалоговыми окнами, невозможно обойти вопрос о сообщениях и их обработке.

Диалоговое окно

Полноценная программа в среде Windows может иметь множество диалоговых окон, каждое из которых будет предназначено для обмена или вывода информации в своем специфическом случае. Описание каждого диалогового окна состоит из двух частей, расположенных в различных файлах: описания ресурсов диалогового окна, для хранения которых используются специальные файлы ресурсов, имеющие, как правило, расширение rc, и описания класса диалогового окна, расположенного в файле реализации. Каждому из этих файлов соответствует свой файл заголовка.

Формирование ресурсов диалогового окна

Каждое приложение в среде Visual C++ содержит единственный файл ресурсов, имеющий расширение rc, имя которого совпадает с именем приложения. В этом файле содержится описание всех ресурсов данного приложения, к которым, помимо диалоговых окон, относятся также таблица назначения клавиш, значки,

меню, таблица строковых ресурсов, панели инструментов и сведения о версии приложения. Данный файл использует файл заголовка Resource.h, содержащий таблицу соответствия идентификаторов ресурсов числовым величинам.

Использование программой единственного файла ресурсов, имеющего имя приложения, не означает, что пользователь не может создавать собственного файла ресурсов. Просто этот файл необходимо включить в главный файл ресурсов приложения с использованием директивы `#include`, как это сделано для некоторых файлов стандартных ресурсов таких, как afxres.rc и afxprint.rc. Однако ресурсы, описанные в данных файлах, не могут редактироваться в стандартных окнах редактирования ресурсов, а сами эти ресурсы не отображаются в окне **Resource View** (Просмотр ресурсов). Это обстоятельство заставляет пользователя три раза подумать прежде, чем он создаст свой файл ресурсов. Использование этих файлов оправдано только в том случае, когда пользователь имеет большую библиотеку ресурсов, которые он хочет применять в данном приложении. В Visual C++ 6.0 этот вопрос мог быть решен с использованием галереи компонентов. В Visual Studio.NET использование галереи компонентов предусматривается только для Visual FoxPro, и не предусмотрено для приложений C++.

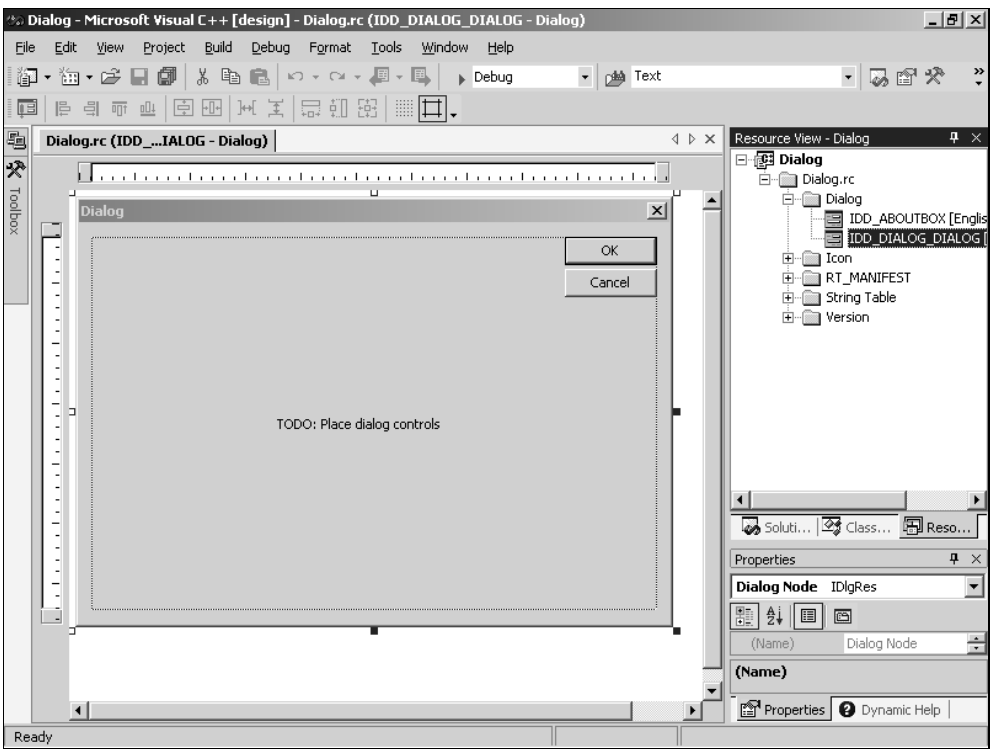


Рис. 3.1. Окно редактирования ресурса диалогового окна

В качестве заготовки для исследования диалоговых окон выберем заготовку приложения Dialog, созданную в *главе 1.* Если вы еще не создали эту заготовку, обратитесь к *главе 1* и выполните описанные там действия по ее созданию. Внешний вид интегрированной оболочки пользователя после завершения работы мастера MFC Application Wizard по созданию диалогового приложения приведен на рис. 3.1.

Полный текст приложения Dialog можно найти на дискете, прилагаемой к данной книге. Чтобы самостоятельно создать из имеющейся заготовки тестовое приложение, содержащее основные элементы управления диалогового окна, выполните следующие действия:

1. Если до этого вы сохраняли приложение, созданное мастером, или производили над ним какие-нибудь действия и у вас закрыто окно **Resource View** (Просмотр ресурсов), выберите команду меню **View | Resource View** (Вид | Просмотр ресурсов) или нажмите комбинацию клавиш <Ctrl>+<Shift>+<E>. Справа от окна редактирования ресурса диалогового окна раскроется окно **Resource View**. Вообще-то это окно является вкладкой другого окна, имя которого Microsoft тщательно скрывает, и поэтому окно **Resource View** (Просмотр ресурсов) может быть раскрыто щелчком мыши по своему ярлыку.
2. Щелкните левой кнопкой мыши по статическому тексту **TODO: Place dialog controls here**, расположенному в центре заготовки диалогового окна, и нажмите клавишу <Delete>. Статический текст исчезнет.
3. Если папка Dialog.rc закрыта, откройте эту папку, а в ней откройте папку Dialog. После чего щелкните левой кнопкой мыши на идентификаторе ресурса `IDD_DIALOG_DIALOG`. Под окном **Resource View** (Просмотр ресурсов) в окне **Properties** (Свойства) появятся обобщенные свойства данного диалогового окна.

   Если окно **Properties** (Свойства) закрыто, существует множество способов получить к нему доступ:

   - щелкните правой кнопкой мыши на идентификаторе ресурса `IDD_DIALOG_DIALOG` и в появившемся контекстном меню выберите команду **Properties** (Свойства). Для появления в меню этой команды должно быть открыто окно редактирования данного ресурса;
   - выберите команду меню **View | Properties Window** (Вид | Окно свойств);
   - нажмите клавишу <F4>;
   - это окно, также является вкладкой другого окна, имя которого, конечно же, засекречено. Поэтому оно может быть открыто и как обычная вкладка.
4. В окне **Properties** (Свойства) щелкните левой кнопкой мыши на строке **Language** (Язык). В соответствующем текстовом поле появится кнопка раскрывающегося списка.
5. Раскройте этот список и выделите в нем строку "Russian" (Русский).
6. Щелкните левой кнопкой мыши на заготовке диалогового окна. Содержимое окна **Properties** (Свойства) изменится.

7. В текстовое поле строки **Caption** (Заголовок) раздела **Appearance** (Внешний вид) иерархического списка свойств введите текст "Тестовое диалоговое приложение".
8. Закройте окно **Output** (Окно вывода). Поместите указатель мыши на синий квадрат, расположенный в правом нижнем углу заготовки диалогового окна, нажмите левую кнопку мыши и, не отпуская ее, переместите указатель мыши вправо вниз. Это приведет к увеличению геометрических размеров заготовки диалогового окна. Установите таким образом размер заготовки диалогового окна 200×320 и отпустите левую кнопку мыши.
9. Поместите указатель мыши на ярлык окна **Toolbox** (Инструментарий). Раскроется соответствующее окно, как это показано на рис. 3.2.

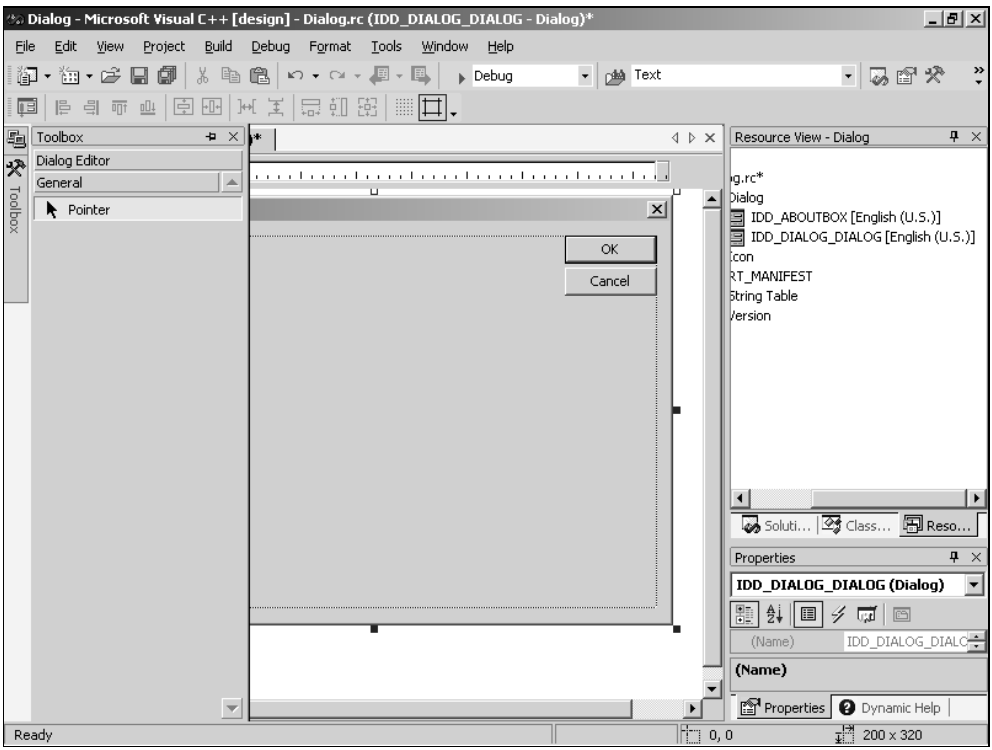


Рис. 3.2. Окно **Toolbox**

10. Раскройте список **Dialog Editor** (Редактор диалогового окна), как это показано на рис. 3.3.
11. Поместите указатель мыши на строку **Static Text** (Статический текст), нажмите левую кнопку мыши и, удерживая ее, перетащите рамку статического

текста в левый верхний угол заготовки диалогового окна. Заготовка диалогового окна примет вид, показанный на рис. 3.4.

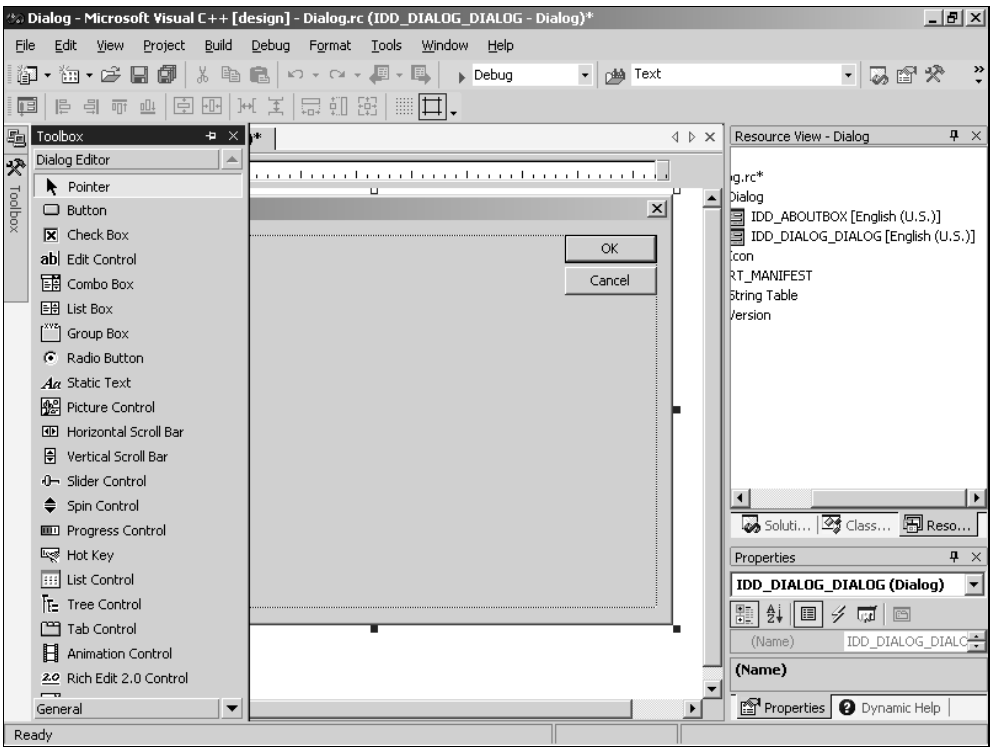


Рис. 3.3. Раскрывающийся список **Dialog Editor**

12. В текстовое поле **Caption** (Заголовок) окна **Properties** (Свойства) введите текст "Текстовое поле".
13. Поместите курсор мыши на строку **Edit Control** (Текстовое поле) в окне **Toolbox** (Инструментарий), нажмите левую кнопку мыши и перетащите рамку текстового поля ниже соответствующего ей статического текста. Отрегулируйте ее размеры и положение.

    Чтобы изменить положение элемента управления, нужно его выделить, щелкнув по нему левой кнопкой мыши. Вокруг выделенного элемента управления появятся восемь синих квадратиков (по одному в каждом углу рамки и по одному в середине каждой стороны рамки).

    При нахождении указателя мыши в пределах рамки, образуемой данными квадратиками, он приобретает вид крестика, состоящего из стрелок, направленных наружу. При нажатой левой кнопке мыши подобный указатель

позволяет перетаскивать выделенный элемент управления без изменения его геометрических размеров.

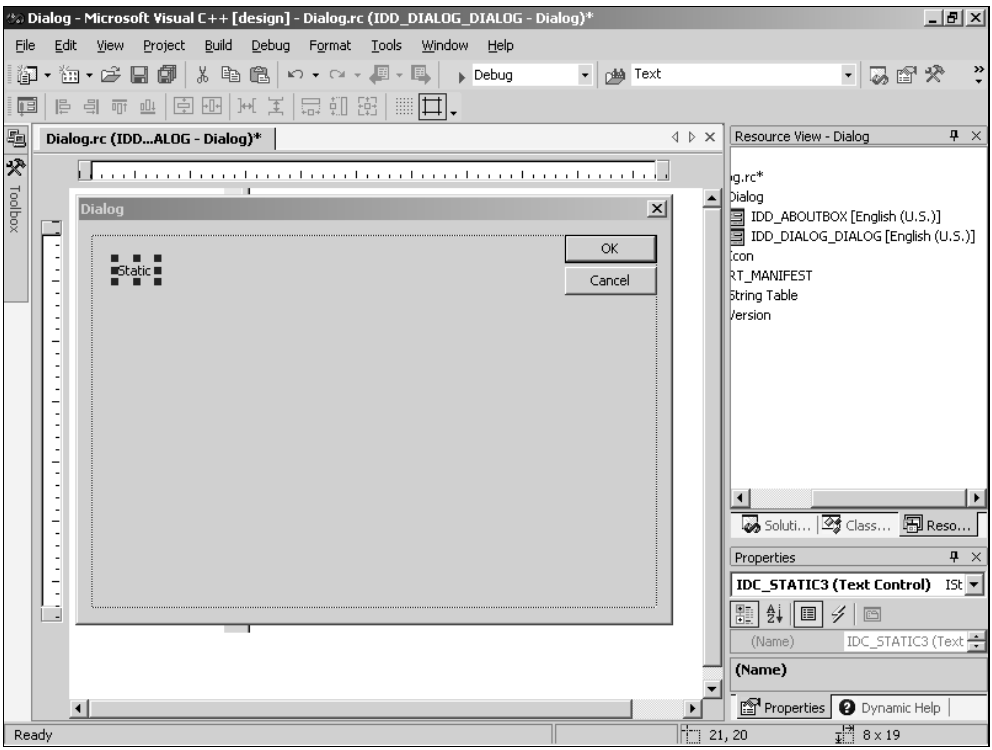


Рис. 3.4. Заготовка диалогового окна с рамкой статического текста

Если указатель мыши помещается на один из восьми синих квадратов, он приобретает форму двунаправленной стрелки, указывающей на возможное направление перемещения границы рамки. При нажатии левой кнопки мыши подобный указатель позволяет перемещать границу рамки в указанном стрелками направлении. При установке указателя мыши на угловой квадрат перемещаются обе стороны рамки, образующие этот угол. При помещении указателя мыши на квадрат, расположенный в центре одной из границ рамки, перемещается только эта граница рамки.

После регулировки размеров и положения текстового поля заготовка диалогового окна примет вид, изображенный на рис. 3.5.

14. Щелкните левой кнопкой мыши на строке **ID** (Идентификатор ресурса) раздела **Misc** (Общие свойства) окна **Properties** (Свойства) и вместо идентификатора `IDC_EDIT1` введите в соответствующее текстовое поле раскрывающегося списка идентификатор `IDC_EDIT_BOX`.

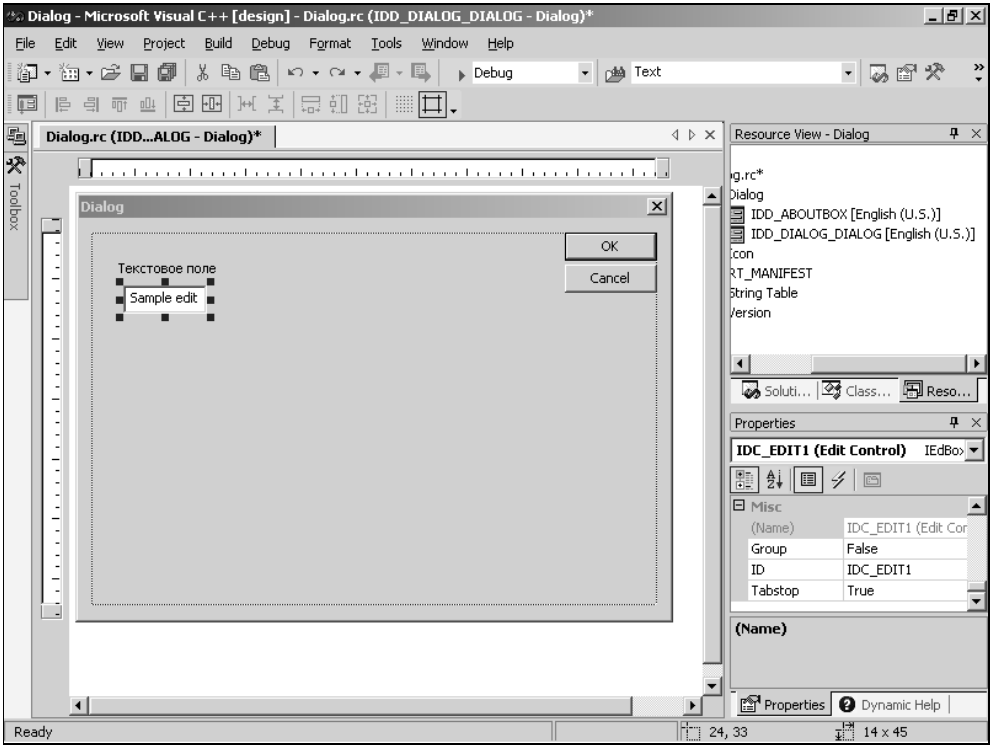


Рис. 3.5. Заготовка диалогового окна с текстовым полем

15. Повторите п.п. 11—14, создайте второе текстовое поле с идентификатором `IDC_BUDDY` и статическим текстом "Инкрементный регулятор", причем последним из этих двух элементов управления должно формироваться текстовое поле.

16. Выберите в окне **Toolbox** (Инструментарий) элемент управления **Spin Control** (Инкрементный регулятор) и поместите его справа от сформированного в предыдущем пункте текстового поля. Заготовка диалогового окна примет вид, изображенный на рис. 3.6.

17. В разделе **Appearance** (Внешний вид) иерархического списка окна **Properties** (Свойства) в раскрывающемся списке **Alignment** (Выравнивание) выберите **Right Align** (Выравнивание по правому краю). В результате этого инкрементный индикатор становится составной частью связанного с ним текстового поля и располагается в его правой части. При установке флажка **Wrap** (Цикл), расположенного в том же разделе, в случае достижения величиной в текстовом поле максимального значения при следующем ее увеличении она сбрасывается в ноль, а при достижении ею нуля при следующем уменьшении в него записывается максимально возможное значение. Установка флажка **ArrowKeys**

(Управление с клавиатуры) позволяет изменять значение в текстовом поле с помощью соответствующих клавиш, если фокус ввода установлен на данном элементе управления. Установка флажка **NoThousands** (Отсутствие разбиения по триадам) удаляет разделитель между триадами цифр.

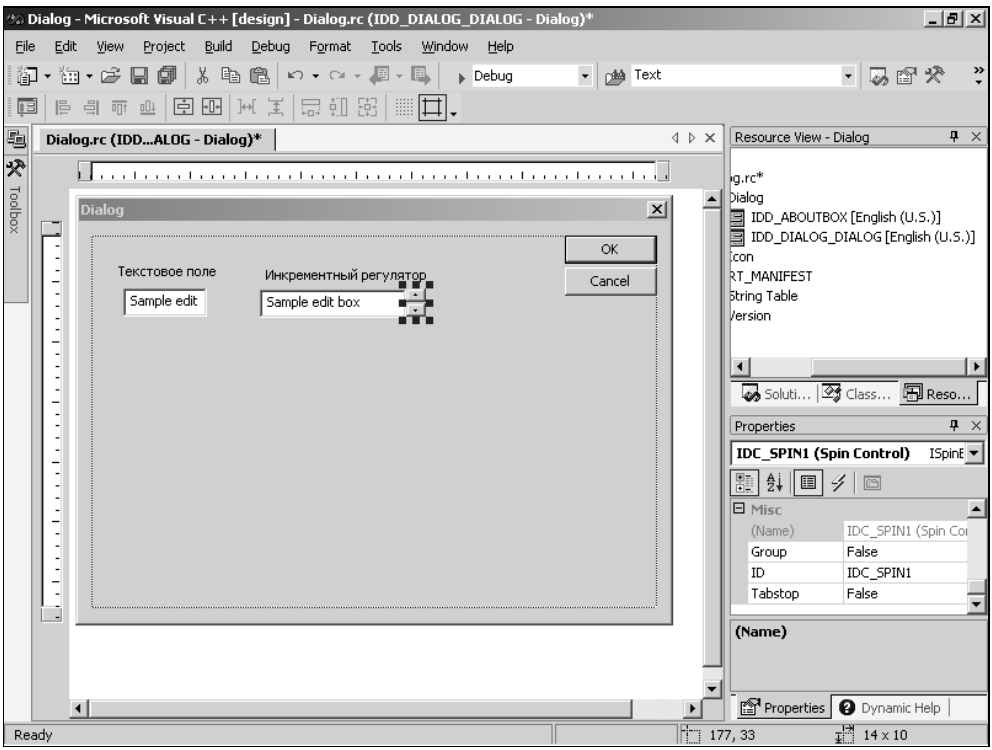


Рис. 3.6. Заготовка диалогового окна с инкрементным регулятором

18. В разделе **Behavior** (Поведение) того же иерархического списка в раскрывающихся списках **AutoBuddy** (Автоматическое связывание с полем) и **SetBuddyInteger** (Рассматривать содержимое связанного поля как целое число без знака), при установке флажка **AutoBuddy** инкрементный регулятор связывается с последним созданным перед ним элементом управления.
19. В текстовом поле раскрывающегося списка **ID** (Идентификатор ресурса), расположенного в разделе **Misc** (Общие свойства), замените установленный по умолчанию идентификатор на `IDC_SPIN`.
20. Расположите под кнопками **OK** и **Cancel** (Отмена) статический текст "Раскрывающийся список".
21. Выберите в окне **Toolbox** (Инструментарий) элемент управления **Combo Box** (Раскрывающийся список) и поместите его в заготовку диалогового окна

под только что введенный статический текст. Щелкните левой кнопкой мыши по кнопке раскрытия списка в раскрывающемся списке, поместите указатель мыши на синий квадратик на нижней границе рамки элемента управления и переместите его вниз, как это показано на рис. 3.7.

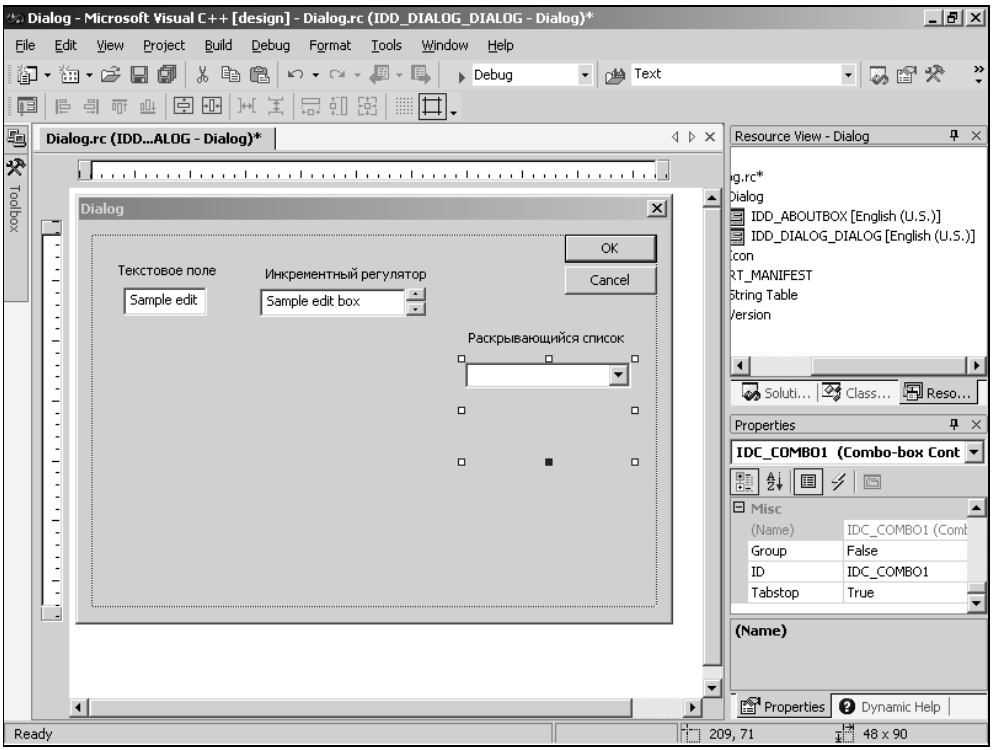


Рис. 3.7. Заготовка диалогового окна с раскрывающимся списком

22. В разделе **Behavior** (Поведение) иерархического списка окна **Properties** (Свойства) введите в текстовое поле раскрывающегося списка **Data** (Данные) текст "4;16;256". Этот текст будет выводиться в строках раскрывающегося списка, причем символ точки с запятой использован для разделения строк этого списка. В раскрывающемся списке **Sort** (Сортировка), расположенном в том же разделе, выделите строку **False** (Ложь).
23. В текстовом поле раскрывающегося списка **ID** (Идентификатор ресурса), расположенного в разделе **Misc** (Общие свойства), замените установленный по умолчанию идентификатор на `IDC_COMBO`.
24. Выберите в окне **Toolbox** (Инструментарий) элемент управления **Check Box** (Флажок) и поместите его в правый нижний угол заготовки диалогового окна.

25. В окне **Properties** (Свойства) измените идентификатор ресурса на `IDC_SAVE` и измените содержимое текстового поля **Caption** (Заголовок) на "Сохранять?". Заготовка диалогового окна примет вид, изображенный на рис. 3.8.

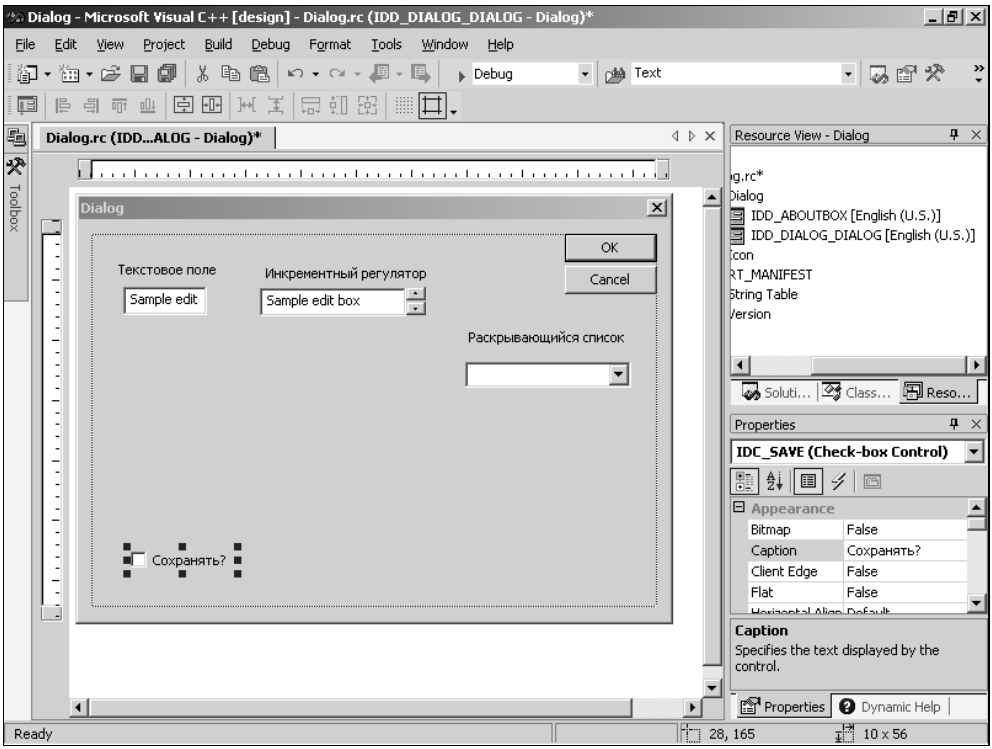


Рис. 3.8. Заготовка диалогового окна с флажком.

26. Выберите в окне **Toolbox** (Инструментарий) элемент управления **Group Box** (Рамка группы), поместите его между текстовым полем и флажком и растяните его на все свободное место.
27. В текстовое поле **Captions** (Заголовок) окна **Properties** (Свойства) введите заголовок "Выбор поля для сохранения".
28. Выберите в окне **Toolbox** (Инструментарий) элемент управления **Radio Button** (Переключатель) и поместите его в верхнюю часть только что созданной рамки группы.
29. В окне **Properties** (Свойства) измените идентификатор ресурса на `IDC_EDIT_SWITCH`, измените содержимое текстового поля **Caption** (Заголовок) на "Текстовое поле" и выберите `True` в текстовом поле **Group** (Группа), расположенном в разделе **Misc** (Общие свойства).

30. Повторите п.п. 28 и 29 для второго переключателя, расположив его под первым переключателем. Используйте для него идентификатор ресурса `IDC_SPIN_SWITCH` и заголовок "Инкрементный регулятор". В этом диалоговом окне не нужно устанавливать флажок **Group** (Группа). Заготовка диалогового окна примет вид, изображенный на рис. 3.9.

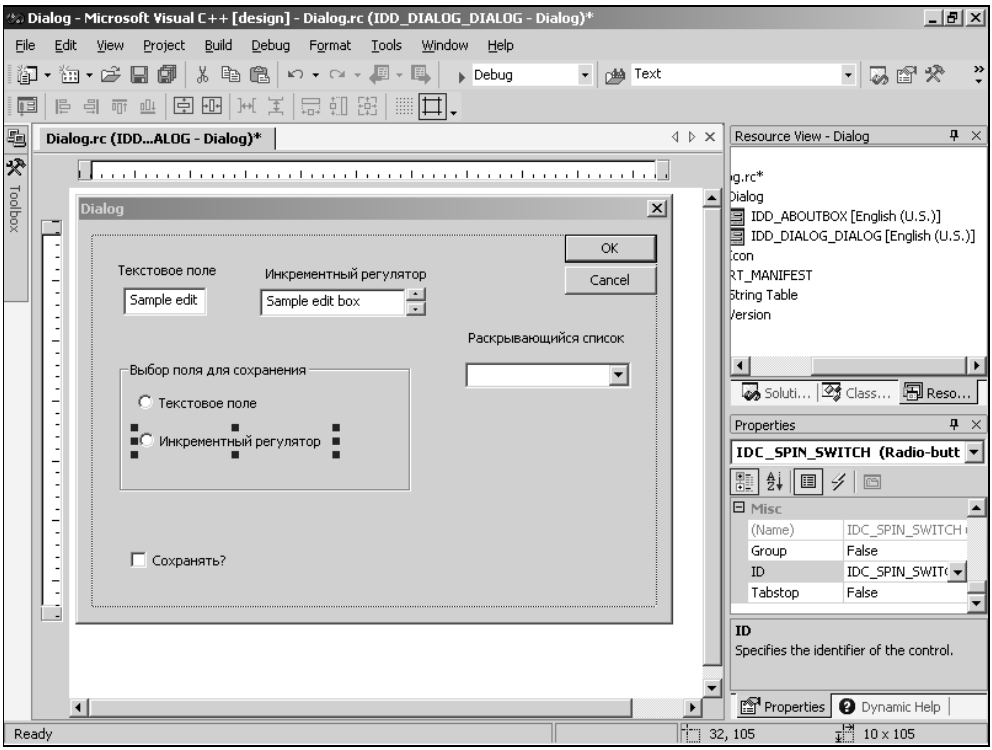


Рис. 3.9. Заготовка диалогового окна с группой переключателей

31. Щелкните левой кнопкой мыши на кнопке **Cancel** (Отмена) и введите в текстовое поле **Caption** (Заголовок) окна **Properties** (Свойства) заголовок "Отмена". В результате всех описанных выше действий заготовка диалогового окна примет вид, изображенный на рис. 3.10.

На этом этап формирования ресурсов диалогового окна можно считать законченным. Дело в том, что одновременно с производимыми нами действиями программная оболочка Visual C++ фиксировала все вносимые изменения в файлах Dialog.rc и Resource.h. В результате наших действий в файле Dialog.rc появилась, среди прочего, структура диалогового окна, приведенная в листинге 3.1.

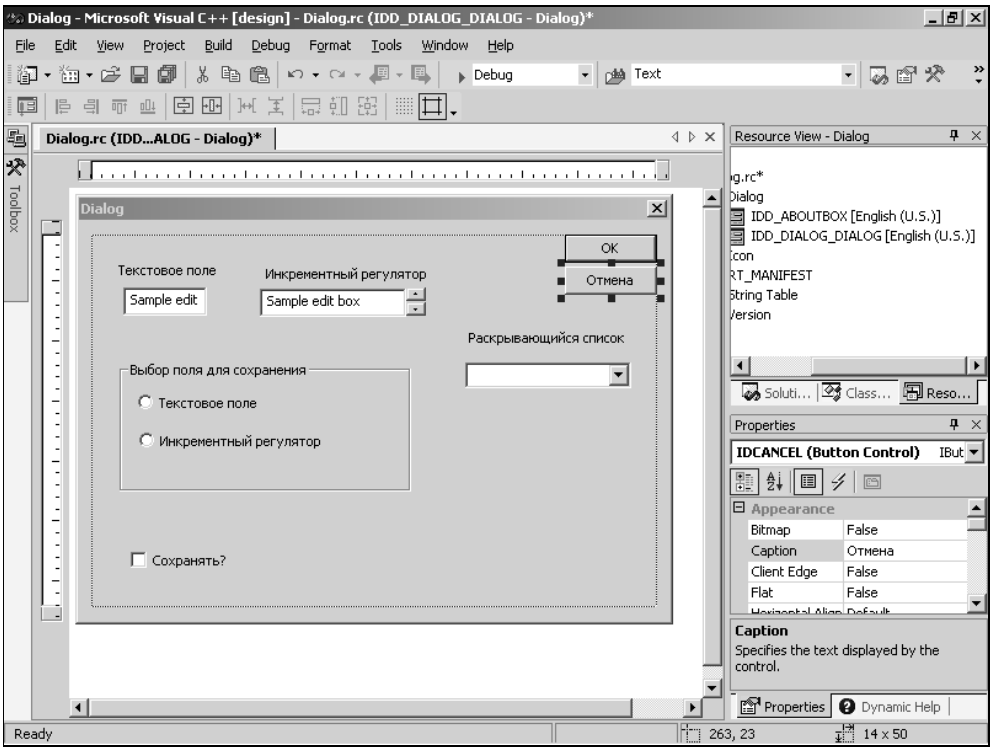


Рис. 3.10. Окончательный вид диалогового окна

Листинг 3.1. Описание ресурса диалогового окна в файле ресурса приложения

```
/////////////////////////////////////////////////////////////////////////////
//
// Dialog
//

IDD_DIALOG_DIALOG DIALOGEX 0, 0, 320, 200
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION |
WS_SYSMENU
EXSTYLE WS_EX_APPWINDOW
CAPTION "Тестовое диалоговое приложение"
FONT 8, "MS Sans Serif"
BEGIN
    DEFPUSHBUTTON   "OK",IDOK,263,7,50,14
    PUSHBUTTON      "Отмена",IDCANCEL,263,23,50,14
```

```
    LTEXT           "Текстовое поле",IDC_STATIC,19,20,55,8
    EDITTEXT        IDC_EDIT_BOX,27,32,40,14,ES_AUTOHSCROLL
    LTEXT           "Инкрементный регулятор",IDC_STATIC,93,20,88,8
    EDITTEXT        IDC_BUDDY,94,32,75,14,ES_AUTOHSCROLL
    CONTROL         "Spin1",IDC_SPIN,"msctls_updown32",UDS_SETBUDDYINT  |
                    UDS_ALIGNRIGHT  |  UDS_AUTOBUDDY  |
                    UDS_ARROWKEYS,170,32,10,14
    LTEXT           "Раскрывающийся список",IDC_STATIC,207,54,88,8
    COMBOBOX        IDC_COMBO,206,67,90,46,CBS_DROPDOWN  |  CBS_SORT  |
                    WS_VSCROLL  |  WS_TABSTOP
    CONTROL         "Сохранять?",IDC_SAVE,"Button",BS_AUTOCHECKBOX  |
                    WS_TABSTOP,40,152,55,10
    GROUPBOX        "Выбор поля для сохранения",IDC_STATIC,27,68,154,67
    CONTROL         "Текстовое поле",IDC_EDIT_SWITCH,"Button",
                    BS_AUTORADIOBUTTON  |  BS_NOTIFY  |
                    WS_GROUP,37,87,70,10
    CONTROL         "Инкрементный регулятор",IDC_SPIN_SWITCH,"Button",
                    BS_AUTORADIOBUTTON,37,107,103,10
END
```

В первой строке этой структуры указан идентификатор ресурса диалогового окна и его размеры в оконных координатах, поэтому верхний левый угол данного окна всегда имеет координаты `0`, `0`. После этого следует описание стиля диалогового окна, его заголовок и описание используемого в нем шрифта. Далее, между программными скобками `BEGIN` и `END`, располагается описание элементов управления данного окна. Описание элемента управления начинается с указания на то, что это за элемент управления, затем следует его заголовок, если он есть, идентификатор и другая специфическая информация. В конце описания элемента управления расположены координаты его рамки.

В случае необходимости пользователь может вносить изменения в описание элемента управления или любого другого ресурса непосредственно в файле описания ресурсов, однако намного удобнее и надежнее делать это с использованием интегрированной оболочки разработчика.

Если вас не устраивает дизайн диалогового окна (см. рис. 3.10), вы можете внести в него изменения. Для того чтобы перенести любой элемент управления на новое место, достаточно выделить его щелчком левой кнопки мыши, что приведет к визуализации рамки элемента управления и превращению курсора мыши в крестик со стрелками на концах. После этого вы можете его перемещать с клавиатуры клавишами управления курсором или мышью, для чего достаточно нажать левую кнопку мыши, когда курсор имеет форму крестика со стрелками на концах и, удерживая кнопку нажатой, переместить элемент управления в новое место. При этом перемещаться будет не сам элемент управления, а его тонкая штриховая рамка.

При необходимости изменить размер элемента управления достаточно выделить его, как описано выше, и поместить курсор на один из синих квадратов в рамке элемента управления. При этом курсор примет форму отрезка линии с двумя стрелками на концах. Направление стрелок указывает, в какую сторону будет перемещаться соответствующая сторона рамки.

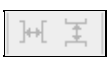
Для выравнивания размеров и положения элементов управления можно воспользоваться услугами панели инструментов **Dialog Editor** (Редактор диалогового окна), обычно расположенной над окном редактирования ресурса. Внешний вид панели инструментов **Dialog Editor** приведен на рис. 3.11 (для удобства эта панель инструментов сделана плавающей).

Рис. 3.11. Панель инструментов **Dialog Editor**

Кнопка (**Test Dialog** (Проверка диалогового окна), <Ctrl+T>) панели инструментов **Dialog Editor** (Редактор диалогового окна), расположенная на панели инструментов первой слева, позволяет, не запуская программы, проверить функционирование элементов управления диалога. Выход из этого режима производится по кнопкам закрытия диалогового окна в запущенном диалоге. В том случае, когда эти кнопки в диалоге отсутствуют, например, при создании вкладок диалогового окна, выход из этого режима осуществляется нажатием клавиши <Esc>.

Следующая группа кнопок управляет выравниванием положения элементов управления. Она включает в себя четыре кнопки: **Align Lefts** (Выровнять по левому краю), **Align Rights** (Выровнять по правому краю), **Align Tops** (Выровнять по верхней границе) и **Align Bottoms** (Выровнять по нижней границе). Чтобы выровнять элементы управления с использованием этих кнопок, необходимо выделить все элементы управления, к которым должна быть применена эта операция. Для этого достаточно последовательно щелкнуть по ним, удерживая нажатой клавишу <Shift> или <Ctrl>, после чего нажать на соответствующую кнопку. В результате рамки всех выделенных элементов управления будут выровнены по соответствующему краю.

За этой группой расположена группа кнопок , реализующая размещение элементов управления в пределах диалогового окна. Эта группа включает в себя две кнопки: **Vertical** и **Horizontal** (Центрирование по вертикали и горизонтали). Эти кнопки размещают элемент управления по центру диалогового окна, по вертикали или по горизонтали.

Следующие две кнопки  управляют взаимным положением элементов управления. Это кнопки **Across** и **Down** (Выровнять по горизонтали и выровнять

по вертикали). В результате нажатия этих кнопок крайние из выделенных элементов управления остаются на своем месте, а находящиеся между ними элементы управления перемещаются таким образом, чтобы выровнять расстояние между соседними элементами в выбранном направлении.

Предпоследняя группа кнопок используется для выравнивания геометрических размеров рамок выделенных элементов управления. В эту группу входят кнопки **Make Same Width**, **Make Same Height** и **Make Same Size** (Установить одинаковую ширину, установить одинаковую высоту и установить одинаковые размеры). Как понятно из названия этих кнопок, в результате нажатия на первую кнопку рамки всех элементов управления будут иметь одну и ту же ширину, в результате нажатия на вторую кнопку — одну и ту же высоту, а при нажатии на третью кнопку — как одинаковую ширину, так и одинаковую высоту.

Две последние кнопки представляют собой переключатель, при установке которого в положение **Toggle Grid** (Сетка установлена) на поверхности заготовки диалогового окна появляется сетка, и размеры каждого элемента управления привязываются к этой сетке. При установке этого переключателя в положение **Toggle Guides** (Убрать сетку) сетка исчезает, и каждый элемент управления получает возможность иметь произвольные размеры и занимать произвольное положение.

Внесение изменений в класс диалогового окна

После того как мы сформировали ресурсы диалогового окна, самое время приступить к созданию класса диалога. Конечно, это сильно сказано, поскольку львиную долю работы по созданию этого класса уже выполнил мастер создания диалогового приложения, а еще некоторую часть работы готова взять на себя среда программирования Visual C++.

Созданный нами ресурс диалогового окна уже связан с классом `CDialogDlg`. Об этом позаботился мастер создания диалогового приложения. От нас требуется только связать введенные нами элементы управления диалогового окна с данным классом. Для этого нам необходимо ввести в класс диалогового окна переменные, в которые будет записываться информация о состоянии элементов управления, и функции обработки сообщений о том, что с данными элементами управления была произведена некоторая операция (нажатие на кнопку, раскрытие списка и т. д.).

Чтобы добавить функции обработки сообщений, посылаемых элементами управления диалогового окна:

1. В заготовке диалогового окна выделите элемент управления **Раскрывающийся список** и нажмите кнопку **ControlEvents** (События элементов управления) в окне **Properties** (Свойства). В окне появится список сообщений, посылаемых данным элементом управления.

2. Выделите в нем сообщение `CBN_CLOSEUP` и раскройте связанный с ним список. В нем будет предложено имя функции обработки сообщения `OnCbnCloseupCombo`, как это показано на рис. 3.12. Выделите эту единственную строку в раскрывающемся списке и закройте его. Выделенная функция появится в текстовом поле раскрывающегося списка.

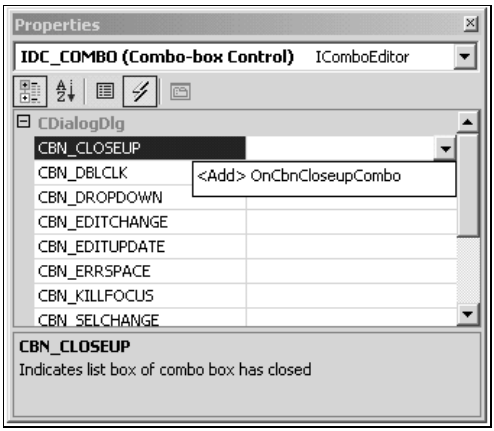


Рис. 3.12. Диалоговое окно **Properties**

3. Повторите п. 2 для сообщения `CBN_EDITCHANGE`. Ему будет сопоставлена функция обработки сообщения `OnCbnEditchangeCombo`.
4. В заготовке диалогового окна выделите инкрементный регулятор и нажмите кнопку **ControlEvents** (События элементов управления) в окне **Properties** (Свойства). В окне появится список сообщений, посылаемых данным элементом управления.
5. Повторите п. 2 для сообщения `UDN_DELTAPOS`. Ему будет сопоставлена функция обработки сообщения `OnDeltaposSpin`.

Чтобы ввести в класс переменные, хранящие информацию о состоянии элементов управления:

1. Выберите команду меню **View | Class View** (Вид | Просмотр классов) или нажмите комбинацию клавиш <Ctrl>+<Shift>+<C>. Справа от окна редактирования ресурса диалогового окна раскроется окно **Class View** (Просмотр классов). Вообще-то это окно является вкладкой того же окна, что и окно **Resource View** (Просмотр ресурсов), и оно может быть раскрыто щелчком мыши по своему ярлыку.
2. Раскройте папку **Dialog** (Окно диалога), если она закрыта, и щелкните правой кнопкой мыши на папке `CDialogDlg`.
3. В раскрывшемся контекстном меню выберите команду **Add | Add Variable** (Добавить | Добавить переменную). Появится диалоговое окно **Add Member Variable Wizard - Dialog** (Мастер добавления переменной в класс), изображенное на рис. 3.13.

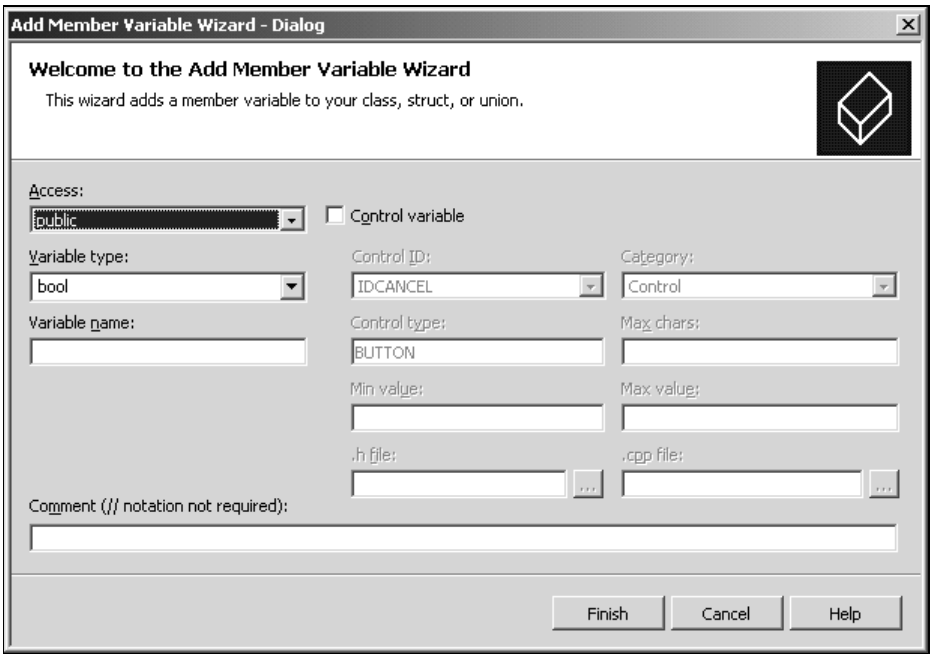


Рис. 3.13. Диалоговое окно **Add Member Variable Wizard - Dialog**

4. Установите флажок **Control variable** (Элемент управления, с которым связана переменная). При этом станут доступны раскрывающийся список **Control ID** (Идентификатор элемента управления) и текстовое поле **Control type** (Тип элемента управления).
5. Раскройте список **Control ID** (Идентификатор элемента управления) и выделите в нем идентификатор ресурса `IDC_EDIT_BOX`.
6. В раскрывающемся списке **Category** (Категория) выделите строку **Value** (Переменная), выберите в раскрывающемся списке **Variable type** (Тип переменной) тип переменной `UINT`, введите в текстовое поле **Variable name** (Идентификатор переменной) идентификатор `m_Edit` и нажмите кнопку **Finish** (Готово). Диалоговое окно **Add Member Variable Wizard** (Мастер добавления переменной в класс) закроется.
7. Повторите п.п. 5 и 6 для идентификатора ресурса `IDC_BUDDY`, сопоставив ему идентификатор переменной `m_Buddy`.
8. Повторите п.п. 5 и 6 для идентификатора ресурса `IDC_COMBO`, сопоставив ему идентификатор переменной `m_Combo`. Данный идентификатор может иметь только тип `CString`, поэтому в данном случае содержимое текстового поля раскрывающегося списка **Variable type** (Тип переменной) остается без изменений.

9. Повторите п.п. 5 и 6 для идентификатора ресурса `IDC_SAVE`, сопоставив ему идентификатор переменной `m_Save`. Данный идентификатор может иметь только тип `BOOL`, поэтому в данном случае содержимое текстового поля раскрывающегося списка **Variable type** (Тип переменной) остается без изменений.
10. Повторите п.п. 5 и 6 для идентификатора ресурса `ID_EDIT_SWITCH`, сопоставив ему идентификатор переменной `m_Switch`. В текстовом поле раскрывающегося списка **Variable type** (Тип переменной) стоит тип `BOOL`, а данный идентификатор может иметь только тип `int`. Этот тип не может быть изменен путем выбора из раскрывающегося списка. Поэтому введите его вручную.
11. Раскройте папку `CDialogDlg`. Окно **Class View - Dialog** (Просмотр классов) примет вид, изображенный на рис. 3.14.

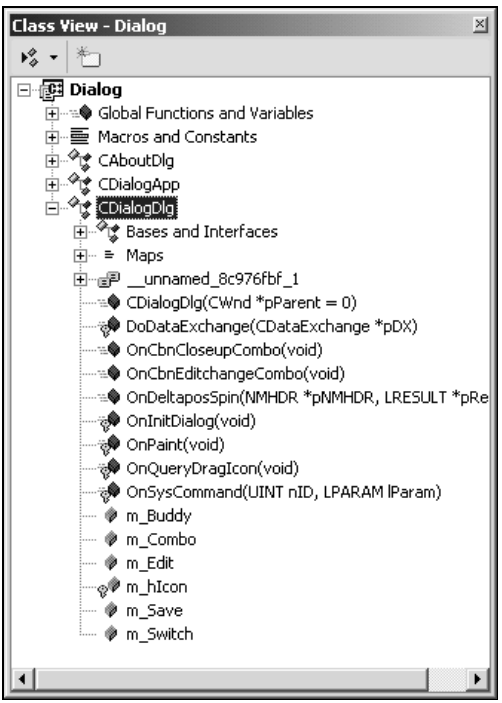


Рис. 3.14. Диалоговое окно **Class View - Dialog**

Как видно из рис. 3.14, в классе `CDialogDlg` присутствуют все добавленные нами члены.

После завершения выполнения описанной выше процедуры все операции, которые могли быть автоматизированы, уже выполнены. Осталось только внести некоторые коррективы вручную.

Текст файла заголовка класса диалогового окна приведен в листинге 3.2.

Листинг 3.2. Файл заголовка класса диалогового окна

```
// Файл заголовка DialogDlg.h
//
#pragma once

// Класс диалогового окна CDialogDlg
class CDialogDlg : public CDialog
{
// Конструкторы
public:
  CDialogDlg(CWnd* pParent = NULL);  // Стандартный конструктор

// Данные диалогового окна
  enum { IDD = IDD_DIALOG_DIALOG };

  protected:
  virtual void DoDataExchange(CDataExchange* pDX);  // Поддержка DDX/DDV

// Реализация
protected:
  HICON m_hIcon;

  // Функции карты сообщений
  virtual BOOL OnInitDialog();
  afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
  afx_msg void OnPaint();
  afx_msg HCURSOR OnQueryDragIcon();
  DECLARE_MESSAGE_MAP()
public:
  afx_msg void OnCbnCloseupCombo();
  afx_msg void OnCbnEditchangeCombo();
  afx_msg void OnDeltaposSpin(NMHDR *pNMHDR, LRESULT *pResult);
  UINT m_Edit;
  UINT m_Buddy;
  CString m_Combo;
  BOOL m_Save;
  int m_Switch;
};
```

Как следует из листинга 3.2, все функции обработки сообщений, как включенные в класс мастером создания диалогового приложения, так и созданные в окне **Properties** (Свойства), помещены в карту сообщений файла заголовка. Это не значит, что все пользовательские программы нужно помещать туда же. Карта сообщений поддерживается соответствующими мастерами и запись и удаление функций из нее лучше всего производить из этих мастеров. Все пользовательские функции, не связанные с обработкой сообщений, включаются в класс диалогового окна таким же образом, как и в любой другой класс. Однако наша программа не будет содержать таких функций.

Файл реализации DialogDlg.cpp содержит не только реализацию класса обработки сообщений созданного нами диалогового окна, но и описание класса обработки сообщений диалогового окна **About**, которое нас в настоящий момент не интересует. Поэтому в листинге 3.3 представлена только та часть этого файла, в которой содержится реализация интересующего нас класса. Для простоты восприятия комментарии, помещенные в него мастером MFC Class Wizard, переведены на русский язык.

Листинг 3.3. Реализация класса диалогового окна

```
// Класс диалогового окна CDialogDlg

CDialogDlg::CDialogDlg(CWnd* pParent /*=NULL*/)
  : CDialog(CDialogDlg::IDD, pParent)
  , m_Edit(0)
  , m_Buddy(0)
  , m_Combo(_T(""))
  , m_Save(FALSE)
  , m_Switch(0)
{
  m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}

void CDialogDlg::DoDataExchange(CDataExchange* pDX)
{
  CDialog::DoDataExchange(pDX);
  DDX_Text(pDX, IDC_EDIT_BOX, m_Edit);
  DDX_Text(pDX, IDC_BUDDY, m_Buddy);
  DDX_CBString(pDX, IDC_COMBO, m_Combo);
  DDX_Check(pDX, IDC_SAVE, m_Save);
  DDX_Radio(pDX, IDC_EDIT_SWITCH, m_Switch);
}
```

```
BEGIN_MESSAGE_MAP(CDialogDlg, CDialog)
  ON_WM_SYSCOMMAND()
  ON_WM_PAINT()
  ON_WM_QUERYDRAGICON()
  //}}AFX_MSG_MAP
  ON_CBN_CLOSEUP(IDC_COMBO, OnCbnCloseupCombo)
  ON_CBN_EDITCHANGE(IDC_COMBO, OnCbnEditchangeCombo)
  ON_NOTIFY(UDN_DELTAPOS, IDC_SPIN, OnDeltaposSpin)
END_MESSAGE_MAP()

// Функции обработки сообщений класса CDialogDlg

BOOL CDialogDlg::OnInitDialog()
{
  CDialog::OnInitDialog();

  // Добавление команды "About..." в системное меню.

  // Идентификатор IDM_ABOUTBOX должен быть определен в области
  // видимости 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);
    }
  }

  // Установка значка диалогового окна. Приложение производит эту
  // операцию автоматически, если главное окно приложения не
  // является диалоговым окном
  SetIcon(m_hIcon, TRUE);    // Установка больших значков
  SetIcon(m_hIcon, FALSE);   // Установка маленьких значков
```

```
  // Сюда можно поместить дополнительные процедуры инициализации

  return TRUE;  // Должно возвращать значение TRUE, если фокус ввода
                // не передан элементу управления
}

void CDialogDlg::OnSysCommand(UINT nID, LPARAM lParam)
{
  if ((nID & 0xFFF0) == IDM_ABOUTBOX)
  {
    CAboutDlg dlgAbout;
    dlgAbout.DoModal();
  }
  else
  {
    CDialog::OnSysCommand(nID, lParam);
  }
}

// Если диалоговое окно содержит кнопку Свернуть, расположенные ниже
// операторы выведут значок диалогового окна при минимизации.
// В приложениях, использующих концепцию Документ/Представление, эта
// операция производится автоматически.

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

    SendMessage(WM_ICONERASEBKGND, reinter-
pret_cast<WPARAM>(dc.GetSafeHdc()), 0);

    // Центрирование значка в рабочей области
    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;
```

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

// Данная функция вызывается системой для получения курсора,
// используемого при перетаскивании минимизированного окна.
HCURSOR CDialogDlg::OnQueryDragIcon()
{
  return static_cast<HCURSOR>(m_hIcon);
}

void CDialogDlg::OnCbnCloseupCombo(void)
{
  // Вставьте сюда текст вашей функции обработки сообщения
}

void CDialogDlg::OnCbnEditchangeCombo(void)
{
  // Вставьте сюда текст вашей функции обработки сообщения
}

void CDialogDlg::OnDeltaposSpin(NMHDR *pNMHDR, LRESULT *pResult)
{
  NM_UPDOWN* pNMUpDown = (NM_UPDOWN*)pNMHDR;
  // Вставьте сюда текст вашей функции обработки сообщения
  *pResult = 0;
}
```

Внесение изменений в этот текст начнем с конструктора класса. Как видно из листинга 3.3, при добавлении в класс диалогового окна новой переменной мастер одновременно включал ее инициализацию в конструктор данного класса. Теперь необходимо внести изменения в начальные значения этих переменных. Текст конструктора класса с измененными начальными значениями переменных приведен в листинге 3.4.

Листинг 3.4. Конструктор класса диалогового окна

```
CDialogDlg::CDialogDlg(CWnd* pParent /*=NULL*/)
  : CDialog(CDialogDlg::IDD, pParent)
  , m_Edit(64)
  , m_Buddy(32)
  , m_Combo(_T("1024"))
  , m_Save(TRUE)
  , m_Switch(0)
{
  m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}
```

Теперь необходимо внести изменения в функции обработки сообщений. Текст этих функций приведен в листинге 3.5.

Листинг 3.5. Функции обработки сообщений

```
// Обработка сообщения о закрытии раскрывающегося списка

void CDialogDlg::OnCbnCloseupCombo(void)
{
  UpdateData();     // Чтение состояния переменных

  if( m_Save )      // Проверка состояния флажка
    if( m_Switch )  // Сохранение в текстовом поле
      SetDlgItemText( IDC_BUDDY, m_Combo );
    else  // Сохранение в инкрементном регуляторе
      SetDlgItemText( IDC_EDIT_BOX, m_Combo );

  UpdateData();     // Сохранение изменений в членах класса
}

// Обработка сообщения о редактировании содержимого текстового поля
// раскрывающегося списка

void CDialogDlg::OnCbnEditchangeCombo(void)
{
  UpdateData();     // Чтение состояния переменных
```

```
  if( m_Save )       // Проверка состояния флажка
    if( m_Switch )   // Сохранение в текстовом поле
      SetDlgItemText( IDC_BUDDY, m_Combo );
    else  // Сохранение в инкрементном регуляторе
      SetDlgItemText( IDC_EDIT_BOX, m_Combo );

  UpdateData();      // Сохранение изменений в членах класса
}

// Изменение знака приращения инкрементного регулятора
void CDialogDlg::OnDeltaposSpin(NMHDR *pNMHDR, LRESULT *pResult)
{
  NM_UPDOWN* pNMUpDown = (NM_UPDOWN*)pNMHDR;
  pNMUpDown-> iDelta = -pNMUpDown-> iDelta;
  *pResult = 0;
}
```

Чтобы запустить приложение на исполнение, выберите команду меню **Debug | Start** (Отладка | Запуск) или нажмите клавишу <F5>. Появится диалоговое окно **Microsoft Development Environment**, изображенное на рис. 3.15. Оно сообщает о том, что текущая конфигурация проекта устарела, и предлагает ее перетранслировать.

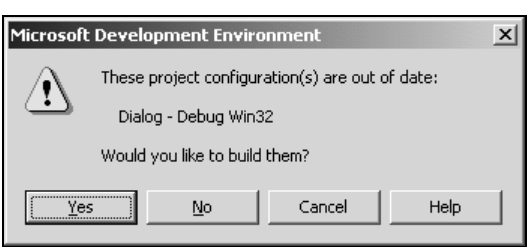


Рис. 3.15. Диалоговое окно **Microsoft Development Environment**

Нажмите кнопку **Yes** (Да). Произойдет трансляция программы и появится диалоговое окно, изображенное на рис. 3.16.

Как следует из рис. 3.16, значения переменных, установленные в конструкторе класса, отображаются в соответствующих им элементах управления. Значение `TRUE` переменной `m_Save` устанавливает флажок **Сохранять?**. Нулевое значение переменной `m_Switch` устанавливает переключатель в положение **Текстовое поле** (первое в группе переключателей и имеющее поэтому нулевой индекс). Строка, записанная в переменную `m_Combo`, выводится в текстовом поле раскрывающе-

гося списка. Значение переменой `m_Buddy` выводится в текстовом поле, связанном с инкрементным регулятором, а значение переменной `m_Edit` — в простом текстовом поле.

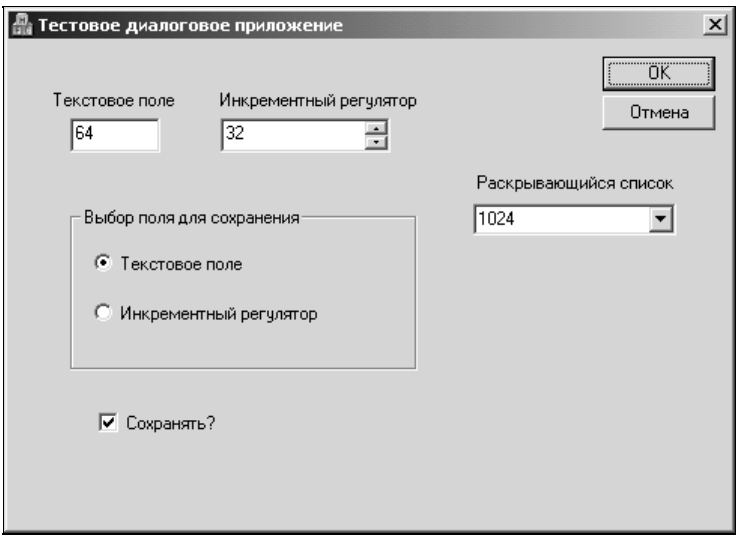


Рис. 3.16. Диалоговое окно программы

Поработайте с диалоговым окном приложения. Попробуйте изменить строку в текстовом поле раскрывающегося списка. Изменение произойдет не только в этом текстовом поле, но и в элементе управления **Текстовое поле**, причем содержимое этих текстовых полей будет совпадать. Установите переключатель **Выбор поля для сохранения** в положение **Инкрементный регулятор**. Дальнейшие изменения в текстовом поле раскрывающегося списка будут копироваться в текстовое поле, связанное с инкрементным регулятором. Раскройте раскрывающийся список и выделите в нем строку 256. После закрытия раскрывающегося списка значение в текстовом поле инкрементного регулятора не изменится. Повторите эту операцию, выбрав в раскрывающемся списке строку 16. После закрытия раскрывающегося списка его текстовое поле будет содержать значение 16, а текстовое поле инкрементного регулятора — значение 256, т. е. предыдущее значение этого текстового поля.

Такое несоответствие связано с тем, что диалоговое окно должно заполняться при его открытии, а информация о состоянии его элементов управления должна считываться только при его закрытии. Поэтому при создании функций обработки сообщений диалогового окна не обращалось никакого внимания на их синхронизацию. Попытки использования диалоговых окон для решения несвойственных им задач связаны с тем, что в Windows имеется только два типа окон: окно и диалоговое окно. Поэтому, все, что нельзя сделать в окне, приходится

делать в диалоговом окне. Вследствие этого на диалоговые окна часто приходится возлагать функции, которые не могут быть реализованы в схеме работы с единственной проверкой данных при закрытии окна. При этом во многих случаях необходимо, чтобы изменения, внесенные в один элемент управления, немедленно отражались на состоянии других элементов управления. В этом случае программисту приходится искать свои способы синхронизации данных в окне. Примером реализации такого подхода является данное приложение.

Тексты функций обработки сообщения `OnCbnCloseupCombo`, вызываемой при закрытии раскрывающегося окна, и `OnCbnEditchangeCombo`, вызываемой при вводе текста в его текстовое поле, абсолютно идентичны. Сначала в них вызывается функция `CWnd::UpdateData`, представляющая собой, по сути, вызов функции `CWnd::DoDataExchange`, сохраняющей информацию о состоянии элементов управления диалогового окна в соответствующих величинах. После этого, в зависимости от текущего состояния флага **Сохранять?** и переключателя **Выбор поля для сохранения** содержимое текстового поля раскрывающегося списка копируется в соответствующее текстовое поле. Для этого используется функция `CWnd::SetDlgItemText`. Для того чтобы внесенные изменения были сохранены в соответствующих переменных класса диалогового окна, после этого снова вызывается функция `UpdateData`.

Примечание

Несмотря на абсолютно одинаковый текст между результатами вызова функций `OnCbnCloseupCombo` и `OnCbnEditchangeCombo` существует различие. В этом легко убедиться, пробуя инкрементировать или декрементировать значение, записанное этими функциями в инкрементный регулятор. Если после вызова функции `OnCbnEditchangeCombo` инкрементный регулятор работает корректно, то после вызова функции `OnCbnCloseupCombo` инкрементируется некая внутренняя переменная инкрементного регулятора.

Сбросьте флажок **Сохранять?**. Изменения, внесенные в текстовое поле, перестанут отображаться в других текстовых полях.

Попробуйте увеличить или уменьшить значение в текстовом поле с помощью инкрементного регулятора. Вместо значения 256 в нем появится значение 100. Дело в том, что здесь используется диапазон значений, установленный по умолчанию, то есть от 0 до 100. Для изменения этого диапазона нужно создать объект класса `CSpinButtonCtrl` и вызвать для него метод `SetRange`.

Для данного элемента управления создается функция обработки сообщения `OnDeltaposSpin`. Она используется для установления естественного соответствия клавиш управления величиной в связанном с данным элементом управления текстовом поле направлению изменения этой величины. Как можно убедиться в режиме моделирования диалогового окна, разработчики библиотеки MFC полагали, что стрелке вниз должно соответствовать увеличение величины, а стрелке вверх — ее уменьшение, в то время как все остальные разработчики Microsoft, например разработчики Word, считали естественным обратное соответствие. Данная функция устанавливает более распространенное соответствие клавиш.

Вкладки и мастера

При работе с продуктами фирмы Microsoft вы постоянно встречались с диалоговыми окнами, содержащими несколько вкладок, и с мастерами, в которых содержимое диалогового окна меняется с использованием кнопок **Next** (Далее) и **Back** (Назад). Их использование позволяет существенно расширить возможности диалога пользователя, предоставляемого приложениями, разработанными в среде программирования Visual C++.

Создание вкладок диалогового окна

Текст тестового диалогового приложения, использующего вкладки в диалоговом окне, располагается в папке Sheet на демонстрационной дискете, поставляемой с данной книгой.

Чтобы самостоятельно создать тестовое диалоговое приложение, использующее вкладки в диалоговом окне:

1. Создайте новое диалоговое приложение с именем Sheet, следуя указаниям по созданию подобного приложения, изложенным в *главе 1*.
2. Выберите команду меню **View | Class View** (Вид | Просмотр классов) или нажмите комбинацию клавиш <Ctrl>+<Shift>+<C>. Справа от окна редактирования ресурса диалогового окна раскроется окно **Class View** (Просмотр классов). Это окно может быть раскрыто и щелчком мыши по своему ярлыку.
3. Щелкните правой кнопкой мыши на папке **Sheet** и выберите в раскрывшемся контекстном меню команду **Add | Add Class** (Добавить | Добавить класс). Появится диалоговое окно **Add Class - Sheet** (Добавить класс), изображенное на рис. 3.17.
4. В окне списка **Templates** (Шаблоны) выделите значок **MFC Class** (Класс MFC) и нажмите кнопку **Open** (Открыть). Появится диалоговое окно **MFC Class Wizard - Sheet** (Мастер создания классов MFC), как показано на рис. 3.18.
5. В раскрывающемся списке **Base class:** (Базовый класс) выделите имя базового класса `CPropertyPage`, введите в текстовое поле **Class name** (Имя класса) имя класса `CPage1` и нажмите кнопку **Finish** (Готово).
6. Повторите п.п. 3—5 для создания классов `CPage2` и `CPage3`.
7. Повторите п.п. 3—4, выделите в раскрывающемся списке **Base class** (Базовый класс) имя базового класса `CPropertySheet`, введите в текстовое поле **Class name** (Имя класса) имя класса `CSheet` и нажмите кнопку **Finish** (Готово). Появится окно, изображенное на рис. 3.19, содержащее сообщение о том, что файл Sheet.h уже существует, и предложение добавить заголовок создаваемого класса в этот файл.
8. Нажмите кнопку **Yes** (Да), согласившись с данным предложением. Появится аналогичное окно, предупреждающее о том, что файл Sheet.cpp уже существует и спрашивающее, нужно ли добавить реализацию создаваемого класса в этот файл.

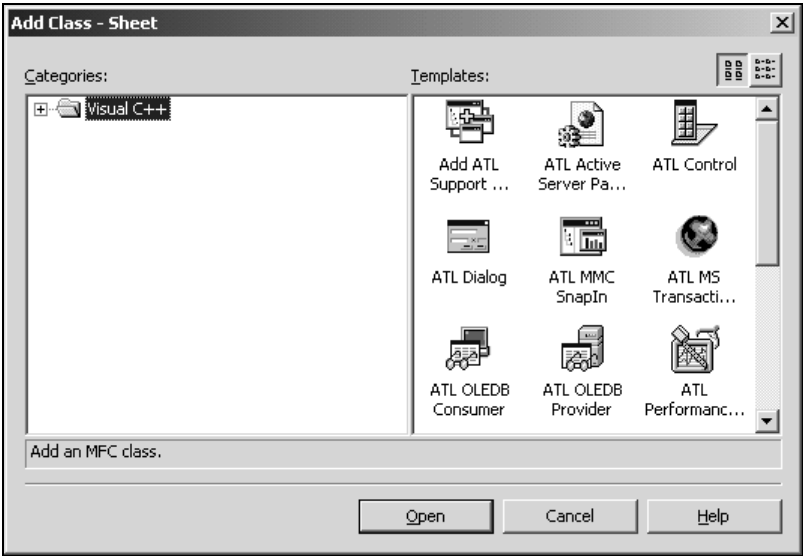


Рис. 3.17. Диалоговое окно **Add Class - Sheet**

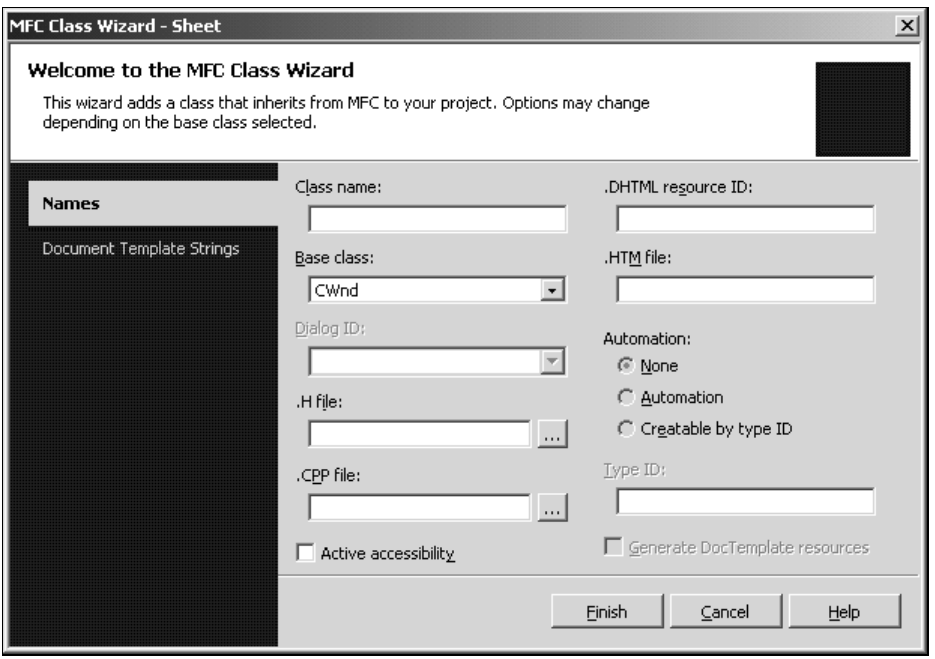


Рис. 3.18. Диалоговое окно **MFC Class Wizard - Sheet**

Рис. 3.19. Диалоговое окно **Microsoft Development Environment**

9. Нажмите кнопку **Yes** (Да), согласившись с данным предложением.
10. Раскройте окно **Resource View** (Просмотр ресурсов) и дважды щелкните левой кнопкой мыши на идентификаторе ресурса `IDD_PAGE1`. Раскроется окно редактирования соответствующего ресурса диалогового окна.
11. Уничтожьте все элементы управления, расположенные на заготовке данного диалогового окна, включая кнопки.
12. Раскройте окно **Properties** (Свойства) и в текстовое поле **Caption** (Заголовок) раздела **Appearance** (Внешний вид) иерархического списка свойств введите текст "Панель 1".
13. В раскрывающемся списке **Border** (Рамка) того же раздела выделите строку "Thin" (Тонкая), в раскрывающемся списке **Style** (Стиль) — строку "Child" (Дочернее окно), а в раскрывающемся списке **System menu** (Системное меню) — "False".
14. В окне **Toolbox** (Инструментарий) выберите элемент управления **Check Box** (Флажок) и поместите его в центр заготовки диалогового окна.
15. Задайте для него идентификатор ресурса `IDC_CHECK` и заголовок "Разрешить переход?". Заготовка диалогового окна примет вид, изображенный на рис. 3.20.

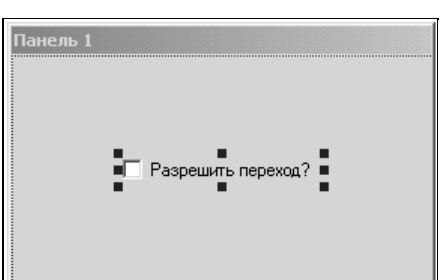


Рис. 3.20. Заготовка вкладки

16. Повторите п.п. 10—15 для идентификатора ресурса `IDD_PAGE2`, использовав при этом заголовок "Панель 2".
17. Повторите п.п. 10—15 для идентификатора ресурса `IDD_PAGE3`, использовав заголовок "Панель 3".

На этом процесс создания заготовок программ закончен. Для завершения приложения SheetApplication:

1. Выберите команду меню **View | Solution Explorer** (Вид | Проводник решения) или нажмите комбинацию клавиш <Ctrl>+<Alt>+<L>. Справа от окна редактирования ресурса диалогового окна раскроется окно **Solution Explorer** (Проводник решения). Это окно может быть раскрыто и щелчком мыши по своему ярлыку.
2. Откройте папку **Sheet**, а в ней откройте папку **Header Files** (Файлы заголовков), как это показано на рис. 3.21.

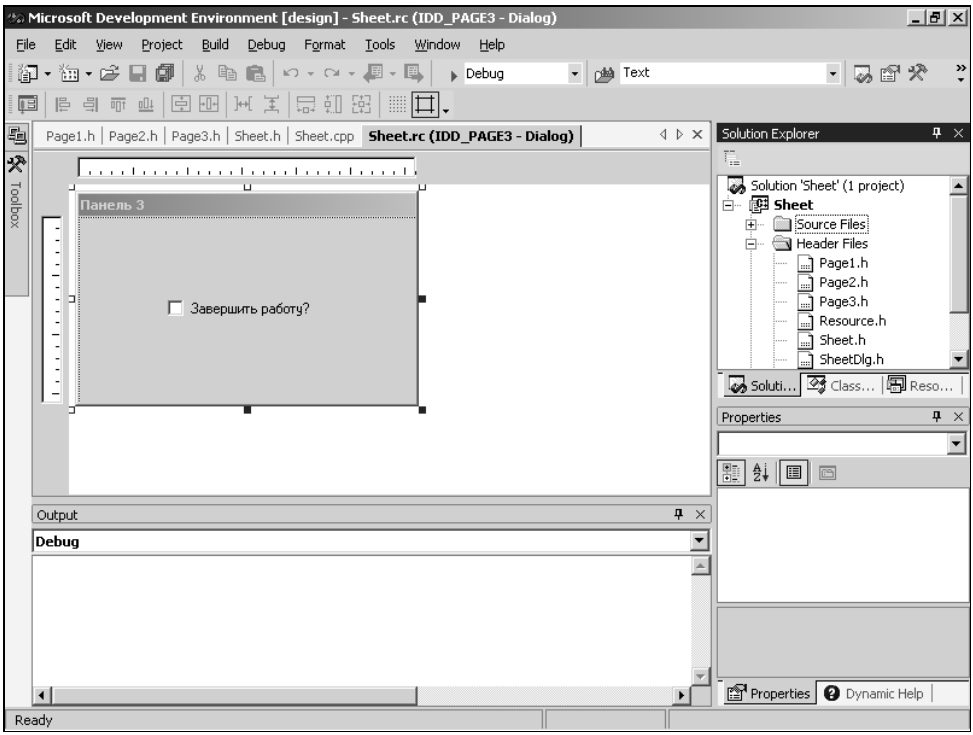


Рис. 3.21. Окно **Solution Explorer**

3. Дважды щелкните левой кнопкой мыши по имени файла Sheet.h. Откроется окно редактирования соответствующего файла.
4. Перед объявлением класса `CSheet` удалите строку `#pragma once` и вставьте операторы включения файлов заголовков классов вкладок диалогового окна:

```
#include "Page1.h"
#include "Page2.h"
#include "Page3.h"
```

5. В секцию `public:` заголовка класса `CSheet` вставьте операторы объявления данных — членов класса, представляющих собой объекты классов вкладок:

```
CPage1  m_page1;
CPage2  m_page2;
CPage3  m_page3;
```

6. В окне **Solution Explorer** (Проводник решения) раскройте папку **Source Files** (Файлы реализации) и дважды щелкните левой кнопкой мыши по имени файла Sheet.cpp.
7. Добавьте в тело функций обоих конструкторов класса `CSheet` операторы включения вкладок в диалоговое окно:

```
// Включение вкладок в диалоговое окно

AddPage(&m_page1);
AddPage(&m_page2);
AddPage(&m_page3);
```

8. В том же файле в теле функции `CSheetApp::InitInstance` замените оператор `CSheetDlg dlg;` оператором `CSheet dlg("Диалоговое окно с вкладками");`.
9. Нажмите клавишу <F5> и запустите проект на исполнение. Появится окно сообщения **Microsoft Development Environment**, сообщающее о необходимости перекомпилировать приложение.
10. Нажмите кнопку **Yes** (Да). Приложение будет скомпилировано и на экране появится диалоговое окно, изображенное на рис. 3.22.

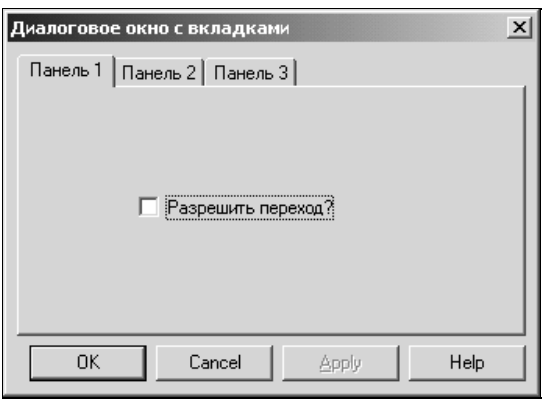


Рис. 3.22. Диалоговое окно с вкладками

Вы можете открыть любую вкладку, но ее содержимое не изменится, поскольку мы ввели во все вкладки идентичную информацию.

Создание мастера

Диалоговое окно мастера так же, как и диалоговое окно с вкладками, позволяет практически полностью менять набор элементов управления диалогового окна, но, в отличие от окна с вкладками, последовательность перехода от одного окна к другому в нем четко зафиксирована, а сам переход происходит посредством нажатия кнопок **Next** (Следующий) и **Back** (Назад).

Окно мастера можно рассматривать как разновидность диалогового окна с вкладками. Чтобы продемонстрировать это, мы используем описанное выше приложение Sheet.

Чтобы превратить приложение Sheet в приложение, демонстрирующее возможности диалогового окна мастера:

1. В файле Sheet.cpp в теле функции `CSheetApp::InitInstance` после оператора `CSheet dlg("Диалоговое окно с вкладками");` вставьте оператор

   ```
   dlg.SetWizardMode();
   ```

2. Раскройте окно **Class View** (Просмотр классов), а в нем раскройте папку **Sheet**.
3. Щелкните правой кнопкой мыши на папке `CPage1` и выберите в появившемся контекстном меню команду **Properties** (Свойства). Раскроется окно **Properties** (Свойства).
4. Нажмите кнопку **Overrides** (Перегружаемые функции базовых классов) в окне **Properties** (Свойства). Раскроется список виртуальных функций базовых классов.
5. В этом списке выделите строку `OnSetActive`. В соответствующем текстовом поле появится значок раскрывающегося списка.
6. Раскройте этот список, он будет содержать единственную строку, предлагающую добавить функцию обработки сообщения `OnSetActive`, как это показано на рис. 3.23.
7. Выделите эту строку. Она появится в текстовом поле данного раскрывающегося списка.
8. Повторите п.п. 5—7 для включения в класс функции `OnWizardNext`.
9. Повторите п.п. 3—8 для класса `CPage2`, но добавьте в строку дополнительно функцию — член класса `OnWizardBack`.
10. Повторите п.п. 3—8 для класса `CPage3`, включив в него функции — члены классов `OnSetActive` и `OnWizardBack`.
11. Откройте окно редактирования файла Page1.cpp и внесите в его функции изменения, зафиксированные в листинге 3.6.

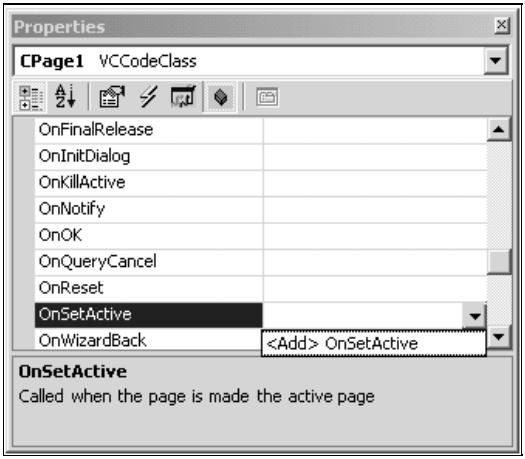


Рис. 3.23. Диалоговое окно **Properties**

Листинг 3.6. Функции обработки сообщений первой панели мастера

```
// Вызывается при переходе к данной вкладке

BOOL CPage1::OnSetActive(void)
{
  // Получение указателя на родительское окно
  CPropertySheet*  parent = (CPropertySheet*) GetParent();

  // Установка кнопок в родительском окне
  parent-> SetWizardButtons(PSWIZB_NEXT);

  return CPropertyPage::OnSetActive();
}

// Вызывается при нажатии кнопки Next>
LRESULT CPage1::OnWizardNext(void)
{
  // Получение указателя на флажок
  CButton*  checkBox = (CButton*) GetDlgItem( IDC_CHECK );
  if( !checkBox->GetCheck() )  // Проверка состояния флажка
  {
    MessageBox("Установите флажок");
```

```
    return -1;
  }
  return CPropertyPage::OnWizardNext();
}
```

12. Откройте файл Page2.cpp и внесите в его функции изменения, зафиксированные в листинге 3.7.

Листинг 3.7. Функции обработки сообщений второй панели мастера

```
// Вызывается при переходе к данной вкладке

BOOL CPage2::OnSetActive(void)
{
  // Получение указателя на родительское окно
  CPropertySheet*  parent = (CPropertySheet*) GetParent();

  // Установка кнопок в родительском окне
  parent-> SetWizardButtons( PSWIZB_NEXT  |  PSWIZB_BACK);
  return CPropertyPage::OnSetActive();
}

// Вызывается при нажатии кнопки <Back
LRESULT CPage2::OnWizardBack(void)
{
  // Получение указателя на флажок
  CButton*  checkBox = (CButton*) GetDlgItem( IDC_CHECK );
  if( !checkBox->GetCheck() )
  {
    MessageBox("Установите флажок");
    return -1;
  }
  return CPropertyPage::OnWizardBack();
}

// Вызывается при нажатии кнопки Next>
LRESULT CPage2::OnWizardNext(void)
{
  // Получение указателя на флажок
  CButton*  checkBox = (CButton*) GetDlgItem( IDC_CHECK );
```

```
  if( !checkBox->GetCheck() )  // Проверка состояния флажка
  {
    MessageBox("Установите флажок");
    return -1;
  }
  return CPropertyPage::OnWizardNext();
}
```

13. Откройте файл Page3.cpp и внесите в его функции изменения, зафиксированные в листинге 3.8.

Листинг 3.8. Функции обработки сообщений третьей панели мастера

```
// Вызывается при переходе к данной вкладке
BOOL CPage3::OnSetActive(void)
{
  // Получение указателя на родительское окно
  CPropertySheet*  parent = (CPropertySheet*) GetParent();

  // Установка кнопок в родительском окне
  parent-> SetWizardButtons(PSWIZB_BACK);
  return CPropertyPage::OnSetActive();
}

// Вызывается при нажатии кнопки <Back
LRESULT CPage3::OnWizardBack(void)
{
  // Получение указателя на флажок
  CButton*  checkBox = (CButton*) GetDlgItem( IDC_CHECK );
  if( !checkBox->GetCheck() )  // Проверка состояния флажка
  {
    MessageBox("Установите флажок");
    return -1;
  }
  return CPropertyPage::OnWizardBack();
}
```

14. Запустите приложение на исполнение. Появится диалоговое окно мастера, изображенное на рис. 3.24.

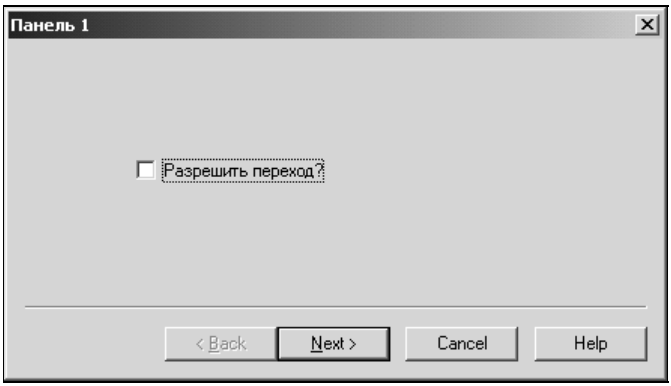


Рис. 3.24. Диалоговое окно мастера. **Панель 1**

15. Нажмите на кнопку **Next** (Далее). Появится окно сообщения, изображенное на рис. 3.25.

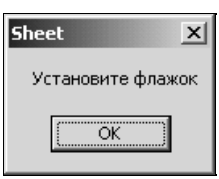


Рис. 3.25. Окно сообщения

16. Нажмите кнопку **OK**, установите флажок **Разрешить переход?** и снова нажмите кнопку **Next** (Далее). Появится диалоговое окно мастера, изображенное на рис. 3.26.

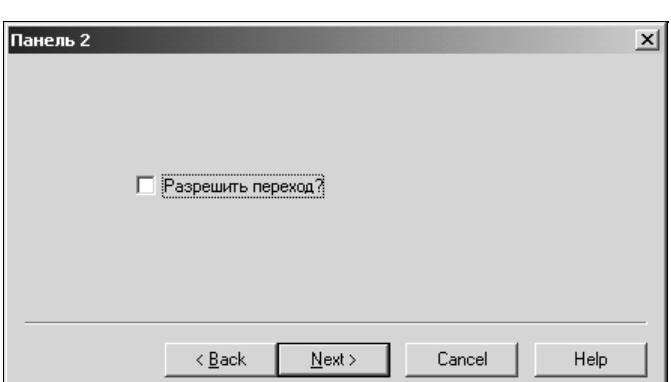


Рис. 3.26. Диалоговое окно мастера. **Панель 2**

17. Установите флажок **Разрешить переход?** и нажмите кнопку **Next** (Далее). Появится диалоговое окно мастера, изображенное на рис. 3.27.

Рис. 3.27. Диалоговое окно мастера. **Панель 3**

18. Установите флажок **Разрешить переход?** и нажмите кнопку **Back** (Назад). Появится диалоговое окно мастера, изображенное на рис. 3.25, но с установленным флажком.

Для превращения диалогового окна со вкладками в окно мастера перед вызовом функции `DoModal` вызывается функция `SetWizardMode`.

Как видно из сравнения рисунков, в диалоговом окне **Панель1** недоступна кнопка **Back** (Назад), в диалоговом окне **Панель3** недоступна кнопка **Next** (Далее), а в диалоговом окне **Панель2** доступны обе эти кнопки. Доступность этих кнопок определяется функцией `CPropertySheet::SetWizardButtons`, обычно вызываемой из функции `CPropertyPage::OnSetActive`, поскольку данная функция может вызываться только после открытия диалога, т. е. после вызова функции `DoModal`. Функция `SetWizardButtons` определяет функцию и внешний вид кнопок диалогового окна мастера. В качестве ее параметра выступает набор флагов, объединенных функцией логического ИЛИ, этот набор включает в себя следующие флаги:

- `PSWIZB_BACK` — делает доступной кнопку **Back** (Назад);
- `PSWIZB_NEXT` — отображает и делает доступной кнопку **Next** (Далее);
- `PSWIZB_FINISH` — отображает и делает доступной кнопку **Finish** (Готово);
- `PSWIZB_DISABLEFINISH` — делает недоступной кнопку **Finish** (Готово).

Следует отметить, что флаги `PSWIZB_NEXT` и `PSWIZB_FINISH` являются несовместимыми, поскольку относятся к одной и той же физической кнопке. Поэтому панель может иметь или кнопку **Next** (Далее) или кнопку **Finish** (Готово), но не обе сразу. Флаг `PSWIZB_DISABLEFINISH` может появиться только в комбинации с флагом `PSWIZB_FINISH`.

Некоторые модификации окна мастера

Окно мастера является очень удобной модификацией диалогового окна, но оно нацелено на использование в одном, достаточно конкретном случае, когда последовательность смены окон определена заранее и не может меняться ни при каких обстоятельствах. Однако на практике часто возникает необходимость определять последовательность перехода между окнами в процессе работы с диалоговым окном, например при реализации окна обработок, когда в первой панели стоит переключатель, определяющий тип обработки, и в зависимости от его положения нажатие кнопки **Next** (Далее) приводит к появлению панели, устанавливающей параметры соответствующей обработки.

Изменим наше приложение таким образом, чтобы из первой панели переход осуществлялся сразу в третью панель, и уже из нее во вторую. Кроме того, сделаем так, чтобы при установке флага в третьей панели работа мастера считалась завершенной, и пользователь уже не мог бы переходить к другим панелям, а мог только завершить работу с диалоговым окном мастера.

Чтобы внести все необходимые для этого изменения в проект:

1. Раскройте окно **Resourse View** (Просмотр ресурсов), а в нем папку Sheet.rc.
2. Раскройте папку **Dialog** (Диалоговое окно) и дважды щелкните левой кнопкой мыши по значку `IDD_PAGE3`. Появится окно редактирования ресурса.
3. Щелкните левой кнопкой мыши на флажке **Разрешить переход?** и раскройте окно **Properties** (Свойства).
4. Введите в текстовое поле **Caption** (Заголовок) заголовок "Завершить работу?".
5. В раскрывающемся списке **Auto** группы **Behavior** (Поведение) выделите строку **False** (Ложь). Это позволит получить доступ к сообщениям, посылаемым данным элементом управления.
6. Нажмите кнопку **ControlEvents** (События элементов управления). Раскроется список сообщений, посылаемых флажком.
7. Выделите в нем сообщение `BN_CLICKED` и раскройте связанный с ним список. В нем будет предложено имя функции обработки сообщения `OnClickedCheck`.
8. Выделите эту единственную строку в раскрывающемся списке и закройте его. Выделенная функция появится в текстовом поле раскрывающегося списка.
9. Раскройте окно **Class View** (Просмотр классов), а в нем — папку **Sheet**.
10. Щелкните правой кнопкой мыши на папке `CPage3` и выберите в появившемся контекстном меню команду **Properties** (Свойства). Раскроется окно **Properties** (Свойства).
11. Нажмите кнопку **Overrides** (Перегружаемые функции базовых классов) в окне **Properties** (Свойства). Раскроется список виртуальных функций базовых классов.
12. В этом списке выделите строку `OnWizardNext`. В соответствующем текстовом поле появится значок раскрывающегося списка.

13. Раскройте этот список, он будет содержать единственную строку, предлагающую добавить функцию обработки сообщения `OnWizardNext`.
14. Выделите эту строку. Она появится в текстовом поле данного раскрывающегося списка.
15. Раскройте окно редактирования файла Page1.cpp. Для этого достаточно щелкнуть левой кнопкой мыши на ярлыке этого окна.
16. Измените тело функции `CPage1::OnWizardNext` в соответствии с содержимым листинга 3.9.

Листинг 3.9. Функция `CPage1::OnWizardNext`

```
// Вызывается при нажатии кнопки Next>
LRESULT CPage1::OnWizardNext(void)
{
  // Получение указателя на родительское окно
  CSheet*    parent = (CSheet*) GetParent();

  // Получение указателя на флажок
  CButton*  checkBox = (CButton*) GetDlgItem( IDC_CHECK );
  if( !checkBox->GetCheck() )  // Проверка состояния флажка
  {
    MessageBox("Установите флажок");
    return -1;
  }

  // Установка страницы, на которую производится переход
  parent-> SetActivePage( parent-> GetPageIndex( &parent-> m_page3) — 1 );
  return CPropertyPage::OnWizardNext();
}
```

17. Раскройте окно редактирования файла Page3.cpp.
18. В текст программы внесите изменения, содержащиеся в листинге 3.10.

Листинг 3.10. Функция обработки сообщений класса `CPage3`

```
// Вызывается при переходе к данной вкладке
BOOL CPage3::OnSetActive(void)
{
  CPropertySheet*  parent = (CPropertySheet*) GetParent();
  parent-> SetWizardButtons(PSWIZB_NEXT  |  PSWIZB_BACK);
```

```
  return CPropertyPage::OnSetActive();
}

// Вызывается при нажатии кнопки <Back
LRESULT CPage3::OnWizardBack(void)
{
  // Получение указателя на родительское окно
  CSheet*    parent = (CSheet*) GetParent();

  // Установка страницы, на которую производится переход
  parent-> SetActivePage( parent-> GetPageIndex( &parent-> m_page1) + 1 );

  return CPropertyPage::OnWizardBack();
}

// Выводит кнопку Finish

void CPage3::OnClickedCheck(void)
{
  // Получение указателя на родительское окно
  CSheet*    parent = (CSheet*) GetParent();

  // Получение указателя на флажок
  CButton*  checkBox = (CButton*) GetDlgItem( IDC_CHECK );
  if( !checkBox->GetCheck() )  // Переключение флажка
    checkBox->SetCheck( 0 );
  else
    checkBox->SetCheck( 1 );
  parent->SetFinishText("Finish");
}

// Вызывается при нажатии кнопки Next>
LRESULT CPage3::OnWizardNext(void)
{
  // Получение указателя на родительское окно
  CSheet*    parent = (CSheet*) GetParent();

  // Установка страницы, на которую производится переход
  parent-> SetActivePage( parent-> GetPageIndex( &parent-> m_page2) — 1 );

  return CPropertyPage::OnWizardNext();
}
```

19. Запустите приложение на исполнение. Появится диалоговое окно мастера, изображенное на рис. 3.24.

На первый взгляд ничего не изменилось, но при нажатии кнопки **Next** (Далее) вы переходите к панели 3, а не к панели 2, как в предыдущем случае. Обратите внимание на то, что в этой панели разблокирована кнопка **Next** (Далее), при нажатии на которую вы переходите в панель 2. Порядок перехода из панели 2 остался неизменным. Если в панели 3 вы установите флажок **Завершить работу?**, то эта панель примет вид, изображенный на рис. 3.28.

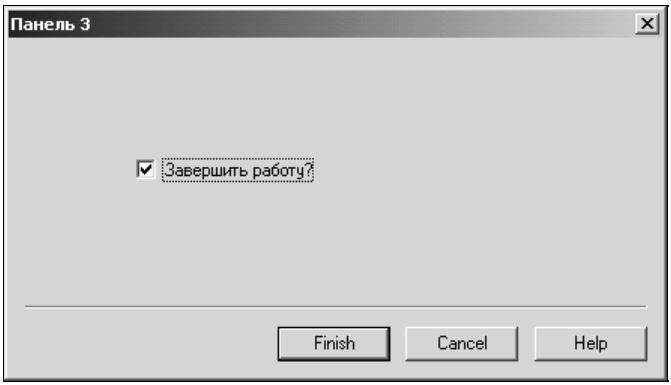


Рис. 3.28. Результат установки флажка **Завершить работу?**

Произвольный переход между панелями достигается использованием функции `CPropertySheet::SetActivePage`. Данная функция возвращает логическую величину, принимающую ненулевое значение в случае успешной активизации вкладки, и нулевое значение в противном случае. В качестве аргумента этой функции может выступать индекс устанавливаемой вкладки или непосредственно указатель на объект класса данной вкладки. Индекс вкладки может принимать любое положительное значение, включая 0, не превышающее количества вкладок, включенных в данный мастер. Порядок нумерации вкладок соответствует порядку их включения в класс мастера функциями `AddPage`. В нашем случае следует учитывать, что функции `SetActivePage` вызываются в функциях `OnWizardNext` и `OnWizardBack`, которые сами осуществляют переход между вкладками, поэтому при указании индекса вкладки необходимо учесть результат выполнения функции, из которой производится вызов.

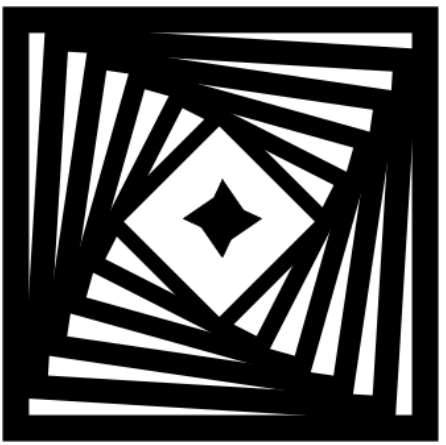
Приведенная программа также демонстрирует результат вызова функции `CPropertySheet::SetFinishText`. По своему названию эта функция служит для определения текста, отображаемого в кнопке **Finish** (Готово). Однако основное назначение этой функции заключается в том, что она убирает из окна мастера кнопки **Back** (Назад) и **Next** (Далее) и заменяет их одной кнопкой **Finish** (Готово) (точнее кнопкой с именем, указанным в аргументе функции), в которой будет отображаться текст, введенный в функцию в качестве параметра. Дан-

ная функция имеет необратимый эффект и после установки флажка **Завершить работу?** его сброс или повторная установка не приведут к появлению кнопок **Back** (Назад) и **Next** (Далее). Обычно эта функция вызывается пользователем после завершения всех процедур, предусмотренных в мастере. Действия, которые будут предприняты при нажатии кнопки **Finish** (Готово), определяются функцией `OnWizardFinish`, перегрузка которой осуществляется аналогично перегрузке других виртуальных функций базовых классов.

На функции `OnWizardFinish` следует остановиться подробнее. Основной вопрос заключается в том, где ее вызывать. Одним из возможных мест вызова данной функции является функция `OnSetActive` последней вкладки мастера, где она может заменить функцию `SetWizardButtons`. После такого вызова окно мастера, в котором будет отображаться последняя вкладка, будет содержать только кнопки **Finish** (Готово) и **Cancel** (Отмена) (кнопка **Help** (Справка) является факультативной). Но в этом случае пользователь лишается возможности внести изменения в установки, произведенные им на предыдущих стадиях настройки мастера. Поскольку на последней странице мастера часто помещается сводка произведенных пользователем установок, то отсутствие на ней кнопки **Back** (Назад) в этом случае можно расценить как тонкую форму издевательства над пользователем.

Поэтому кнопку **Finish** (Готово) лучше всего устанавливать в том случае, когда установки, произведенные им на предыдущих стадиях настройки мастера, используются как параметры некоторой достаточно длительной процедуры, которая может быть в любой момент прервана пользователем с сохранением результатов обработки. В этом случае кнопка **Finish** (Готово) появляется после запуска этой обработки и служит для ее завершения без потери результата или выхода из диалогового окна после завершения обработки, если в результате проведенной обработки в диалоговое окно выводится информация, которая должна быть оценена пользователем.

При рассмотрении данного приложения следует обратить внимание на один момент, не связанный напрямую с рассматриваемым вопросом. Дело в том, что в третьей вкладке мастера функция `OnWizardFinish` вызывается в функции обработки сообщения об изменении состояния флажка. Для доступа к этой функции нам пришлось отказаться от автоматической обработки сообщений данного элемента управления. Поэтому в этой функции не только вызывается функция `OnWizardFinish`, но и выполняются все операции, производимые при обработке этого сообщения. В данном случае — это отображение и удаление флажка с экрана. Для этого используются функции `CButton::GetCheck` (для получения текущего состояния флажка) и `CButton::SetCheck` (для изменения его на противоположное).



ЧАСТЬ II

Программирование интерфейса пользователя

Глава 4. Классы элементов управления

Глава 5. Сообщения и команды

Глава 6. Вывод информации на экран

Глава 7. Работа с файлами документов

Глава 8. Работа с текстовыми документами

Глава 9. Панели инструментов и строка состояния

Глава 10. Печать документов и организация прокрутки в окне

Глава 4

Классы элементов управления

В *главе 3* было описано создание класса диалога и рассмотрены простейшие элементы управления. Эти элементы управления, как правило, представляют собой кнопку или текстовое поле и работа с ними понятна на интуитивном уровне. Однако существуют элементы управления, реализация которых описывается достаточно сложными классами, каждый из которых нуждается в отдельном описании. Ниже будут рассмотрены основные классы элементов управления.

Основным отличием классов элементов управления от всех остальных является то, что объект данного класса может быть сформирован только после того, как соответствующее диалоговое окно будет выведено на экран. Нельзя вызывать функции — члены класса до того, как будет порожден объект класса диалогового окна. Поэтому все операции с объектами данных классов должны производиться в функциях — членах класса родительского диалогового окна или в вызываемых ими функциях. Это является гарантией того, что объект класса диалогового окна будет существовать на момент подобного вызова.

Класс списка

Для демонстрации возможностей элемента управления **List** создано приложение List, которое можно найти в папке List дискеты, поставляемой с данной книгой. Чтобы самостоятельно создать тестовое приложение для исследования возможностей класса списка:

1. Создайте заготовку диалогового приложения по методике, описанной в *главе 1*, и назовите его List.
2. Раскройте папку **Dialog** (Диалог) в окне **Resource View** (Просмотр ресурсов). Щелкните правой кнопкой мыши на идентификаторе ресурса `IDD_LIST_DIALOG` и в раскрывшемся при этом контекстном меню выберите команду **Properties** (Свойства). Раскроется окно **Properties** (Свойства).
3. Щелкните левой кнопкой мыши на строке **Language** (Язык). В соответствующем текстовом поле появится значок раскрывающегося списка.
4. Раскройте этот список и выделите в нем строку **Russian** (Русский).
5. Закройте окно **Output** (Выход) и растяните заготовку диалогового окна.
6. В текстовое поле **Caption** (Заголовок) окна **Properties** (Свойства) введите заголовок "Демонстрация окна списка".

7. Щелкните левой кнопкой мыши на статическом тексте в заготовке диалогового окна и введите в текстовое поле **Caption** (Заголовок) окна **Properties** (Свойства) заголовок "Список".
8. Перетащите статический текст в левый верхний угол заготовки диалогового окна.
9. В окне **Toolbox** (Инструментарий) выберите элемент управления **List Box** (Список) и поместите его под статическим текстом, растянув его по вертикали на все оставшееся свободное место, а по горизонтали — примерно на половину заготовки диалогового окна. Окно редактирования ресурса примет вид, изображенный на рис. 4.1.

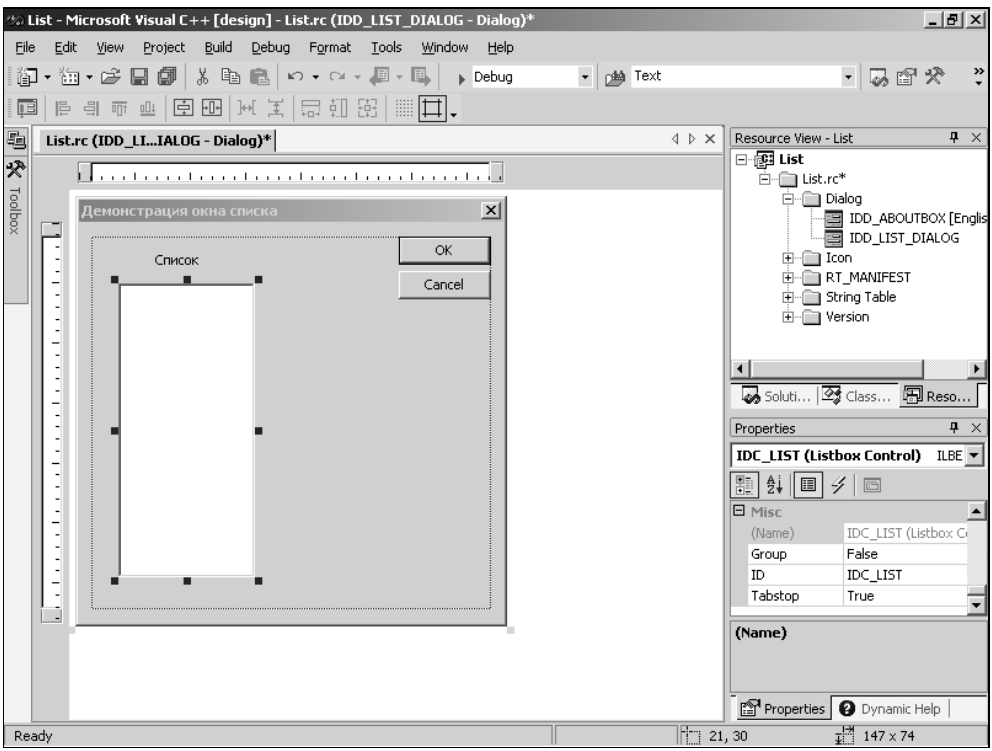


Рис. 4.1. Окно редактирования ресурса с окном списка

10. В окне **Toolbox** (Инструментарий) выберите элемент управления **Static Text** (Статический текст) и поместите его в заготовку диалогового окна под кнопками.
11. В текстовое поле **Caption** (Заголовок) окна **Properties** (Свойства) введите заголовок "Элемент списка".

12. В окне **Toolbox** (Инструментарий) выберите элемент управления **Edit Box** (Окно редактирования) и поместите его в заготовку диалогового окна под только что введенный статический текст.
13. В окне **Properties** (Свойства) измените идентификатор ресурса на `IDC_EDIT`. Окно редактирования ресурса диалогового окна примет вид, изображенный на рис. 4.2.

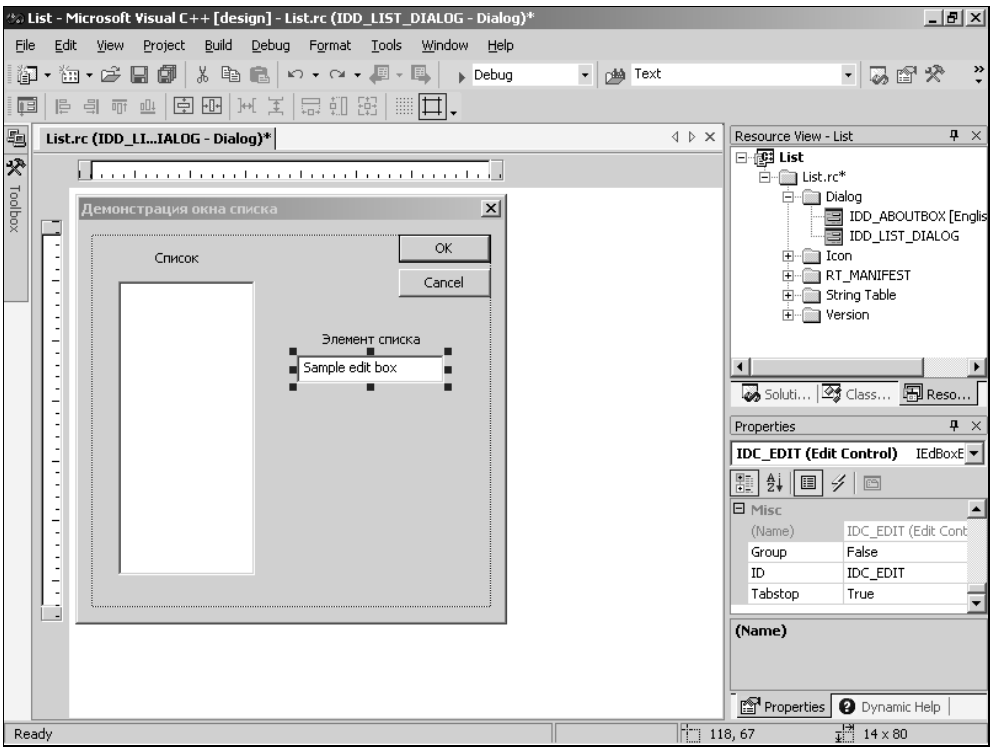


Рис. 4.2. Окно редактирования ресурса с окном списка и текстовым полем

14. В окне **Toolbox** (Инструментарий) выберите элемент управления **Button** (Кнопка) и поместите его в заготовку диалогового окна под текстовое поле.
15. Введите в текстовое поле **ID** (Идентификатор ресурса) окна **Properties** (Свойства) идентификатор ресурса `IDC_ADD`, а в текстовое поле **Caption** (Заголовок) введите заголовок "&Добавить".
16. Повторите операции, описанные в п.п. 14 и 15, использовав идентификатор ресурса `IDC_DELETE` и заголовок "&Удалить". В результате этих действий заготовка диалогового окна примет вид, изображенный на рис. 4.3.

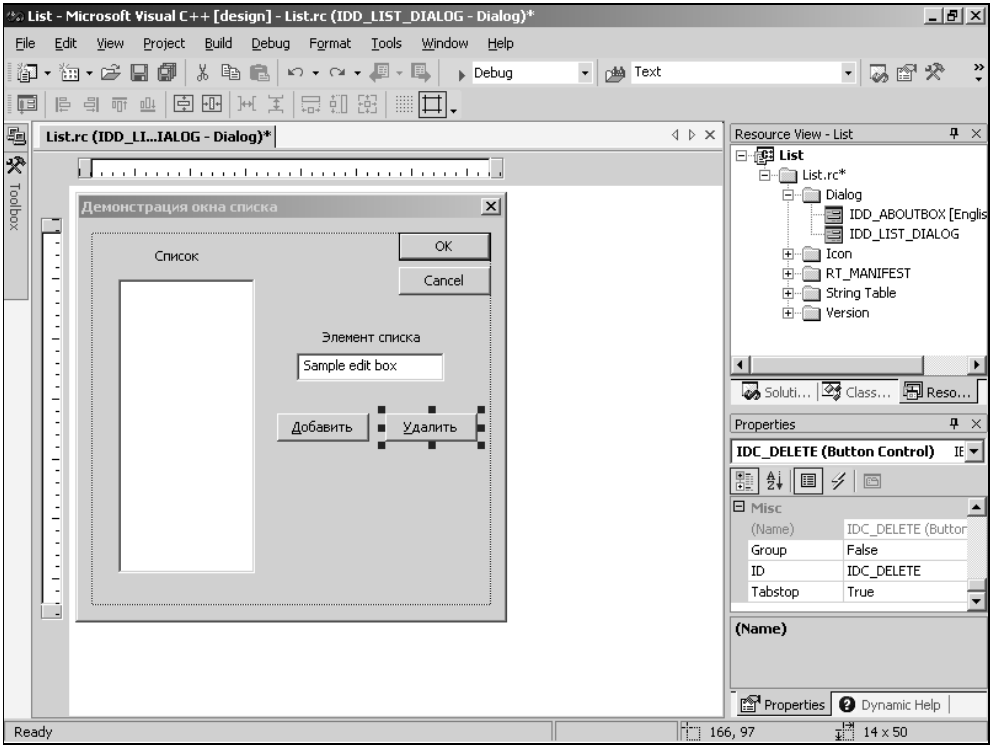


Рис. 4.3. Заготовка диалогового окна с окном списка

17. Щелкните левой кнопкой мыши по кнопке **Cancel** (Отмена) и введите в текстовом поле **Caption** (Заголовок) окна **Properties** (Свойства) заголовок "Отмена".

18. В заготовке диалогового окна выделите элемент управления `IDC_LIST` и нажмите кнопку **ControlEvents** (События элементов управления) в окне **Properties** (Свойства). Раскроется список сообщений, посылаемых данным элементом управления.

19. В раскрывшемся списке щелкните левой кнопкой мыши на строке `LBN_DBLCLK`. В соответствующем текстовом поле появится значок раскрывающегося списка.

20. Раскройте этот список, он будет содержать единственную строку, предлагающую добавить функцию обработки сообщения `OnLbnDblclkList`, как это показано на рис. 4.4.

21. Выделите эту строку. Имя функции обработки сообщения появится в текстовом поле, в окно редактирования будут добавлены панели окон редактирования файла заголовка и файла реализации класса `CListDlg`, окно редак-

тирования файла реализации будет открыто и текстовый курсор будет в нем помещен в заготовку создаваемой функции.

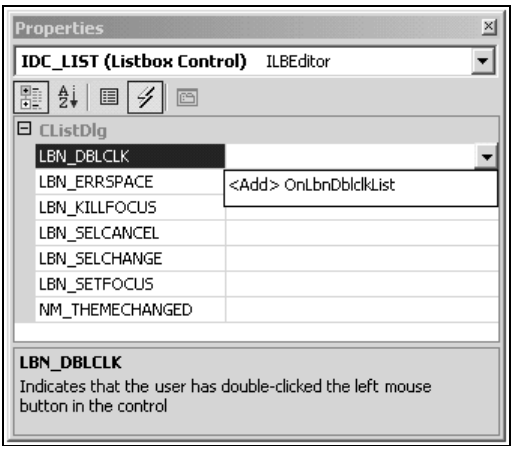


Рис. 4.4. Диалоговое окно **Properties**

22. Откройте окно редактирования ресурса диалогового окна, выделите в его заготовке кнопку **Добавить** и нажмите кнопку **ControlEvents** (События элементов управления) в окне **Properties** (Свойства). Раскроется список сообщений, посылаемых данным элементом управления.
23. В раскрывшемся списке щелкните левой кнопкой мыши на строке `BN_CLICKED`. В соответствующем текстовом поле появится значок раскрывающегося списка.
24. Раскройте этот список, он будет содержать единственную строку, предлагающую добавить функцию обработки сообщения `OnBnClickedAdd`.
25. Выделите эту строку. Имя функции обработки сообщения появится в текстовом поле.
26. Повторите п.п. 22—25 для кнопки **Удалить**.
27. Откройте окно **Class View** (Просмотр классов), а в нем — список **List**.
28. Щелкните правой кнопкой мыши на папке `CListDlg` и выберите в появившемся контекстном меню команду **Add | Add Variable** (Добавить | Добавить переменную). Появится диалоговое окно **Add Member Variable Wizard - List** (Мастер добавления переменной в класс), изображенное на рис. 4.5.
29. Установите флажок **Control variable** (Связь с элементом управления). Раскрывающийся список **Control ID** (Идентификатор элемента управления) и текстовое поле **Control type** (Тип элемента управления) станут доступными.
30. В раскрывающемся списке **Control ID** (Идентификатор элемента управления) выделите идентификатор ресурса `IDC_EDIT`.

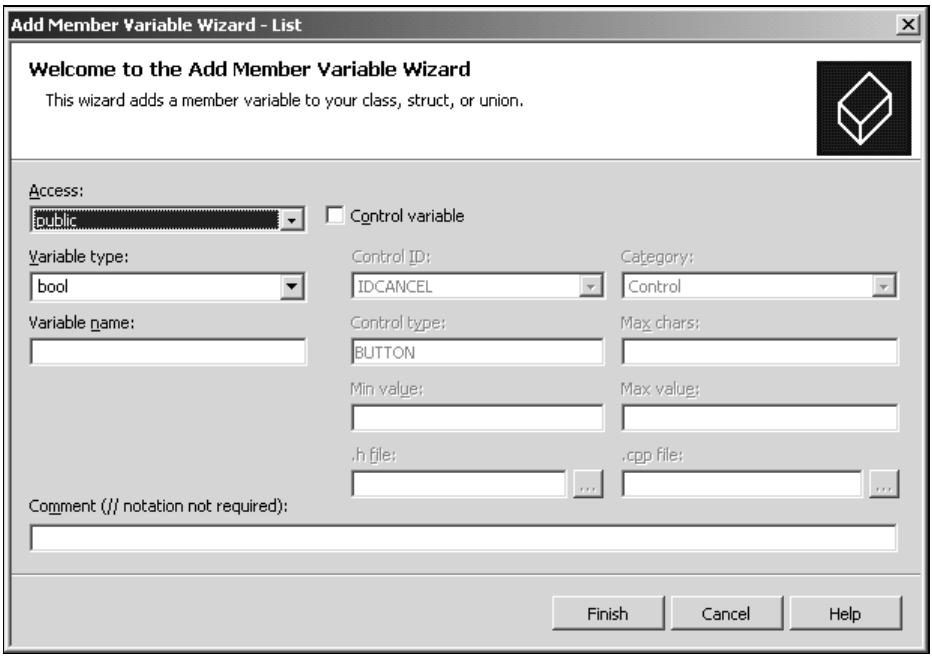


Рис. 4.5. Добавление переменной в класс

31. В раскрывающемся списке **Category** (Категория) выделите строку **Value** (Переменная), введите в текстовое поле **Variable name** (Имя переменной) идентификатор переменной `m_Edit` и нажмите кнопку **Finish** (Готово).
32. Повторите п.п. 28 и 29 и выделите в раскрывающемся списке **Control ID** (Идентификатор элемента управления) идентификатор ресурса `IDC_LIST`.
33. Оставьте переключатель в положении **Control** (Управление), введите в текстовое поле **Variable name** (Имя переменной) идентификатор переменной `m_List` и нажмите кнопку **Finish** (Готово).
34. Откройте окно редактирования файла ListDlg.cpp и внесите в него изменения в соответствии с листингом 4.1.

Листинг 4.1. Функции обработки сообщений класса `CListDlg`

```
// Обрабатывает сообщение о двойном щелчке мыши

void CListDlg::OnLbnDblclkList()
{
  CString  Temp;
  if( m_List.GetCurSel() != LB_ERR )  // Проверка наличия выделения
```

```
  {
    // Чтение строки списка
    m_List.GetText(m_List.GetCurSel(), Temp);

    // Вывод ее в текстовое поле
    SetDlgItemText(IDC_EDIT, Temp);
  }
}

// Обрабатывает нажатие кнопки Add
void CListDlg::OnBnClickedAdd()
{
  UpdateData();       // Чтение состояния элементов
                      // управления диалогового окна
  m_List.AddString(m_Edit);  // Добавление новой строки
}

// Обрабатывает нажатие кнопки Delete
void CListDlg::OnBnClickedDelete()
{
  if( m_List.GetCurSel() != LB_ERR )   // Проверка наличия выделения
    m_List.DeleteString(m_List.GetCurSel());  // Удаление выделения
  UpdateData();                        // Синхронизация с классом
}
```

35. Нажмите клавишу <F5> и ответьте утвердительно на вопрос о необходимости обновления проекта в появившемся после этого окне сообщения. Начнется компиляция программы, по завершении которой она будет запущена на исполнение. Появится диалоговое окно, изображенное на рис. 4.6.
36. Введите в текстовое поле **Элемент списка** текст "Первый" и нажмите кнопку **Добавить**. Этот текст появится в списке.
37. Введите в текстовое поле **Элемент списка** текст "Второй" и нажмите кнопку **Добавить**. Этот текст появится в списке на первом месте.
38. Введите в текстовое поле **Элемент списка** текст "Третий" и нажмите кнопку **Добавить**. Этот текст появится в списке на последнем месте.

    Дело в том, что при создании элемента управления **List Box** (Список) в разделе **Behavior** (Поведение) окна **Properties** (Свойства) в текстовом поле **Sort** (Сортировка) содержалось значение **True** (Истина), поэтому содержимое списка сортируется по алфавиту.
39. Дважды щелкните левой кнопкой мыши по второй строке списка. В текстовом поле **Элемент списка** появится строка "Первый", как это показано на рис. 4.7.

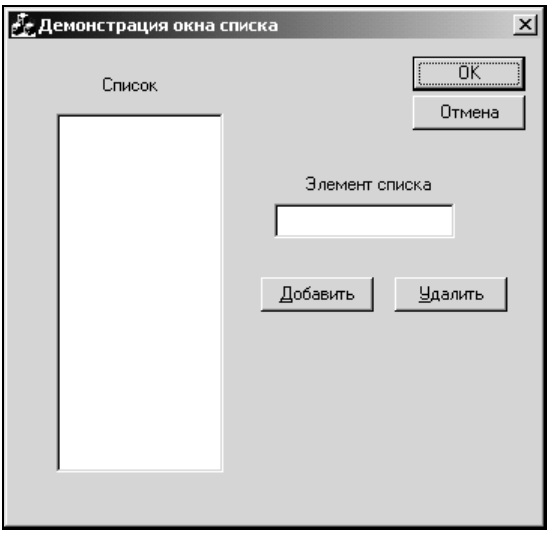


Рис. 4.6. Диалоговое окно списка

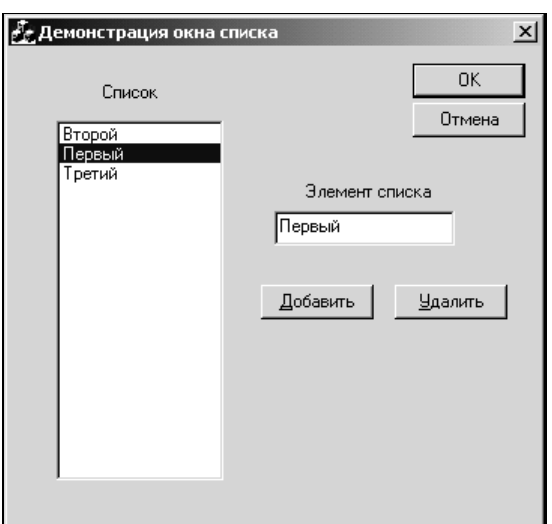


Рис. 4.7. Заполненное диалоговое окно списка

40. Оставив выделение второй строки списка, нажмите кнопку **Удалить**. Эта строка удалится из списка, но останется в текстовом поле, как это показано на рис. 4.8.

В данном примере продемонстрировано применение основных функций работы со списком. Полный набор этих функций достаточно обширен и включает в

себя, помимо прочих, функции для работы с множественным выделением элементов. Подробное их описание можно найти в справочной системе Visual C++. Здесь мы остановимся только на использованных нами функциях.

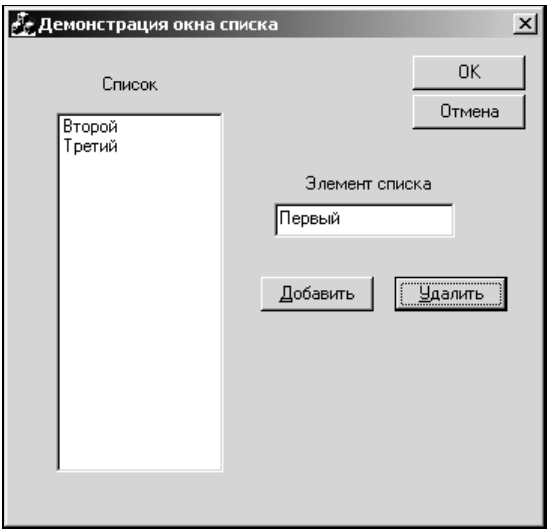


Рис. 4.8. Диалоговое окно списка с удаленным элементом

Функция `OnLbnDblclkList` создает объект класса `CString` с именем `Temp` и, с использованием функции `CListBox::GetText`, записывает в нее содержимое выделенной строки списка. После чего, с использованием функции `CWnd::SetDlgItemText`, содержимое переменной `Temp` записывается в текстовое поле. Если двойной щелчок был произведен на пустой строке списка и ни один элемент при этом не выделился, то чтение элемента списка и запись его в текстовое поле не производятся.

Функция `OnBnClickedAdd` включает в себя вызовы двух функций: функция `UpdateData` производит запись содержимого текстового поля в соответствующую переменную, а функция `CListBox::AddString` добавляет значение этой переменной в список. Функция `OnBnClickedDelete` проверяет наличие в списке выделенной переменной, для чего вызывает функцию `CListBox::GetCurSel`, возвращающую индекс выделенной строки, и проверяет, не содержит ли возвращаемая величина значение ошибки. Если ошибки не обнаружено, то эта функция вызывается еще один раз в качестве аргумента функции `CListBox::DeleteString`, уничтожающей строку, индекс которой указан ей в качестве аргумента.

В случае, когда необходимо установить или снять выделение элемента списка из программы, можно воспользоваться функцией `CListBox::SetSel`, первый аргумент которой представляет собой индекс строки, а второй — является логиче-

ской переменной, которой присваивается значение `TRUE`, если необходимо выделить данную строку списка, и — `FALSE`, если необходимо снять выделение.

Классы линейного регулятора и линейного индикатора

Для демонстрации возможностей линейного регулятора и линейного индикатора было создано демонстрационное приложение Progress, текст которого расположен в одноименной папке на дискете, прилагаемой к данной книге. Чтобы самостоятельно создать это тестовое приложение:

1. Создайте заготовку диалогового приложения по методике, описанной в *главе 1*. И назовите её Progress.
2. Раскройте папку **Dialog** (Диалог) в окне **Resource View** (Просмотр ресурсов). Выделите идентификатор ресурса `IDD_PROGRESS_DIALOG` и нажмите клавишу <F4>. Раскроется окно **Properties** (Свойства).
3. Щелкните левой кнопкой мыши на строке **Language** (Язык). В соответствующем текстовом поле появится значок раскрывающегося списка.
4. Раскройте этот список и выделите в нем строку **Russian** (Русский).
5. Закройте окно **Output** (Окно вывода) и растяните заготовку диалогового окна.
6. В текстовое поле **Caption** (Заголовок) окна **Properties** (Свойства) введите заголовок "Демонстрация линейного регулятора".
7. Удалите с заготовки диалогового окна статический текст.
8. Щелкните левой кнопкой мыши по кнопке **Cancel** (Отмена) и введите в текстовом поле **Caption** (Заголовок) окна **Properties** (Свойства) заголовок **Отмена**.
9. Перетащите кнопки **OK** и **Отмена** в нижнюю часть заготовки диалогового окна.
10. В окне **Toolbox** (Инструментарий) выберите элемент управления **Progress** (Линейный индикатор) и поместите его в заготовку диалогового окна. Измените его горизонтальный размер таким образом, чтобы он занимал бóльшую часть диалогового окна.
11. Введите в текстовое поле раскрывающегося списка **ID** (Идентификатор ресурса) окна **Properties** (Свойства) идентификатор ресурса `IDC_PROGRESS`. Окно редактирования ресурса примет вид, изображенный на рис. 4.9.
12. В окне **Toolbox** (Инструментарий) выберите элемент управления **Slider Control** (Линейный регулятор) и поместите его в заготовку диалогового окна под линейным индикатором.
13. Введите в текстовое поле раскрывающегося списка **ID** (Идентификатор ресурса) окна **Properties** (Свойства) идентификатор ресурса `IDC_SLIDER`.

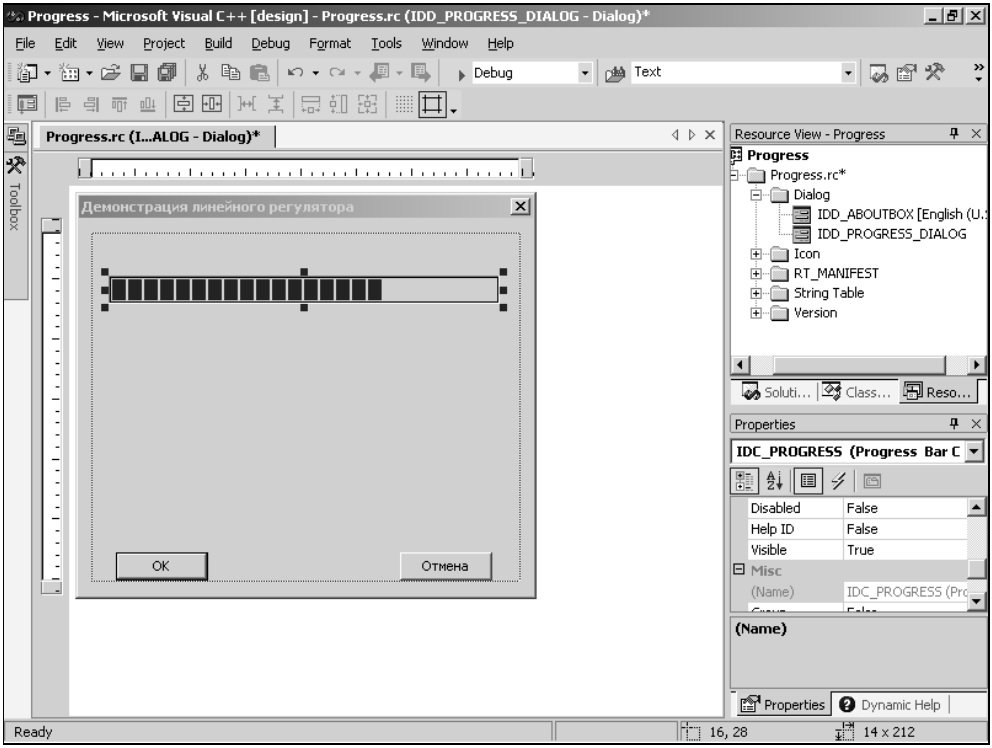


Рис. 4.9. Окно редактирования ресурса с линейным индикатором

14. В раскрывающихся списках **AutoTicks** (Устанавливать автоматически), **ClientEdge** (Рамка) и **TickMarks** (Использовать метки на шкале регулятора) того же окна выделите значение **True** (Истина).
15. Измените вертикальный размер линейного регулятора таким образом, чтобы его вертикальный размер позволял полностью отображать метки, а его горизонтальный размер и положение согласуйте с горизонтальным размером и положением линейного индикатора. Окно редактирования ресурса примет вид, изображенный на рис. 4.10.
16. В окне **Toolbox** (Инструментарий) выберите элемент управления **Picture Control** (Изображение) и поместите его в заготовку диалогового окна под линейным регулятором.
17. Введите в текстовое поле раскрывающегося списка **ID** (Идентификатор ресурса) окна **Properties** (Свойства) идентификатор ресурса `IDC_PICTURE`.
18. В раскрывающемся списке **Sunken** (Углубленный) того же окна выделите значение **True** (Истина).
19. Отрегулируйте размеры и положение данного элемента управления в соответствии с рис. 4.11.

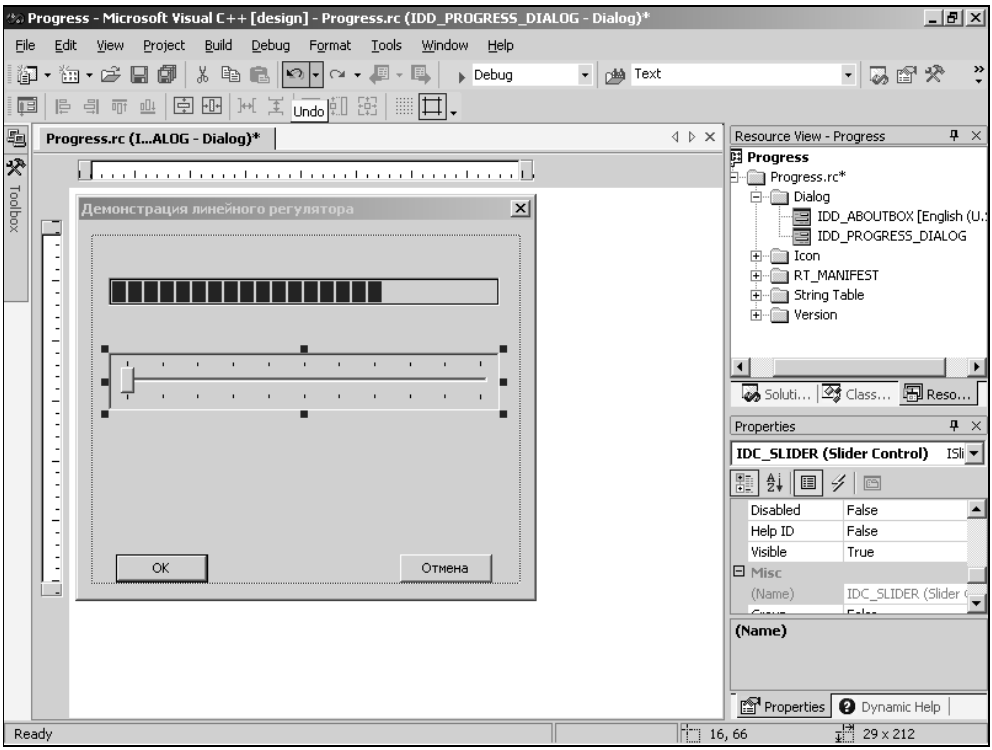


Рис. 4.10. Окно редактирования ресурса с линейным индикатором и линейным регулятором

20. В окне **Toolbox** (Инструментарий) выберите элемент управления **Button** (Кнопка) и поместите его в заготовку диалогового окна между кнопками **OK** и **Отмена**.
21. Введите в текстовое поле раскрывающегося списка **ID** (Идентификатор ресурса) окна **Properties** (Свойства) идентификатор ресурса `IDC_RESET`.
22. Введите в текстовое поле **Caption** (Заголовок) того же окна заголовок "&Сброс". Заготовка диалогового окна примет вид, изображенный на рис. 4.12.
23. Оставив выделение кнопки, нажмите кнопку **ControlEvents** (События элементов управления) в окне **Properties** (Свойства). Раскроется список сообщений, посылаемых данным элементом управления.
24. В этом списке щелкните левой кнопкой мыши на сообщении `BN_CLICKED`. В соответствующем текстовом поле появится значок раскрывающегося списка.
25. Раскройте этот список, он будет содержать единственную строку, предлагающую добавить функцию обработки сообщения `OnBnClickedReset`.

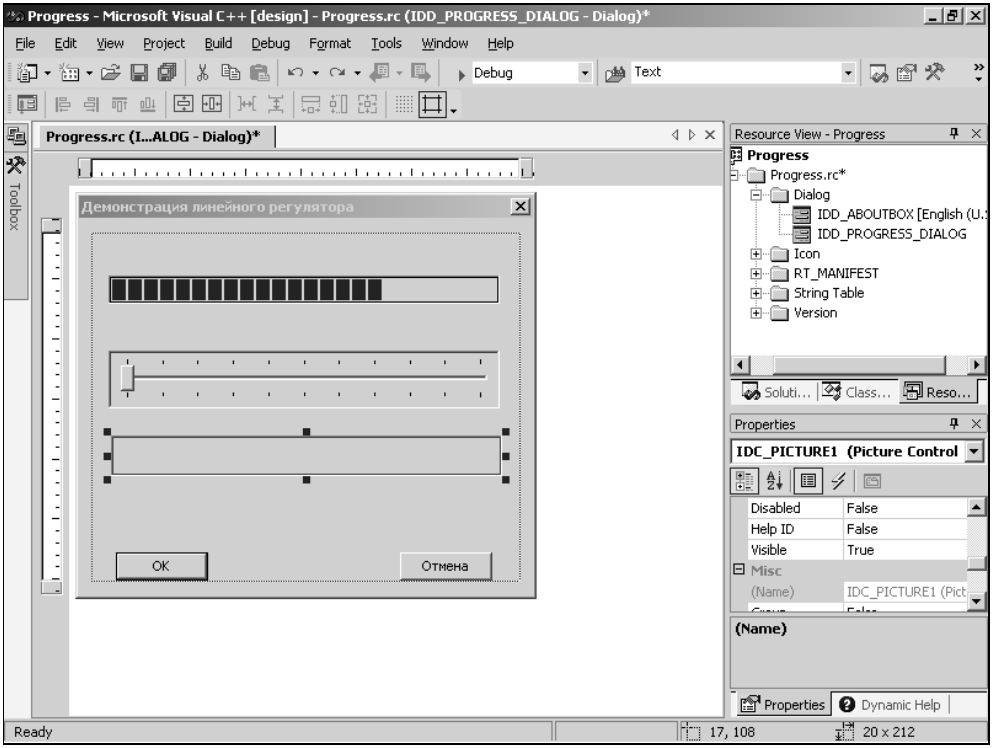


Рис. 4.11. Окно редактирования ресурса

26. Выделите эту строку. Имя функции обработки сообщения появится в текстовом поле.
27. Откройте окно **Class View** (Просмотр классов) и щелкните левой кнопкой мыши на папке `CProgressDlg`.
28. В окне **Properties** (Свойства) нажмите кнопку **Messages** (Сообщения). Откроется список сообщений, посылаемых данному классу.
29. В этом списке щелкните левой кнопкой мыши на сообщении `WM_HSCROLL`. В соответствующем текстовом поле появится значок раскрывающегося списка.
30. Раскройте этот список, он будет содержать единственную строку, предлагающую добавить функцию обработки сообщения `OnHScroll`, как это показано на рис. 4.13.
31. Выделите эту строку. Имя функции обработки сообщения появится в текстовом поле, откроется окно редактирования файла реализации класса `CProgressDlg` и текстовый курсор будет помещен в заготовку данной функции.
32. Щелкните правой кнопкой мыши на папке `CProgressDlg` и выберите в появившемся контекстном меню команду **Add | Add Variable** (Добавить | Доба-

вить переменную). Появится диалоговое окно **Add Member Variable Wizard** (Мастер добавления переменной в класс).

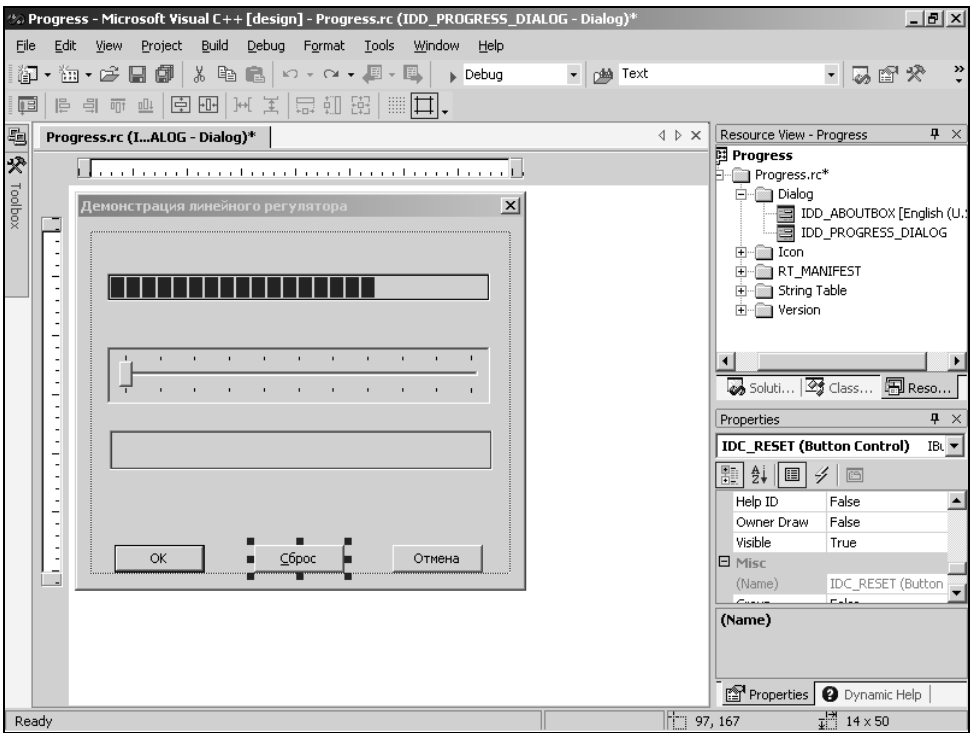


Рис. 4.12. Заготовка диалогового окна

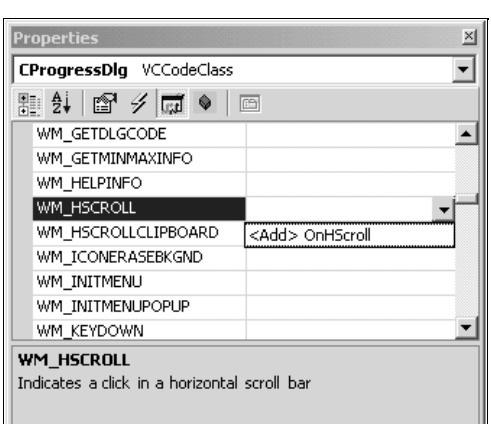


Рис. 4.13. Окно **Properties**

33. Установите флажок **Control variable** (Связь с элементом управления). Раскрывающийся список **Control ID** (Идентификатор элемента управления) и текстовое поле **Control type** (Тип элемента управления) станут доступными.
34. В раскрывающемся списке **Control ID** (Идентификатор элемента управления) выделите идентификатор ресурса `IDC_PROGRESS`.
35. Введите в текстовое поле **Variable name** (Имя переменной) идентификатор переменной `m_Progress` и нажмите кнопку **Finish** (Готово).
36. Повторите п.п. 33—36 для идентификатора ресурса `IDC_SLIDER`, сопоставив ему переменную `m_Slider`.
37. В окне редактирования файла ProgressDlg.cpp внесите изменения в функции обработки сообщений класса `CProgressDlg` в соответствии с листингом 4.2.

Листинг 4.2. Функции обработки сообщений класса `CProgressDlg`

```
// Функции обработки сообщений класса CProgressDlg

BOOL CProgressDlg::OnInitDialog()
{
  CDialog::OnInitDialog();
  // Добавление команды "About..." в системное меню.

  // Идентификатор IDM_ABOUTBOX должен быть определен в области
  // видимости 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);
    }
  }

  // Установка значка диалогового окна. Приложение производит эту
  // операцию автоматически, если главное окно приложения
  // не является диалоговым окном
```

```
  SetIcon(m_hIcon, TRUE);    // Установка больших значков
  SetIcon(m_hIcon, FALSE);   // Установка маленьких значков

  // Инициализация линейного регулятора и линейного индикатора
  m_Slider.SetRange(0, 1024, TRUE);
  m_Slider.SetTicFreq( 128 );
  m_Slider.SetLineSize( 32 );
  m_Slider.SetPageSize( 256 );
  m_Progress.SetRange(0, 1024);

  return TRUE;  // Должно возвращать значение TRUE, если фокус ввода
                // не передан элементу управления
}

// Обрабатывает нажатие кнопки Сброс
void CProgressDlg::OnBnClickedReset()
{
  m_Progress.SetPos( 0 );  // Сброс линейного индикатора
}

// Обработка сообщений линейного регулятора
void CProgressDlg::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar* pScroll-
Bar)
{
  //  Преобразование типа аргумента
  CSliderCtrl*  slider = (CSliderCtrl*) pScrollBar;
  // Проверка текущего положения
  if( m_Progress.GetPos() < slider-> GetPos() )
    m_Progress.SetPos( slider-> GetPos() );  // Перемещение
                                             // линейного индикатора

  CDialog::OnHScroll(nSBCode, nPos, pScrollBar);
}
```

38. Нажмите клавишу <F5> и запустите приложение на исполнение. Появится диалоговое окно, изображенное на рис. 4.14.
39. Поместите указатель мыши на бегунок линейного регулятора, нажмите левую кнопку мыши и, удерживая ее, переместите бегунок вправо. Одновременно с перемещением бегунка вправо начнет заполняться линейный индикатор.

Рис. 4.14. Диалоговое окно приложения

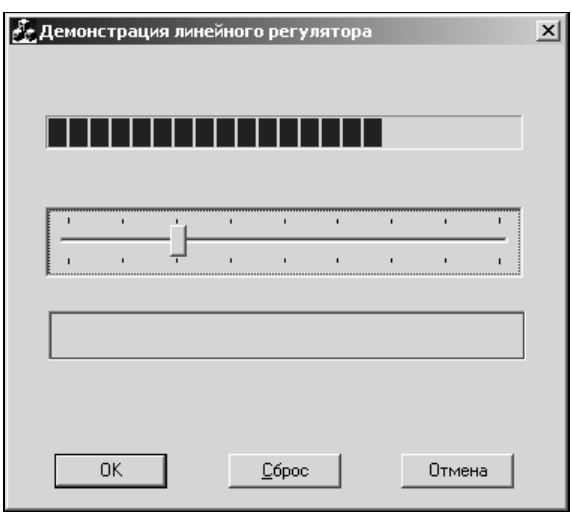


Рис. 4.15. Диалоговое окно приложения в работе

40. Переместите бегунок влево. Линейный индикатор останется в прежнем положении, как это показано на рис. 4.15.
41. Переместите бегунок еще дальше вправо. Линейный индикатор начнет заполняться дальше.
42. Нажмите кнопку **Сброс**. Линейный регулятор очистится, как это показано на рис. 4.16.

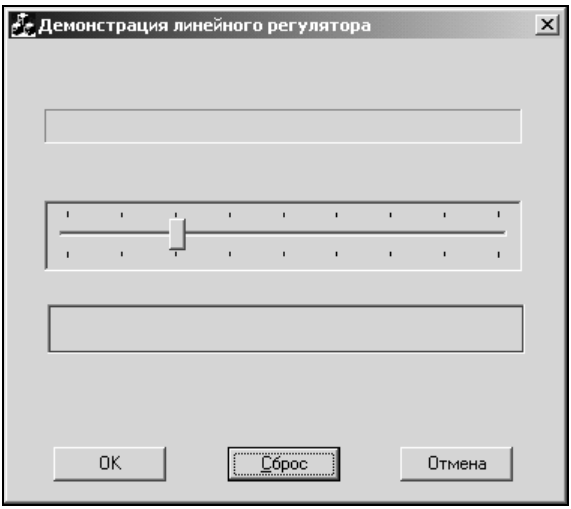


Рис. 4.16. Нажата кнопка **Сброс**

43. Переместите бегунок линейного регулятора. Линейный индикатор продолжит отслеживать крайнее правое положение линейного регулятора.
44. Нажмите на клавиши управления курсором. После того как фокус ввода, пройдя другие элементы управления, установится на инкрементном регуляторе, он начнет управляться этими клавишами. При этом стрелка вправо и стрелка вниз будут передвигать движок линейного регулятора вправо, а стрелка влево и стрелка вверх — влево. Нажатие клавиши <Home> установит линейный регулятор в крайнюю левую позицию, а нажатие клавиши <End> — в крайнюю правую. Клавиша <PgUp> будет перемещать регулятор влево с повышенной скоростью, а клавиша <PgDn> будет перемещать его вправо, также, с повышенной скоростью.

Для того чтобы добиться подобного эффекта при управлении с клавиатуры, в функции `OnInitDialog`, вызываются функции настройки параметров линейного регулятора. Функция `CSliderCtrl::SetRange` имеет три аргумента, первый из которых определяет нижнюю границу диапазона регулирования, второй — определяет верхнюю границу этого диапазона, а третий — необходимость перерисовки элемента управления после установки заданных параметров. Функция `CSliderCtrl::SetTicFreq` устанавливает расстояние, измеряемое в тех же единицах, что и диапазон регулирования, через который будут устанавливаться метки на шкале регулятора. Функция `CSliderCtrl::SetLineSize` устанавливает перемещение бегунка линейного регулятора при нажатии на клавиши управления курсором с изображением стрелок. Функция `CSliderCtrl::SetPageSize` устанавливает перемещение бегунка линейного регулятора при нажатии на клавиши <PgUp> и <PgDn>. Функция `CProgressCtrl::SetRange` действует аналогично подобной функции в классе `CSliderCtrl` и используется для синхронизации диапазонов объектов этих двух классов.

Линейный регулятор представляет собой одну из разновидностей полосы прокрутки. При перемещении своего бегунка линейный регулятор формирует сообщение `WM_HSCROLL`, которое перехватывается функцией — членом класса диалогового окна `OnHScroll`. Первый аргумент этой функции содержит код операции, выполняемой с полосой прокрутки. Полный перечень этих кодов и более полное описание методики работы с полосой прокрутки содержатся в *главе 10*. Второй аргумент этой функции определяет положение бегунка, однако в данном случае он не используется. Дело в том, что этот аргумент используется только в двух случаях: для отслеживания перемещения бегунка мышью и для установления бегунка в определенную позицию. При работе с клавиатурой содержимое данного аргумента игнорируется. Третьим аргументом данной функции является указатель на объект класса `CScrollBar`. Именно этот указатель и будет использоваться для получения текущей позиции бегунка. Прежде всего, мы преобразовываем указатель на объект класса `CScrollBar` в указатель на объект класса `CsliderCtrl`, что, строго говоря, мы не имеем права делать, поскольку родительским классом для `CSliderCtrl` является класс `CWnd`, а не `CScrollBar`, но в Windows возможно и не такое. После получения указателя на объект класса мы используем функцию `CSliderCtrl::GetPos` для получения текущей позиции бегунка. Аналогичная функция вызывается и для класса `CProgressCtrl`. Эта функция возвращает текущее положение линейного индикатора. Если текущее положение линейного регулятора превышает по величине текущее положение линейного индикатора (бегунок линейного регулятора расположен правее текущей позиции линейного индикатора), то вызывается функция `CProgressCtrl::SetPos`, устанавливающая текущую позицию линейного индикатора, а в качестве ее аргумента используется текущая позиция линейного регулятора *(см. главу 10)*.

Функция `OnBnClickedReset` является функцией обработки сообщения о нажатии кнопки **Сброс**. Эта функция сбрасывает линейный индикатор вызовом функции `CProgressCtrl::SetPos` с нулевым значением.

Создание пользовательского линейного индикатора

Вы, наверно, уже заметили, что в данном приложении существует элемент управления, который до сих пор не принимал никакого участия в его работе. Это элемент управления **Picture Control** (Изображение). Дело в том, что стандартный линейный индикатор является достаточно грубым индикатором. Как вы уже успели убедиться, его точность составляет плюс-минус "кирпич", в прямом смысле этого выражения.

Если данный индикатор используется для отображения прохождения процесса загрузки файла с диска, то нет ничего страшного в том, что индикатор покажет начало процесса только после того, как значительная часть его будет уже завершена. Однако представьте себе, что весь процесс занимает несколько часов, и только через десять минут после его начала вы получаете возможность убедиться, что он действительно начался. Или, что еще хуже, вы испытываете новый

алгоритм и не знаете, сколько времени займет его реализация. Знаете только, что достаточно много, и ждете первых признаков его начала.

В этом случае вам хотелось бы получить подтверждение о начале работы, выражающееся в начале заполнения линейного индикатора, как можно раньше. При использовании соответствующих стилей в функции `CProgressCtrl::Create` может быть создан линейный индикатор, не имеющий разбиения своей полосы на блоки, но вызов этой функции выходит за рамки стандартных методов работы с данным элементом управления. То есть, при ее использовании придется полностью взять на себя обработку сообщений от данного элемента управления. Ниже будет представлен способ самостоятельного формирования линейного индикатора с использованием элемента управления **Picture Control** (Изображение). Этот простой пример поможет понять основные принципы работы с данным элементом управления.

Чтобы самостоятельно создать линейный индикатор, внесите в созданный ранее проект следующие изменения:

1. Откройте окно редактирования файла ProgressDlg.cpp.
2. Добавьте перед описанием класса `CAboutDlg` после макроса `END_MESSAGE_MAP()` строку

   ```
   CRect  P_Rect;
   ```

3. Внесите изменения в соответствующие фрагменты файла реализации ProgressDlg.cpp в соответствии с листингом 4.3.

Листинг 4.3. Файлы, реализующие пользовательский линейный индикатор

```
// Если диалоговое окно содержит кнопку Свернуть, расположенные ниже
// операторы выведут значок диалогового окна при минимизации.
// В приложениях, использующих концепцию Документ/Представление эта
// операция производится автоматически.

void CProgressDlg::OnPaint()

{
  if (IsIconic())
  {
    CPaintDC dc(this);  // Контекст устройства для рисования
    SendMessage(WM_ICONERASEBKGND, reinter-
pret_cast<WPARAM>(dc.GetSafeHdc()), 0);

    // Центрирование значка в рабочей области
    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;

    // Вывод значка
    dc.DrawIcon(x, y, m_hIcon);
  }
  else
  {
    CDialog::OnPaint();
  }

  CClientDC  clientDC(this);  // Контекст устройства для рисования
  CBrush    brBack( RGB(255, 255, 255) );  // Фоновая кисть

  //  Получение координат области вывода информации
  ::GetWindowRect( ::GetDlgItem( m_hWnd, IDC_PICTURE), &P_Rect );
  ScreenToClient( &P_Rect );

  // Исключение рамки
  P_Rect.left  += 2;
  P_Rect.right  -= 2;
  P_Rect.top  += 2;
  P_Rect.bottom  -= 2;

  // Заполнение цветом фона
  clientDC.FillRect( P_Rect, &brBack );
}

// Данная функция вызывается системой для получения курсора,
// используемого при перетаскивании минимизированного окна.
HCURSOR CProgressDlg::OnQueryDragIcon()
{
  return static_cast<HCURSOR>(m_hIcon);
}

// Обрабатывает нажатие кнопки Сброс
void CProgressDlg::OnBnClickedReset(void)
{
  m_Progress.SetPos( 0 );     // Сброс линейного индикатора
```

```
  CClientDC  clientDC(this);  // Контекст устройства для рисования
  CRgn       pRgn;            // Область вывода
  CBrush     brBack( RGB(255, 255, 255) );  // Фоновая кисть

  // Заполнение цветом фона

  pRgn.CreateRectRgnIndirect( P_Rect );
  clientDC.SelectClipRgn( &pRgn );
  clientDC.FillRect( P_Rect, &brBack );
}

// Обработка сообщений линейного регулятора
void CProgressDlg::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar*
pScrollBar)
{
  //  Преобразование типа аргумента
  CSliderCtrl*  slider = (CSliderCtrl*) pScrollBar;

  if( m_Progress.GetPos() < slider-> GetPos() )  // Проверка
                                        // текущего положения
  {
    CClientDC  clientDC(this);  // Контекст устройства для рисования
    CRgn       pRgn;            // Область вывода
    CBrush     brDone( RGB(255, 0, 0) );  // Кисть линейного индикатора
    CRect      D_Rect = P_Rect;

    // Задание области отсечки

    pRgn.CreateRectRgnIndirect( P_Rect );
    clientDC.SelectClipRgn( &pRgn );

    // Вывод пользовательского линейного индикатора
    D_Rect.right = P_Rect.left + (int)(P_Rect.Width()*slider->
    GetPos())/slider->GetRangeMax();
    clientDC.FillRect( D_Rect, &brDone );

    // Перемещение стандартного линейного индикатора
    m_Progress.SetPos( slider-> GetPos() );
  }

  CDialog::OnHScroll(nSBCode, nPos, pScrollBar);
}
```

4. Нажмите клавишу <F5> и запустите приложение на исполнение. Появится диалоговое окно, изображенное на рис. 4.17.

Примечание

В данном окне пользовательский линейный индикатор имеет белый фон.

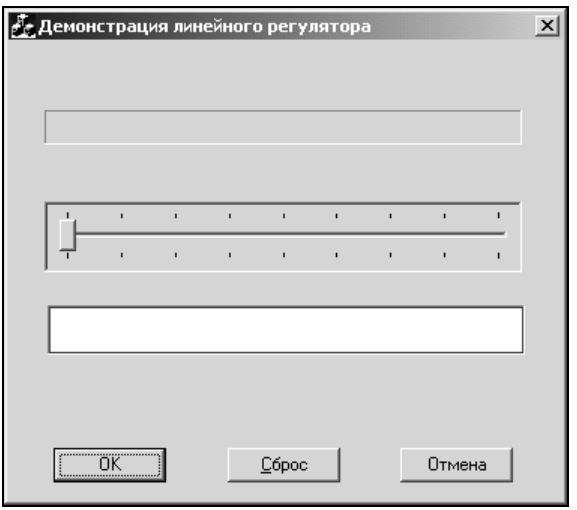


Рис. 4.17. Диалоговое окно с пользовательским линейным индикатором

5. Переместите бегунок линейного регулятора вправо. Диалоговое окно примет вид, изображенный на рис. 4.18.

В приложение были введены элементы интерфейса графических устройств Windows. Подробное описание интерфейса приведено в *главе 6*. Здесь же основное внимание будет сосредоточено на элементе управления **Picture Control** (Изображение) и способам работы с ним.

Как следует из текста программы, для работы с пользовательским линейным индикатором была введена всего одна глобальная переменная. Ее можно было бы оформить переменной — членом класса, но это потребовало бы внесения изменений также и в файл заголовка. Объявление глобальными переменными или переменными — членами класса ресурсов кистей недопустимо, поскольку все ресурсы должны создаваться непосредственно перед их использованием и освобождаться сразу, как только в них отпала необходимость.

Первое изменение было внесено в функцию `OnPaint`. Приложение вызывает эту функцию всякий раз, как только операционная система Windows или само приложение выдает заявку на перерисовку фрагмента окна приложения. Эта функция является обработчиком сообщения `WM_PAINT`. Если окно не минимизирова-

но, то данная функция вызывает функцию — член родительского класса `CDialog::OnPaint`, после чего создаются контекст устройства и кисть для рисования фона пользовательского элемента управления. Затем следует вызов глобальной функции `GetWindowRect`, позволяющей получить экранные координаты окна элемента управления. Для получения указателя на объект окна элемента управления используется глобальная функция `GetDlgItem`, первым аргументом которой является дескриптор диалогового окна, являющийся членом класса `CWnd`, а вторым аргументом — идентификатор ресурса элемента управления.

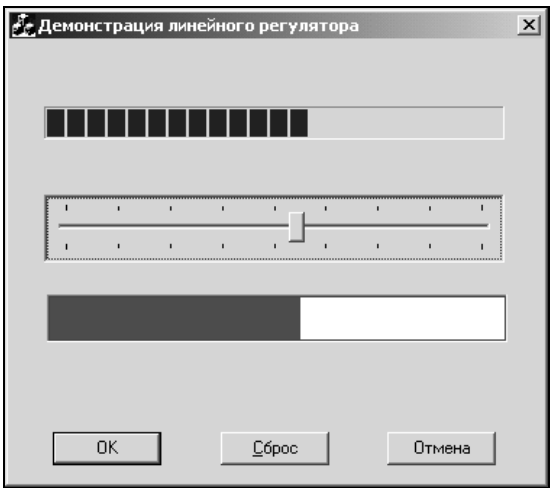


Рис. 4.18. Диалоговое окно с пользовательским линейным индикатором в работе

Поскольку в результате вызова функции `GetWindowRect` были получены экранные координаты окна, то следующим оператором является вызов функции `CWnd::ScreenToClient`, преобразующей экранные координаты в координаты рабочей области окна. В качестве аргумента этой функции могут выступать указатели на объекты структур `POINT` и `RECT`. Причем входная и выходная величины располагаются в том же самом объекте.

Полученные координаты относятся ко всему элементу управления и соответствуют внешним его границам, но при его создании в окне **Properties** (Свойства) был установлен флажок **Sunken** (Углубленный). Это привело к тому, что данный элемент управления имеет рамку, имитирующую его заглубление в поверхность диалогового окна. Эта рамка имеет ширину в два элемента изображения с каждой стороны рамки. Поэтому для корректного вывода информации необходимо внести изменения в координаты окна, чтобы они соответствовали его рабочей области.

Последним оператором функции `OnPaint` является вызов функции `CDC::FillRect`, осуществляющей заливку области, ограниченной прямоугольни-

ком, заданным в ее первом аргументе указателем на объект структуры `RECT`, цветом, определяемым в ее втором аргументе, указателем на класс `CBrush`. Эта функция заполняет всю рабочую область элемента управления цветом фона. В качестве цвета фона в конструкторе соответствующей кисти установлен белый цвет (яркости всех основных цветов присваивается максимальное значение).

В функции `OnBnClickedReset`, кроме всего прочего, производится заливка цветом фона всей рабочей области окна пользовательского элемента управления. Эта операция реализуется тем же набором операторов, что и в функции `OnPaint`, но в эту последовательность добавлены дополнительные операторы. Прежде всего, в данной функции создается объект класса `CRgn`. Данный класс служит для задания эллиптической или многоугольной области для вывода изображения. Функция `CRgn::CreateRectRgnIndirect` позволяет создать прямоугольную область вывода на экран и сохранить ее в данном объекте класса `CRgn`. Указатель на созданный объект используется в качестве аргумента функции `CDC::SelectClipRgn`, задающей способ добавления изображения в указанную область и указывающий, что любая часть выводимого изображения, выходящая за пределы данной области, не будет отображаться. Если в функции указан только один аргумент, то изображение просто копируется в эту область.

В функции `OnHScroll` создается новая кисть для отображения полосы линейного индикатора и создаются объект класса `CRect` для отображения этой полосы, которому первоначально присваивается значение объекта всей рабочей области элемента управления. После этого производится вычисление правой границы этой полосы и она выводится на экран.

Поскольку в данном случае, как и в большинстве реальных случаев, размер полосы линейного индикатора может только увеличиваться, то нет необходимости при каждом изменении ее размера заново перерисовывать участок, которому соответствует цвет фона. Если такая необходимость возникнет, то, во избежание мерцания полосы при ее перерисовке, следует ограничиться изменением цвета только на том участке, где он действительно меняется.

Поскольку в функции `OnDraw` производится только закрашивание рабочей области элемента управления цветом фона, то при восстановлении окна из пиктограммы пользовательский линейный индикатор останется незаполненным до следующего вызова функции `OnHScroll`. В реальных задачах необходимость полного восстановления пользовательского линейного индикатора в функции `OnPaint` зависит от частоты вызова функции его обновления. Если это обновление происходит через доли секунды, то нет никакой необходимости в полном восстановлении пользовательского линейного индикатора в функции `OnPaint`.

Обработка даты и времени

Для обработки информации о дате и времени в Windows предусмотрены специальные элементы управления. Для демонстрации принципов работы с ними создано приложение DateTime, текст которого расположен в одноименной папке

на дискете, прилагаемой к данной книге. Чтобы самостоятельно создать это демонстрационное приложение:

1. Создайте заготовку диалогового приложения по методике, описанной в *главе 1*, и назовите его DateTime.
2. В раскрытой папке **Dialog** (Диалог) в окне **Resource View** (Просмотр ресурсов) выделите идентификатор ресурса `IDD_DATETIME_DIALOG` и нажмите клавишу <F4>. Раскроется окно **Properties** (Свойства).
3. Щелкните левой кнопкой мыши на строке **Language** (Язык). В соответствующем текстовом поле появится значок раскрывающегося списка.
4. Раскройте этот список и выделите в нем строку **Russian** (Русский).
5. Закройте окно **Output** (Окно вывода) и растяните заготовку диалогового окна до размера 200×300.
6. Оставив выделенным заготовку диалогового окна, введите в текстовое поле **Caption** (Заголовок) окна **Properties** (Свойства) заголовок "Календарь".
7. Щелкните левой кнопкой мыши на статическом тексте в заготовке диалогового окна и введите в текстовое поле **Caption** (Заголовок) окна **Properties** (Свойства) заголовок "Дата", а в текстовое поле раскрывающегося списка **ID** (Идентификатор ресурса) того же окна введите идентификатор ресурса `IDC_CAPTION`.
8. Перетащите статический текст в левый верхний угол заготовки диалогового окна и измените размер его поля в соответствии с размерами содержащегося в нем текста, оставив при этом небольшой запас.
9. Перетащите кнопки **OK** и **Cancel** (Отмена) в нижнюю часть диалогового окна.
10. Выделите кнопку **Cancel** (Отмена) и введите в текстовое поле **Caption** (Заголовок) окна **Properties** (Свойства) заголовок "Отмена".
11. В окне **Toolbox** (Инструментарий) выберите элемент управления **Date Time Picker** (Выбор даты и времени) и поместите его под статическим текстом. Окно редактирования ресурса примет вид, изображенный на рис. 4.19.
12. В текстовом поле раскрывающегося списка **ID** (Идентификатор ресурса) окна **Properties** (Свойства) измените идентификатор ресурса на `IDC_DATETIMEPICKER`.
13. В окне **Toolbox** (Инструментарий) выберите элемент управления **Static Text** (Статический текст) и поместите его в правую верхнюю часть заготовки диалогового окна.
14. В текстовое поле **Caption** (Заголовок) окна **Properties** (Свойства) введите заголовок "Календарь".
15. В окне **Toolbox** (Инструментарий) выберите элемент управления **Month Calendar Control** (Календарь на месяц) и поместите его в заготовку диалогового окна под только что введенный статический текст.

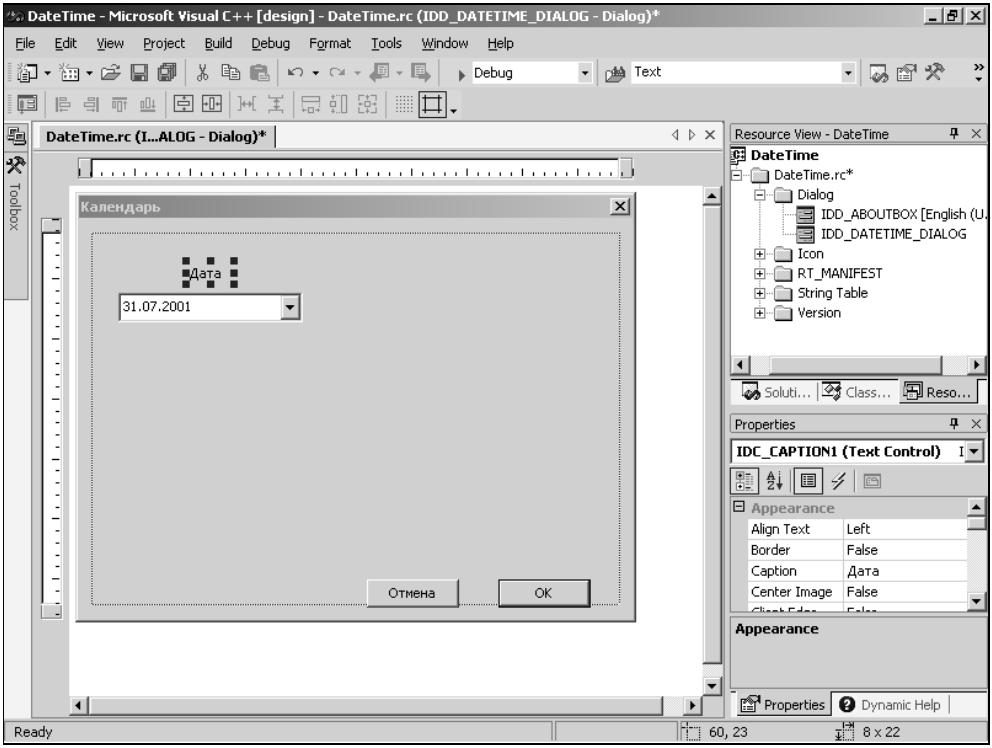


Рис. 4.19. Окно редактирования ресурса с элементом выбора даты и времени

16. В окне **Properties** (Свойства) измените его идентификатор ресурса на `IDC_MONTHCALENDAR` и выделите в раскрывающемся списке **ClientEdge** (Рамка) строку **True** (Истина). Окно редактирования ресурса диалогового окна примет вид, изображенный на рис. 4.20.
17. В окне **Toolbox** (Инструментарий) выберите элемент управления **Button** (Кнопка) и поместите его в заготовку диалогового окна слева от кнопки **Отмена**.
18. Введите в текстовое поле раскрывающегося списка **ID** (Идентификатор ресурса) окна **Properties** (Свойства) идентификатор ресурса `IDC_DATETIME`, а в текстовое поле **Caption** (Заголовок) того же окна введите заголовок "Время".
19. Повторите п.п. 17 и 18, поместив в левый нижний угол заготовки диалогового окна кнопку с идентификатором ресурса `IDC_SELECT` и заголовком "Сброс". Заготовка диалогового окна примет вид, изображенный на рис. 4.21.
20. В заготовке диалогового окна выделите элемент управления **Date Time Picker** (Выбор даты и времени) и нажмите кнопку **Control Events** (События элементов управления) в окне **Properties** (Свойства). Раскроется список событий, генерируемых данным элементом управления.

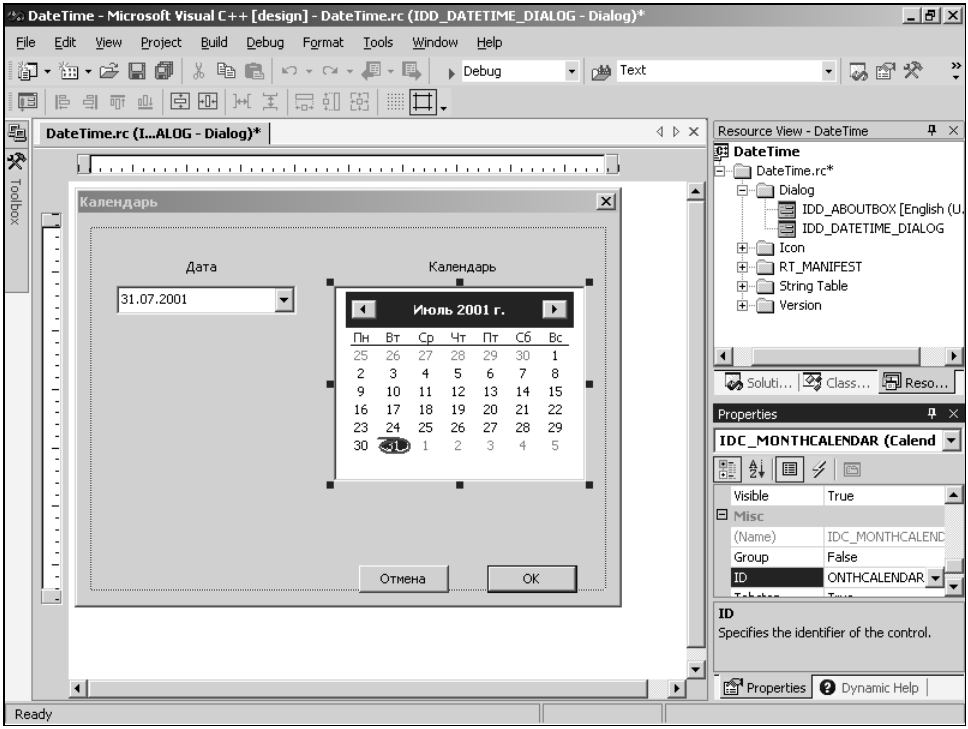


Рис. 4.20. Окно редактирования ресурса с окном списка и текстовым полем

21. Выделите извещение `DTN_DATETIMECHANGE`. В связанном с ним текстовом поле появится значок раскрывающегося списка.
22. Раскройте этот список и выделите в нем единственную строку. В приложение будет добавлена функция обработки данного сообщения.
23. Повторите п.п. 20—22 для элемента управления **Month Calendar Control** (Календарь на месяц), создав при этом функцию обработки сообщения `MCN_SELECT`.
24. Повторите п.п. 20—22 для кнопки **Время**, создав при этом функцию обработки сообщения `BN_CLICKED`.
25. Повторите п.п. 20—22 для кнопки **Сброс**, создав при этом функцию обработки сообщения `BN_CLICKED`.
26. Откройте окно **Class View** (Просмотр классов), а в нем раскройте папку **DateTime**.
27. Щелкните правой кнопкой мыши на папке `CDateTimeDlg` и выберите в появившемся контекстном меню команду **Add | Add Variable** (Добавить | Добавить переменную). Появится диалоговое окно **Add Member Variable Wizard** (Мастер добавления переменной в класс).

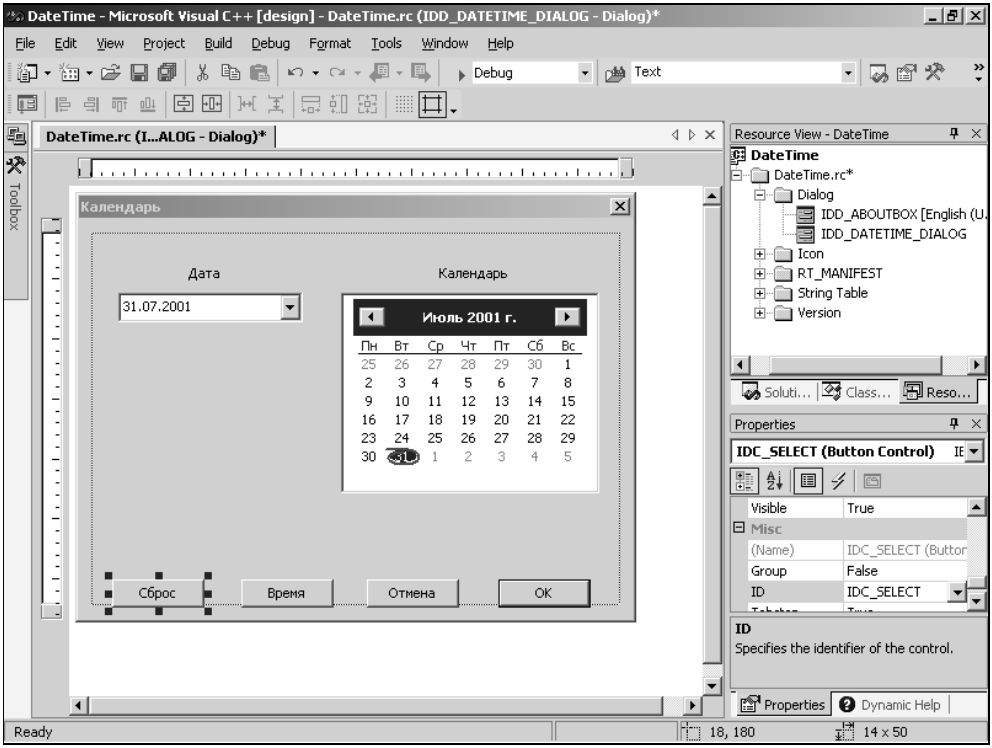


Рис. 4.21. Окончательный вид заготовки диалогового окна

28. В текстовое поле раскрывающегося списка **Variable type** (Тип переменной) введите тип переменной `bool`, в текстовое поле **Variable name** (Имя переменной) введите идентификатор переменой `m_bTime` и нажмите кнопку **Finish** (Готово). Эта переменная будет добавлена в класс.
29. Повторите п.п. 27 и 28 для переменной `m_bSelect`.
30. Повторите п.п. 27 и 28 для переменной `m_Time`, имеющей тип `CTime`.
31. Щелкните правой кнопкой мыши на папке `CDateTimeDlg` и выберите в появившемся контекстном меню команду **Add | Add Variable** (Добавить | Добавить переменную). Появится диалоговое окно **Add Member Variable Wizard** (Мастер добавления переменной в класс).
32. Установите флажок **Control variable** (Связь с элементом управления), раскройте ставший после этого доступным список **Control ID** (Идентификатор элемента управления) и выделите в нем идентификатор ресурса `IDC_DATETIMEPICKER`.
33. Не изменяя установок других элементов управления диалогового окна, введите в текстовое поле **Variable name** (Имя переменной) идентификатор пе-

ременой `m_DateTime` и нажмите кнопку **Finish** (Готово). Эта переменная будет добавлена в класс.

34. Повторите п.п. 31—33 для сопоставления идентификатору ресурса `IDC_MONTHCALENDAR` переменной `m_Month`.
35. Откройте окно редактирования файла DateTimeDlg.cpp и измените текст функций обработки сообщений в соответствии с листингом 4.4.

Листинг 4.4. Функции обработки сообщений

```
// Обработка сообщений элемента управления Date Time Picker


void CDateTimeDlg::OnDtnDatetimechangeDatetimepicker(NMHDR *pNMHDR, LRESULT
*pResult)
{
  LPNMDATETIMECHANGE pDTChange = reinter-
pret_cast<LPNMDATETIMECHANGE>(pNMHDR);


  m_DateTime.GetTime( m_Time );  // Считывание информации
                                 // из элемента управления
  if( !m_bTime )  // Введена дата
    m_Month.SetCurSel( m_Time );
  *pResult = 0;
}

// Обработка сообщений от календаря
void CDateTimeDlg::OnMcnSelectMonthcalendar(NMHDR *pNMHDR, LRESULT *pResult)
{
  LPNMSELCHANGE pSelChange = reinterpret_cast<LPNMSELCHANGE>(pNMHDR);
  m_Month.GetCurSel( m_Time );      // Считывание информации
                                    // из элемента управления
  if( !m_bTime )                    // Введена дата
    m_DateTime.SetTime( &m_Time );  // Внесение изменений в календарь
  *pResult = 0;
}

// Обработка сообщения о нажатии кнопки Время
void CDateTimeDlg::OnBnClickedDatetime()
{
  if( m_bTime )  // Проверка состояния кнопки
```

```
  {
    // Вывод даты
    SetDlgItemText( IDC_DATETIME, "Время" );
    SetDlgItemText( IDC_CAPTION, "Дата" );
    m_DateTime.SetFormat( _T("dd-MM-yyyy") );
    m_bTime = false;
  }
  else
  {
    // Вывод времени
    SetDlgItemText( IDC_DATETIME, "Дата" );
    SetDlgItemText( IDC_CAPTION, "Время" );
    m_DateTime.SetFormat( _T("H-mm-ss") );
    m_bTime  = true;
  }
}

// Обработка сообщения о нажатии кнопки Сброс
void CDateTimeDlg::OnBnClickedSelect()
{
  if( m_bSelect )  // Проверка состояния кнопки
  {
    // Отмена выделения текущей даты
    SetDlgItemText( IDC_SELECT, "Сброс" );
    m_Month.ModifyStyle(MCS_NOTODAYCIRCLE,MCS_NOTODAY);
    m_bSelect  = false;
  }
  else
  {
    // Выделение текущей даты
    SetDlgItemText( IDC_SELECT, "Выделить" );
    m_Month.ModifyStyle(MCS_NOTODAY,MCS_NOTODAYCIRCLE);
    m_bSelect  = true;
  }
}
```

36. Нажмите клавишу <F5> и запустите приложение на исполнение. Появится диалоговое окно, изображенное на рис. 4.22.

37. Раскройте элемент управления **Дата**. Появится второй месячный календарь, как показано на рис. 4.23.

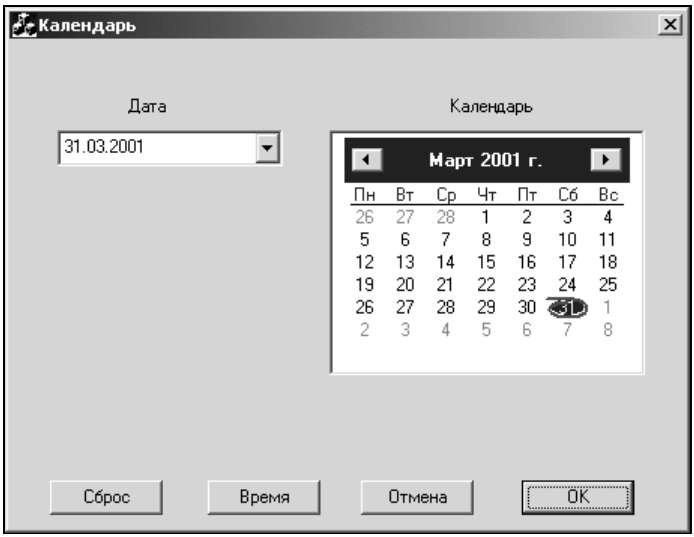


Рис. 4.22. Исходный вид диалогового окна

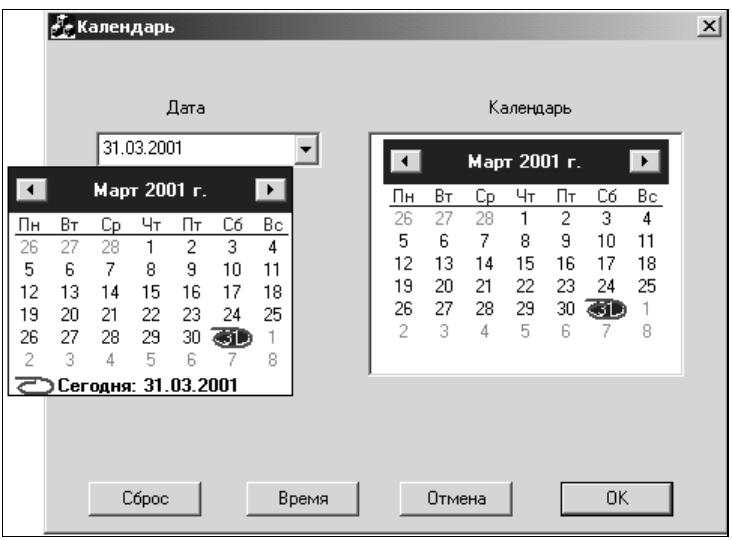


Рис. 4.23. Раскрытый календарь

38. Выделите в нем новую дату. Она будет выведена в текстовое поле этого элемента управления.
39. Нажмите кнопку **Сброс**. В окне календаря исчезнет выделение текущей даты красным обводом, и она появится в нижней его части в текстовом виде. Кроме того, текст нажатой кнопки изменится, как это показано на рис. 4.24.

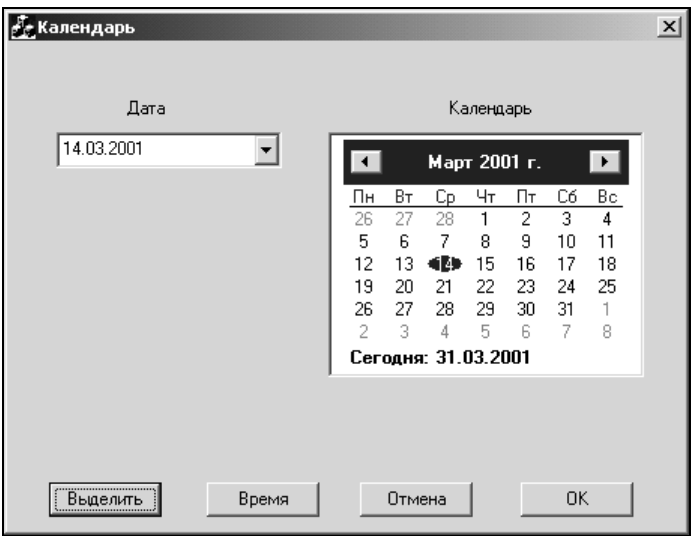


Рис. 4.24. Сброс выделения текущей даты

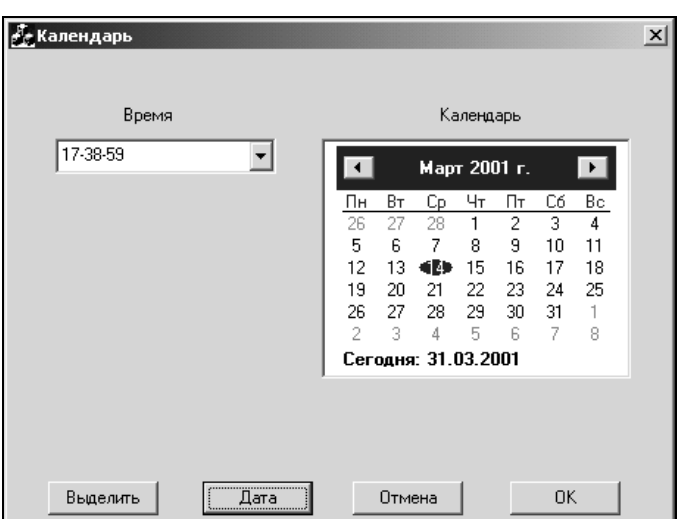


Рис. 4.25. Отображение времени

40. Выделите в календаре другую дату. Дата изменится в обоих элементах управления.
41. Нажмите кнопку **Время**. В текстовое поле будет выведено текущее время, а его заголовок и надпись в нажатой кнопке изменятся, как это показано на рис. 4.25.
42. Нажмите кнопки **Выделить** и **Дата**. Окно вернется к своему первоначальному виду, но в нем будет выделена дата, выбранная в п. 40.
43. Нажмите кнопку **OK**. Диалоговое окно **Календарь** закроется.

Как видно из описания работы данного диалогового окна, в нем реализована синхронизация информации в элементах управления, изменение их стиля и изменение текста в кнопках.

Функция `CDateTimeDlg::OnDatetimechangeDatetimepicker` является функцией обработки сообщения о внесении пользователем изменений в текстовое поле элемента управления **Date Time Picker** (Выбор даты и времени). Она вызывается при выходе пользователя из отредактированного поля данного элемента управления (например, после перехода из отредактированного поля месяца в поле даты) и при закрытии раскрывающегося календаря. Для считывания информации из элемента управления используется функция `CDateTimeCtrl::GetTime`. Информация считывается в объект класса `CTime`. Этот объект класса оформлен как член класса `CDateTimeDlg`, хотя его можно было бы оформить как локальную переменную соответствующих функций обработки сообщений. Если в соответствующем элементе управления выводится информация о дате, то эта информация с использованием функции `CMonthCalCtr::SetCurSel` передается в элемент управления календаря.

Функция `CDateTimeDlg::OnSelectMonthcalendar` является функцией обработки сообщения об изменении выделения в элементе управления календаря на месяц. Для считывания информации из элемента управления используется функция `CMonthCalCtr::GetCurSel`. Полученная информация передается в другой элемент управления только в том случае, если в нем выводится информация о дате. Для этого используется функция `CDateTimeCtrl::SetTime`.

Функция `CDateTimeDlg::OnClickedSelect` является функцией обработки сообщения о нажатии пользователем кнопки **Сброс**. В ней производится анализ текущего состояния этой кнопки и, в зависимости от него, производятся изменения в элементе управления календаря и изменение текста в этой кнопке. Для изменения текста в кнопке используется функция `CWnd::SetDlgItemText`, первым аргументом которой является идентификатор ресурса элемента управления диалогового окна, в котором нужно поменять заголовок, а вторым аргументом — текстовая строка, содержащая данный заголовок. Для изменения стиля календаря используется функция базового класса `CWnd::ModifyStyle`. В первом аргументе данной функции указываются стили, которые следует удалить из данного окна, во втором аргументе — стили, которые следует добавить в это

окно, а в третьем необязательном аргументе указываются флаги, передаваемые функции `SetWindowPos`. Третий аргумент не используется в том случае, если нет необходимости вызывать функцию `SetWindowPos`. После внесения изменений в элементы управления диалогового окна производится модификация значения флага `m_bSelect`, отражающего текущее состояние этой кнопки.

Аналогичные процедуры производятся и в функции `CDateTimeDlg::OnClickedDatetime`, обрабатывающей сообщение о нажатии пользователем кнопки **Время**. Эта кнопка используется для переключения режима отображения даты и времени. Поскольку в данном случае изменяется не только текст в нажимаемой кнопке, но и статический текст над соответствующим элементом управления, функция `CWnd::SetDlgItemText` вызывается дважды. Для задания формата вывода информации в элементе управления используется функция `CDateTimeCtrl::SetFormat`.

Глава 5

Сообщения и команды

В предыдущих главах вам часто встречалось выражение "функция обработки сообщения". Что же понимается под сообщением и почему его нужно обрабатывать. Сообщения являются основной отличительной особенностью операционной системы Windows от операционной системы DOS.

Для того чтобы иметь возможность работать с каким-либо устройством, например с клавиатурой или мышью, программам DOS приходилось постоянно отслеживать состояние этих устройств и ожидать их реакции на посланные им сообщения. В Windows картина кардинально меняется. Данная система управляется сообщениями и уже не программа ожидает реакции от устройства, а сообщение о реакции устройства запускает ту или иную программу. Та часть программы, которая запускается в ответ на конкретное сообщение, называется функцией его обработки.

Бóльшую часть работы по передаче сообщений и вызову соответствующих функций обработки берут на себя внутренние процедуры Windows и функции библиотеки MFC. Однако, поскольку в большинстве случаев посылка сообщения является единственной возможностью обеспечить интерфейс между отдельными частями программы пользователя, прежде чем приступить к детальному рассмотрению различных программ, необходимо глубже познакомиться с механизмом работы с сообщениями.

Обработка сообщений

Для идентификации сообщений операционная система использует целые числа. Однако для удобства пользователей вместо этих чисел используются символьные идентификаторы. Для установления соответствия между идентификатором и соответствующей ему числовой величиной используется огромное количество директив `#define`. Незначительное количество директив, относящихся к сообщениям и ресурсам пользователя, расположены в файле заголовка Resource.h, имеющемся в каждом приложении. Вы можете открыть этот файл и посмотреть директивы. Оконные сообщения в Windows имеют префикс `WM_`, что означает сообщение Windows (Windows Message).

В любом сообщении содержится информация о том, какому окну оно предназначено. Кроме того, любое сообщение может иметь до двух аргументов, в которых могут передаваться различные данные (в том числе и указатели на большие структуры данных).

Обработка различных сообщений выполняется разными компонентами операционной системы, некоторые из которых включают в себя собственные функции обработки данного сообщения. Например, при перемещении указателя мыши необходимо не только сообщить об этом системе и передать ей его новые координаты, но и нарисовать указатель на новом месте. Поэтому многие функции обработки сообщений включают в себя вызов соответствующего метода базового класса.

Библиотека MFC берет на себя обработку большинства сообщений низкого уровня, таких как `WM_KEYDOWN`, `WM_MOUSEMOVE` и других, и позволяет пользователю получать более информативные сообщения о том, что в данное текстовое поле введен некоторый текст или что нажата кнопка **OK**. До появления этой библиотеки пользователям приходилось самостоятельно отслеживать большинство сообщений низкого уровня, что негативно сказывалось на объеме программы и ее читабельности.

Каждая программа Windows имеет свой цикл обработки сообщений (Message Loop), который практически всегда находится в функции `WinMain`. Библиотека MFC скрыла эту функцию в своих недрах, и теперь она не появляется в текстах пользовательских приложений.

Обычно цикл обработки сообщений реализуется с использованием оператора `while`, внутри которого вызывается функция `GetMessage`, отыскивающая сообщение в очереди сообщений данного потока и помещающая его в объект соответствующей структуры. Эта функция позволяет отыскивать не только сообщения, направленные другими окнами, но и сообщения от потоков, направленные функцией `PostThreadMessage`. При поиске сообщений производится проверка на нахождение их в определенном диапазоне значений. Данная функция не может отыскивать сообщения, направленные другим окнам или приложениям. Если найдено любое другое сообщение, отличное от `WM_QUIT`, данная функция возвращает ненулевое значение. В противном случае она возвращает ноль. В случае ошибки возвращаемое значение равно −1, поэтому при построении цикла обработки сообщений необходимо проверять выходную величину не только на ненулевое значение, но и на отсутствие ошибки.

Следующей функцией, вызываемой в цикле обработки сообщений, является `TranslateMessage`. Данная функция переводит сообщения виртуальных клавиш в сообщения о введенных символах. Эти сообщения помещаются в очередь сообщений и впоследствии считываются функциями `GetMessage` и `PeekMessage`. Единственным аргументом данной функции является указатель на структуру, содержащую сообщение.

Последней функцией, вызываемой в стандартном цикле обработки сообщений, является функция `DispatchMessage`, передающая сообщение, содержащееся в структуре, на которую указывает ее единственный аргумент процедуре окна.

Функция процедуры окна обычно представляет собой огромный оператор-переключатель (оператор `switch`), в котором перечислены все возможные сообщения и каждому из них сопоставлен вызов соответствующей функции обработки. Естественно, что работать с таким оператором было бы очень сложно, поэтому в библиотеке MFC предусмотрено использование карты сообщений.

Карта сообщений

Использование *карты сообщений* (Message Map) является основным подходом библиотеки MFC к вопросу обработки сообщений. Вместо того чтобы самому писать функцию процедуры окна и указывать в ней все возможные сообщения, пользователь использует стандартный набор макросов библиотеки MFC, позволяющих определить для каждого класса набор обрабатываемых им сообщений и сопоставить им функцию обработки. Кроме того, среда программирования Visual C++ позволяет освободить пользователя даже от необходимости составления карты сообщений: она формируется автоматически.

Каждый класс, производный от класса `CCmdTarget`, должен включать в себя карту сообщений для обработки поступающих ему сообщений. Карта сообщений состоит из двух частей, одна из которых расположена в файле заголовка класса, а вторая — в файле реализации. Первая часть карты сообщений, располагаемая в файле заголовка, должна заканчиваться макросом `DECLARE_MESSAGE_MAP`, не имеющим аргументов. Вторая часть карты сообщений, располагаемая в файле реализации, должна начинаться макросом `BEGIN_MESSAGE_MAP`, первым аргументом которого является имя класса, сообщения которого будут обрабатываться данной картой, а вторым аргументом — имя базового класса. Вторая часть карты сообщений должна заканчиваться макросом `END_MESSAGE_MAP`, не имеющим аргументов.

Описанная выше строгая структура карты сообщений была характерна для Visual C++ версий 5.0 и 6.0. В версии 7.0 указанные выше правила соблюдаются не очень строго. По неизвестным причинам в Visual C++ 7.0 в карту сообщений заголовка класса помещаются только функции обработки сообщений, созданные мастером MFC Application Wizard. Все функции обработки сообщений, созданные пользователем с использованием окна **Properties** (Свойства), помещаются в заголовке класса после карты сообщений и являются открытыми (`public`).

В листинге 5.1 приведен файл заголовка класса диалогового окна тестового приложения, демонстрирующего реализацию простейших элементов управления, описанного в *главе 4*.

Листинг 5.1. Файл заголовка с картой сообщений

```
// Файл заголовка DialogDlg.h
//

#pragma once

// Класс диалогового окна CDialogDlg
class CDialogDlg : public CDialog
{
// Конструкторы
```

```
public:
  CDialogDlg(CWnd* pParent = NULL);  // Стандартный конструктор
// Данные диалогового окна
  enum { IDD = IDD_DIALOG_DIALOG };

  protected:
  virtual void DoDataExchange(CDataExchange* pDX);  // Поддержка DDX/DDV

// Реализация
protected:
  HICON m_hIcon;

  // Функции карты сообщений
  virtual BOOL OnInitDialog();
  afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
  afx_msg void OnPaint();
  afx_msg HCURSOR OnQueryDragIcon();
  DECLARE_MESSAGE_MAP()
public:
  afx_msg void OnCbnCloseupCombo();
  afx_msg void OnCbnEditchangeCombo();
  afx_msg void OnDeltaposSpin(NMHDR *pNMHDR, LRESULT *pResult);
  UINT m_Edit;
  UINT m_Buddy;
  CString m_Combo;
  BOOL m_Save;
  int m_Switch;
};
```

В этом файле пользовательские функции обработки сообщений являются открытыми (`public`) и по своему синтаксису и местоположению ничем не отличаются от обычных функций.

Вторая часть карты сообщений расположена в файле реализации данного класса. В листинге 5.2 представлена вторая часть карты сообщений из файла реализации класса, первая часть которой приведена в листинге 5.1.

Листинг 5.2. Карта сообщений, располагаемая в файле реализации класса

```
BEGIN_MESSAGE_MAP(CDialogDlg, CDialog)
  ON_WM_SYSCOMMAND()
  ON_WM_PAINT()
```

```
  ON_WM_QUERYDRAGICON()
  //}}AFX_MSG_MAP
  ON_CBN_CLOSEUP(IDC_COMBO, OnCbnCloseupCombo)
  ON_CBN_EDITCHANGE(IDC_COMBO, OnCbnEditchangeCombo)
  ON_NOTIFY(UDN_DELTAPOS, IDC_SPIN, OnDeltaposSpin)
END_MESSAGE_MAP()
```

Поскольку карта сообщений формируется автоматически, пользователю нет особой необходимости разбираться в ее макросах. Поэтому ниже приводится далеко не полный список макросов, используемых в карте сообщений:

- `ON_COMMAND` — определяет функцию, которая будет обрабатывать сообщение `WM_COMMAND`, обычно посылаемое из меню или из панели инструментов. Данный макрос имеет два аргумента, первым из которых является идентификатор команды, а вторым — имя функции, обрабатывающей данное сообщение;
- `ON_COMMAND_RANGE` — работает аналогично макросу `ON_COMMAND`, однако сопоставляет функции обработки не одно сообщение, а целый диапазон сообщений. В связи с этим, данный макрос имеет три аргумента, первый из которых определяет нижнюю границу диапазона идентификаторов команд, второй — верхнюю границу диапазона, а третий — функции обработки команд данного диапазона. Одному сообщению, независимо от того, входит ли оно в заданный диапазон или для него предусмотрен отдельный макрос, должна соответствовать только одна функция его обработки;
- `ON_UPDATE_COMMAND_UI` — определяет функцию, которая будет обрабатывать сообщение об обновлении команды интерфейса пользователя. Данное сообщение будет выдаваться каждый раз, когда приложению понадобится определить состояние элементов пользовательского интерфейса при их отображении на экране. Данный макрос имеет два аргумента, первым из которых является идентификатор сообщения, а вторым — имя функции, обрабатывающей данное сообщение;
- `ON_UPDATE_COMMAND_UI_RANGE` — этот макрос работает аналогично макросу `ON_UPDATE_COMMAND_UI`, однако определяет состояние не одного, а целого диапазона элементов интерфейса пользователя. Так же, как и `ON_COMMAND_RANGE`, данный макрос имеет три аргумента аналогичного назначения;
- `ON_CONTROL` — определяет функцию обработки извещения от пользовательского элемента управления. Извещения от пользовательского элемента управления направляются его родительскому окну. Каждому подобному извещению должен соответствовать один макрос `ON_CONTROL`. Данный макрос имеет три аргумента, первый из которых определяет код извещения элемента управления, второй — идентификатор команды, а третий — функцию обработки сообщения для данной команды;
- `ON_CONTROL_RANGE` — используется для задания функции обработки сообщений целому диапазону идентификаторов команд при едином для всех идентификаторов коде извещения элемента управления, например, `BN_PAINT`. Данный

макрос имеет четыре аргумента, первый и последний из которых совпадают с соответствующими аргументами макроса ON_CONTROL, второй определяет нижнюю границу диапазона идентификаторов, а третий — верхнюю;

- ON_MESSAGE — определяет функцию обработки сообщения пользователя. Обычно сообщения пользователя находятся в диапазоне от WM_USER до 0x7FFF и под ними понимаются любые сообщения, кроме стандартного сообщения Windows WM_MESSAGE. Каждому сообщению пользователя должен соответствовать свой макрос ON_MESSAGE. Данный макрос имеет два аргумента, первым из которых является идентификатор сообщения, а вторым — имя функции, обрабатывающей данное сообщение.

Сообщения в Windows 3.*x*

В Windows 3.*x* элементы управления посылали своим родительским окнам извещения о таких событиях, как щелчки кнопки мыши, изменение содержимого полей и о выделении объектов. С помощью механизма сообщений осуществлялась также фоновая перерисовка содержимого окон. Простейшие извещения посылались как специальные сообщения формата WM_COMMAND, содержащего код извещения (например, BN_PAINT), а также идентификатор элемента управления, передаваемый в аргументе wParam данного сообщения, и дескриптор элемента управления, передаваемый в аргументе lParam сообщения. Поскольку в данном формате сообщения используются оба его аргумента, то в нем не может быть передана никакая дополнительная информация. Поэтому данное сообщение может служить только для извещения о событии. Например, при передаче кода извещения BN_CLICKED нет никакой возможности передать информацию о положении курсора мыши в момент щелчка.

При необходимости передачи дополнительной информации в извещении при работе в Windows 3.*x* приходилось использовать множество специальных сообщений, таких как WM_CTLCOLOR, WM_VSCROLL, WM_HSCROLL, WM_DRAWITEM, WM_MEASUREITEM, WM_COMPAREITEM, WM_DELETEITEM, WM_CHARTOITEM, WM_VKEYTOITEM и др. Эти сообщения могли возвращаться родительским окном тому сообщению, которое их послало.

Сообщения в Win32

Для элементов управления, унаследованных от Windows 3.1, Win32 API в большинстве случаев использует те же специальные сообщения. Однако в Win32 определены также достаточно сложные элементы управления, которых не было в Windows 3.*x*. В большинстве случаев при работе с данными элементами управления возникает необходимость в передаче дополнительных данных вместе с извещением о событии. Вместо того чтобы создавать для каждого нового типа данных, который необходимо передать вместе с извещением о событии, новое сообщение формата WM_*, разработчики Win32 API ввели только одно новое сообщение данного формата, а именно — WM_NOTIFY, позволяющее передавать любое количество дополнительной информации стандартным образом.

Сообщение WM_NOTIFY содержит идентификатор элемента управления, посылающего данное сообщение, в аргументе wParam и указатель на объект некоторой структуры данных в аргументе lParam. Этот объект может являться или объектом структуры NMHDR, или объектом другой структуры, первым элементом которой является объект структуры NMHDR. Поскольку объект структуры NMHDR является первым элементом структуры, то указатель на объект любой структуры, передаваемый в аргументе lParam сообщения, может рассматриваться как указатель на структуру NMHDR.

В большинстве случаев объект структуры NMHDR является только первым элементом большей структуры, при обращении к которой необходимо применять квалифицированную ссылку. К немногим исключениям из данного правила относятся простейшие извещения (имена которых начинаются с NM_) и извещения о вызове справки по элементу управления TTN_SHOW и TTN_POP, использующие указатель на объект структуры NMHDR.

Структура NMHDR содержит дескриптор и идентификатор элемента управления, пославшего данное сообщение, и код извещения (например, TTN_SHOW). Данная структура имеет следующий формат:

```
typedef struct tagNMHDR
{
  HWND hwndFrom;
  UINT idFrom;
  UINT code;
} NMHDR;
```

Для сообщения TTN_SHOW переменная code должна содержать код извещения TTN_SHOW.

Рассмотрим в качестве примера структуры, первым элементом которой является объект структуры NMHDR, структуру, используемую сообщением LVN_KEYDOWN просмотрового окна списка. Структура, на которую ссылается указатель в сообщении LV_KEYDOWN, имеет следующий формат:

```
typedef struct tagLV_KEYDOWN
{
  NMHDR  hdr;
  WORD  wVKey;
  UINT  flags;
} LV_KEYDOWN;
```

Сообщения, посылаемые всеми новыми элементами управления Windows

Имеется ряд сообщений, посылаемых всеми новыми элементами управления Windows. Их отличительной особенностью является то, что они используют ука-

затель непосредственно на объект структуры NMHDR. К подобным сообщениям относятся:

- NM_CLICK — сообщение о том, что пользователь щелкнул левой кнопкой мыши в рабочей области элемента управления;
- NM_DBLCLK — сообщение о том, что пользователь произвел двойной щелчок левой кнопкой мыши в рабочей области элемента управления;
- NM_RCLICK — сообщение о том, что пользователь щелкнул правой кнопкой мыши в рабочей области элемента управления;
- NM_RDBLCLK — сообщение о том, что пользователь произвел двойной щелчок правой кнопкой мыши в рабочей области элемента управления;
- NM_RETURN — сообщение о том, что пользователь нажал клавишу <Return>, когда фокус ввода принадлежал данному элементу управления;
- NM_SETFOCUS — сообщение о том, что данный элемент управления получил фокус ввода;
- NM_KILLFOCUS — сообщение о том, что данный элемент управления потерял фокус ввода;
- NM_OUTOFMEMORY — сообщение о том, что данный элемент управления не может завершить операцию вследствие недостатка оперативной памяти.

Обработка извещений

Обработку извещений осуществляет функция CWnd::OnNotify. По умолчанию она производит просмотр карты сообщений для определения функций обработки поступивших в нее извещений. Обычно эта функция не перегружается. Вместо этого в карту сообщений класса окна добавляются функции обработки новых извещений.

Среда программирования Visual C++ позволяет не только ввести соответствующие макросы ON_NOTIFY для обработки сообщения, но и создать для пользователя заготовки функций обработки этих сообщений. Макрос ON_NOTIFY имеет три аргумента, первым из которых является код извещения, которое следует обработать, например LVN_KEYDOWN, вторым — идентификатор элемента управления, пославшего данное извещение, а третьим — функция обработки данного сообщения.

Функция обработки сообщения должна иметь следующий формат:

```
afx_msg void memberFxn(NMHDR *pNMHDR, LRESULT *pResult)
```

где:

- memberFxn — имя функции обработки сообщения;
- pNMHDR — указатель на структуру извещения, описанную выше;
- pResult — указатель на код результата, устанавливаемый пользователем перед выходом из процедуры.

Например, чтобы указать, что функция OnDeltaposSpin должна обрабатывать сообщение UDN_DELTAPOS, поступающее от объекта класса CSpinButtonCtrl с идентификатором IDC_SPIN, в карту сообщений будет добавлен следующий макрос:

```
ON_NOTIFY(UDN_DELTAPOS, IDC_SPIN, OnDeltaposSpin)
```

а в раздел реализации соответствующего класса будет помещена следующая заготовка функции реализации данного сообщения:

```
void CDialogDlg::OnDeltaposSpin(NMHDR *pNMHDR, LRESULT *pResult)
{
  NM_UPDOWN* pNMUpDown = (NM_UPDOWN*)pNMHDR;
  // TODO: Add your control notification handler code here

  *pResult = 0;
}
```

Обратите внимание на то, что заготовка функции обработки сообщения автоматически обеспечивает соответствие классов применяется переменных. Для доступа к объекту класса NM_UPDOWN используется указатель на этот класс pNMUpDown, получаемый из указателя pNMHDR с использованием непосредственного преобразования типов указателей.

В файле заголовка данная функция должна быть объявлена следующим образом:

```
void memberFxn(UINT id, NMHDR * pNMHDR, LRESULT *pResult)
```

Чтобы данное сообщение обрабатывалось не одним, а несколькими объектами, участвующими в обработке сообщений, используйте вместо макроса ON_NOTIFY (или ON_NOTIFY_RANGE) макрос ON_NOTIFY_EX (или ON_NOTIFY_EX_RANGE). Единственным отличием расширенной версии макроса является то, что макросы с окончанием EX возвращают величину типа BOOL, указывающую, нужно ли продолжать обработку данного сообщения на следующих уровнях его обработки. Если возвращаемая величина имеет значение FALSE, то данное сообщение может быть обработано на следующих уровнях.

Автоматическое создание макросов ON_NOTIFY_EX или ON_NOTIFY_EX_RANGE невозможно, поэтому вся работа по заполнению обеих частей карты сообщений ложится в данном случае на пользователя.

Макросы, помещаемые в карту сообщений, имеют следующий вид:

```
ON_NOTIFY_EX(nCode, id, memberFxn)
ON_NOTIFY_EX_RANGE(wNotifyCode, id, idLast, memberFxn)
```

Назначение их аргументов полностью совпадает с назначением аргументов обычных версий данных макросов. В файле соответствующая функция обработки сообщения должна быть объявлена следующим образом:

```
BOOL memberFxn(UINT id, NMHDR * pNotifyStruct, LRESULT * result);
```

В обоих случаях аргумент id содержит идентификатор элемента управления, пославшего данное сообщение. Данная функция должна возвращать значение

TRUE, если обработка данного сообщения полностью завершена, или FALSE, если предполагается дальнейшая обработка данного сообщения на следующих уровнях обработки сообщений.

Обработка отраженных сообщений

Отражение сообщений является новой особенностью библиотеки MFC 4.0. Необходимость в подобных сообщениях связана с тем, что при работе с элементами управления часто возникает необходимость послать извещение родительскому окну. Например, многие элементы управления посылают своему родительскому окну извещение WM_CTLCOLOR или одну из его разновидностей, чтобы передать ему свой запрос на кисть для заполнения фона рабочей области данного элемента управления.

В Windows и в библиотеках MFC более ранних версий, чем версия 4.0, каждое родительское окно, в котором используется данный элемент управления, должно было включать в себя функцию обработки данного сообщения. Таких классов родительских окон, содержащих практически идентичные по тексту функции обработки, могло набраться очень много. Например, в предыдущем примере каждое диалоговое окно, использующее элементы управления с определенным пользователем цветом фона, должно было содержать функцию обработки запроса на кисть для заполнения фона. Намного проще было бы создать класс элемента управления, который сам мог бы работать с цветом фона рабочей области своего окна.

В библиотеке MFC 4.0 и более поздних ее версиях продолжает использоваться старый механизм обработки извещений, когда обязанность их обработки возложена на класс родительского окна. Но, кроме того, данная библиотека допускает повторное использование программного кода за счет применения механизма, называемого *обработка отраженного сообщения* (message reflection), позволяющего производить обработку извещений в классе элемента управления, в классе родительского окна или одновременно в обоих классах. В нашем примере с цветом фона рабочей области окна элемента управления обработка сообщения WM_CTLCOLOR может быть реализована методами самого класса элемента управления, без внесения каких-либо изменений в класс его родительского окна. Следует, однако, отметить, что, поскольку механизм отражения сообщений реализован средствами библиотеки MFC, а не средствами Windows, класс родительского окна элемента управления, в котором реализован механизм обработки отраженного сообщения, должен быть производным от класса CWnd.

Для достижения аналогичного эффекта в ранних версиях библиотеки MFC использовались виртуальные функции для обработки некоторых сообщений, таких как сообщения для созданных пользователем окон списков (WM_DRAWITEM и другие). Однако механизм обработки отраженного сообщения представляет собой единую и стройную систему. Процедуры, использующие этот механизм, могут быть применены и в программах, написанных с использованием ранних версий библиотеки MFC.

Если в классе родительского окна предусмотрена обработка данного извещения или диапазона извещений, то эти функции обработки сообщений перегружают функции класса элемента управления, и механизм отражения сообщений перестает работать для данного класса родительского окна.

При обработке сообщения или диапазона сообщений `WM_NOTIFY` применяются диаметрально противоположные принципы перегрузки функций обработки отраженного сообщения. В этом случае функция обработки сообщения, расположенная в классе родительского окна, будет вызвана только в том случае, когда в классе элемента управления отсутствует соответствующая функция обработки отраженного сообщения и в карте сообщений данного класса отсутствует макрос `ON_NOTIFY_REFLECT`. При наличии в карте сообщений класса элемента управления макроса `ON_NOTIFY_REFLECT_EX`, обработка соответствующего сообщения может продолжаться, а может и не продолжаться в классе родительского окна. Если функция обработки данного сообщения, расположенная в классе элемента управления, возвращает `TRUE`, то обработка данного сообщения будет продолжена функцией — членом класса родительского окна, если же функция — член класса элемента управления возвращает `FALSE`, то обработка сообщения считается законченной и никаких дальнейших обработок данного сообщения не производится.

После того как элемент управления послал сообщение `WM_NOTIFY`, право первой обработки данного сообщения предоставляется функции — члену класса данного элемента управления. Если посылается любое другое отраженное сообщение, то право первой обработки данного сообщения предоставляется функции — члену класса родительского окна данного элемента управления. Для того чтобы вызвать нужную функцию обработки, класс должен включать в себя соответствующую функцию, а в карте сообщений должен находиться соответствующий макрос.

Макросы карты сообщений, обрабатывающие отраженные сообщения, имеют особый синтаксис: к обычному имени макроса добавляется окончание `_REFLECT`. Например, для обработки сообщения `WM_NOTIFY` в родительском окне в карту сообщений соответствующего класса окна добавляется макрос `ON_NOTIFY`, а для обработки того же отраженного сообщения в карту сообщений соответствующего класса элемента управления добавляется макрос `ON_NOTIFY_REFLECT`. В некоторых случаях данные макросы имеют даже различные аргументы. Однако в большинстве случаев за правильностью составления списка формальных параметров функций и макросов следит среда разработки Visual C++, предоставляющая также заготовки функций обработки сообщений.

Макросы карты сообщений и заготовки функций обработки отраженных сообщений

Для обработки отраженных сообщений, посылаемых классами элементов управления, используются макросы карты сообщений и заготовки функций обработки отраженных сообщений. Для классов элементов управления, использующих

библиотеку MFC, процесс создания этих макросов и заготовок функций обработки сообщений может быть автоматизирован.

Между именами макросов карты сообщений и идентификаторами отраженных сообщений, как правило, существует взаимно однозначное соответствие: для того чтобы преобразовать идентификатор отраженного сообщения в обрабатывающий его макрос, необходимо добавить к нему префикс `ON_` и окончание `_REFLECT`. Например, сообщение `WM_CTLCOLOR` обрабатывает макрос `ON_WM_CTLCOLOR_REFLECT`. Для того чтобы определить, какое сообщение обрабатывает данный макрос, необходимо произвести обратное преобразование.

Из этого правила существуют три исключения:

- сообщению `WM_COMMAND` соответствует макрос `ON_CONTROL_REFLECT`;
- сообщению `WM_NOTIFY` соответствует макрос `ON_NOTIFY_REFLECT`;
- сообщению `ON_UPDATE_COMMAND_UI` соответствует макрос `ON_UPDATE_COMMAND_UI_REFLECT`.

Подробную справку по аргументам и возвращаемой величине функции обработки сообщения можно получить в разделе справки по данной функции или по функции, имя которой получено добавлением префикса `On` к имени данной функции. Например, справка по функции `CtlColor` содержится в описании функции `OnCtlColor`. Некоторые функции обработки отраженных сообщений имеют меньше аргументов, чем соответствующие им функции обработки данного сообщения в классе родительского окна. Для установления соответствия аргументов данных функций необходимо воспользоваться информацией о типах и стандартных именах аргументов, приведенных в документации по этим функциям.

В табл. 5.1 приведены макросы карты сообщений и соответствующие им стандартные имена функций обработки данных сообщений.

Таблица 5.1. *Макросы карты сообщений и соответствующие им функции обработки*

Макрос карты сообщений	Функция обработки сообщения
`ON_CONTROL_REFLECT(wNotifyCode, memberFxn)`	`afx_msg void memberFxn ();`
`ON_NOTIFY_REFLECT(wNotifyCode, memberFxn)`	`afx_msg void memberFxn (NMHDR * pNotifyStruct, LRESULT* result);`
`ON_UPDATE_COMMAND_UI_REFLECT (memberFxn)`	`afx_msg void memberFxn (CCmdUI* pCmdUI);`
`ON_WM_CTLCOLOR_REFLECT()`	`afx_msg HBRUSH CtlColor (CDC* pDC, UINT nCtlColor);`
`ON_WM_DRAWITEM_REFLECT()`	`afx_msg void DrawItem (LPDRAWITEMSTRUCT lpDrawItemStruct);`
`ON_WM_MEASUREITEM_REFLECT()`	`afx_msg void MeasureItem (LPMEASUREITEMSTRUCT lpMeasureItemStruct);`

***Таблица 5.1** (окончание)*

Макрос карты сообщений	Функция обработки сообщения
ON_WM_DELETEITEM_REFLECT()	afx_msg void DeleteItem (LPDELETEITEMSTRUCT lpDeleteItemStruct);
ON_WM_COMPAREITEM_REFLECT()	afx_msg int CompareItem (LPCOMPAREITEMSTRUCT lpCompareItemStruct);
ON_WM_CHARTOITEM_REFLECT()	afx_msg int CharToItem (UINT nKey, UINT nIndex);
ON_WM_VKEYTOITEM_REFLECT()	afx_msg int VKeyToItem (UINT nKey, UINT nIndex);
ON_WM_HSCROLL_REFLECT()	afx_msg void HScroll (UINT nSBCode, UINT nPos);
ON_WM_VSCROLL_REFLECT()	afx_msg void VScroll (UINT nSBCode, UINT nPos);
ON_WM_PARENTNOTIFY_REFLECT()	afx_msg void ParentNotify (UINT message, LPARAM lParam);

Макросы ON_NOTIFY_REFLECT и ON_CONTROL_REFLECT обеспечивают возможность обработки одного и того же сообщения несколькими объектами (класса элемента управления и класса его родительского окна). В табл. 5.2. приведен синтаксис соответствующих функций обработки сообщения.

***Таблица 5.2.** Синтаксис функций обработки сообщений*

Макрос карты сообщений	Функция обработки сообщения
ON_NOTIFY_REFLECT_EX (wNotifyCode, memberFxn)	afx_msg BOOL memberFxn (NMHDR * pNotifyStruct, LRESULT* result);
ON_CONTROL_REFLECT_EX (wNotifyCode, memberFxn)	afx_msg BOOL memberFxn ();

Пример обработки отраженных сообщений

Как следует из вышеизложенного, отраженные сообщения используются в тех случаях, когда разработчики Microsoft специально или непреднамеренно недоработали элемент управления и пользователю приходится самому его дорабатывать. К первой группе относятся достаточно сложные элементы управления, такие как просмотровое окно списка или иерархический список. Элементы этих списков, как правило, представляют собой достаточно сложные структуры, содержащие графическую и текстовую информацию, правила работы с которыми не могут быть однозначно формализованы на этапе разработки этих элементов управления. Их можно рассматривать как заготовки элементов управления, до-

рабатываемых пользователем по своему усмотрению. Вторую группу следует рассматривать как ошибки разработчика и, по идее, таких элементов управления не должно быть.

Единственным примером элемента управления, который Microsoft упорно не хочет дорабатывать, является инкрементный регулятор, который в приложениях Visual C++ работает не так, как во всех остальных приложениях. В *главе 3* уже был рассмотрен способ приведения этого элемента управления к нормальному виду. Продемонстрируем, как эта задача может быть решена с использованием отраженных сообщений. Для этого внесем изменения в приложение Dialog *(см. главу 3)*:

1. Выберите команду меню **File | Open Solution** (Файл | Открыть приложение). Откроется диалоговое окно **Open Solution** (Открыть приложение), изображенное на рис. 5.1.

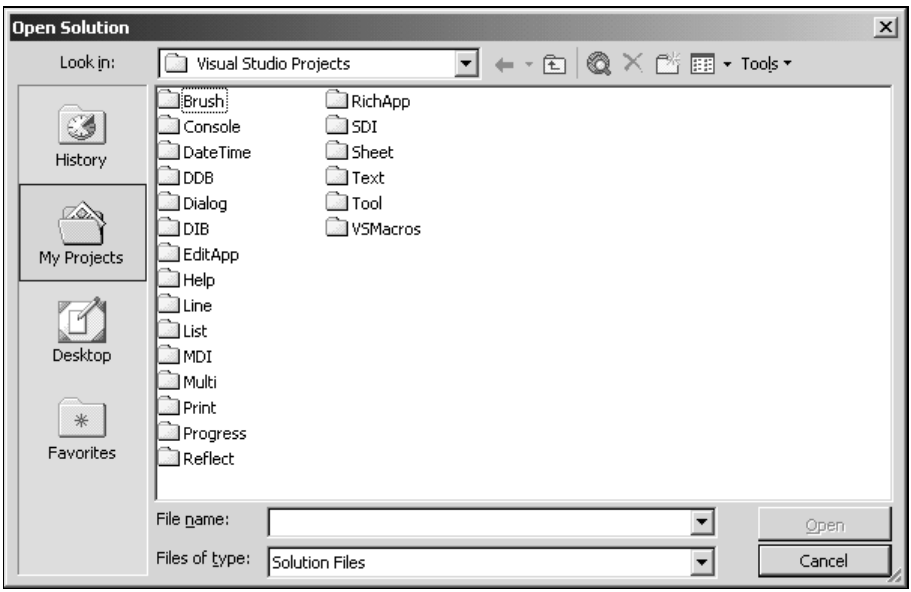


Рис. 5.1. Диалоговое окно **Open Solution**

2. Раскройте папку **Dialog** и дважды щелкните левой кнопкой мыши на значке **Dialog**. В среду программирования Visual C++ загрузится проект Dialog.
3. Откройте окно **Class View** (Просмотр класса), щелкните правой кнопкой мыши на папке **Dialog** (Диалог) и выберите в появившемся контекстном меню команду **Add | Add Class** (Добавить | Добавить класс). Появится диалоговое окно **Add Class - Dialog** (Добавить класс), показанное на рис. 5.2.

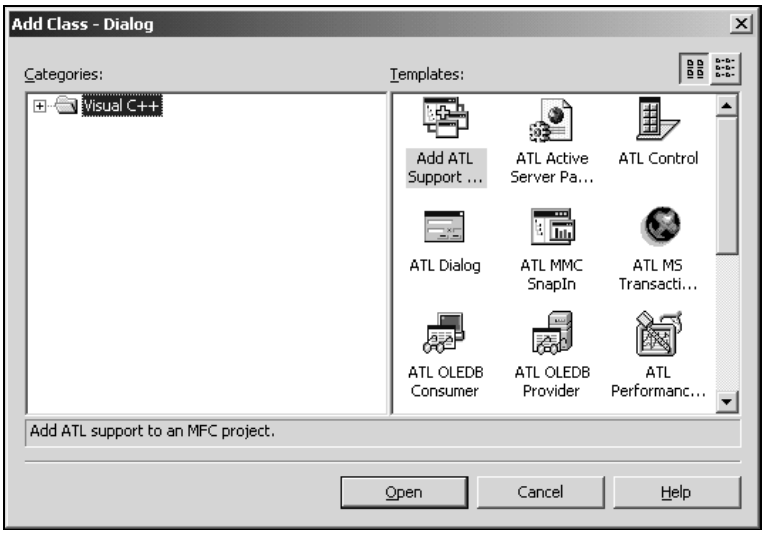


Рис. 5.2. Диалоговое окно **Add Class - Dialog**

4. В окне **Templates** (Шаблоны) выделите значок **MFC Class** (Класс MFC) и нажмите кнопку **Open** (Открыть). Откроется окно **MFC Class Wizard - Dialog** (Мастер создания классов MFC), изображенное на рис. 5.3.

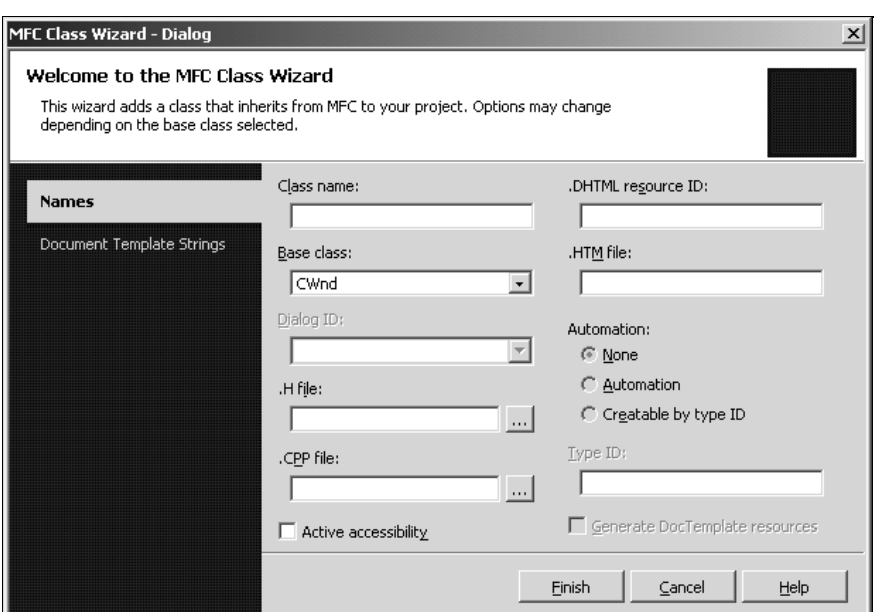


Рис. 5.3. Диалоговое окно **MFC Class Wizard - Dialog**

5. Введите в текстовое поле **Class name** (Имя класса) имя нового класса `MySpinCtrl`, `CSpinButtonCtrl` и нажмите кнопку **Finish** (Готово). Откроется окно редактирования файла заголовка данного класса.
6. В окне **Class View** (Просмотр класа) раскройте папку **Dialog** (Диалог) и щелкните правой кнопкой мыши на папке `MySpinCtrl`.
7. В раскрывшемся контекстном меню выберите команду **Properties** (Свойства). Откроется окно **Properties** (Свойства).
8. В окне **Properties** (Свойства) нажмите кнопку **Messages** (Сообщения). Раскроется список сообщений, обрабатываемых данным классом. Отраженные сообщения в этом списке отмечены знаком равенства (=) и расположены в верхней части данного списка, как это показано на рис. 5.4.

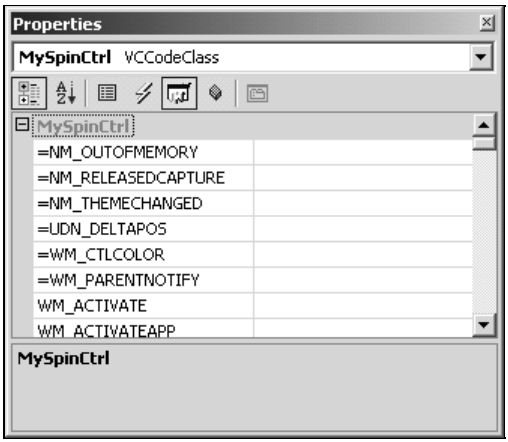


Рис. 5.4. Группа отраженных сообщений

9. Выделите строку `=UDN_DELTAPOS`. В соответствующей строке появится значок раскрывающегося списка.
10. Раскройте этот список и выделите в нем единственную строку. В приложение будет добавлена функция обработки соответствующего сообщения, откроется окно редактирования файла MySpinCtrl.cpp и текстовый курсор будет помещен в заготовку добавленной функции.
11. Измените функцию `OnDeltapos` в соответствии с текстом листинга 5.3.

Листинг 5.3. Функция `OnDeltapos`

```
// Изменение направления приращения инкрементного регулятора
void MySpinCtrl::OnDeltapos(NMHDR *pNMHDR, LRESULT *pResult)
{
  LPNMUPDOWN pNMUpDown = reinterpret_cast<LPNMUPDOWN>(pNMHDR);
```

```
  pNMUpDown-> iDelta = -pNMUpDown-> iDelta;
  *pResult = 0;
}
```

12. В окне **Class View** (Просмотр класса) выделите папку `CDialogDlg` и нажмите кнопку **Events** (События) в окне **Properties** (Свойства). Откроется список событий, обрабатываемых данным классом.
13. В этом списке раскройте папку `IDC_SPIN` и выделите строку `UDN_DELTAPOS`. У имени функции обработки данного сообщения появится значок раскрывающегося списка.
14. Раскройте этот список и выделите в нем строку **<Delete> OnDeltaposSpin**. Данная функция обработки сообщения будет удалена из приложения. (Текст ее реализации и ее записи в карте сообщений будут заремаркированы.)
15. В окне **Class View** (Просмотр класса) щелкните правой кнопкой мыши на папке `CDialogDlg` и выберите в появившемся контекстном меню команду **Add | Add Variable** (Добавить | Добавить переменную). Появится диалоговое окно **Add Member Variable Wizard - Dialog** (Мастер добавления переменной в класс), изображенное на рис. 5.5.

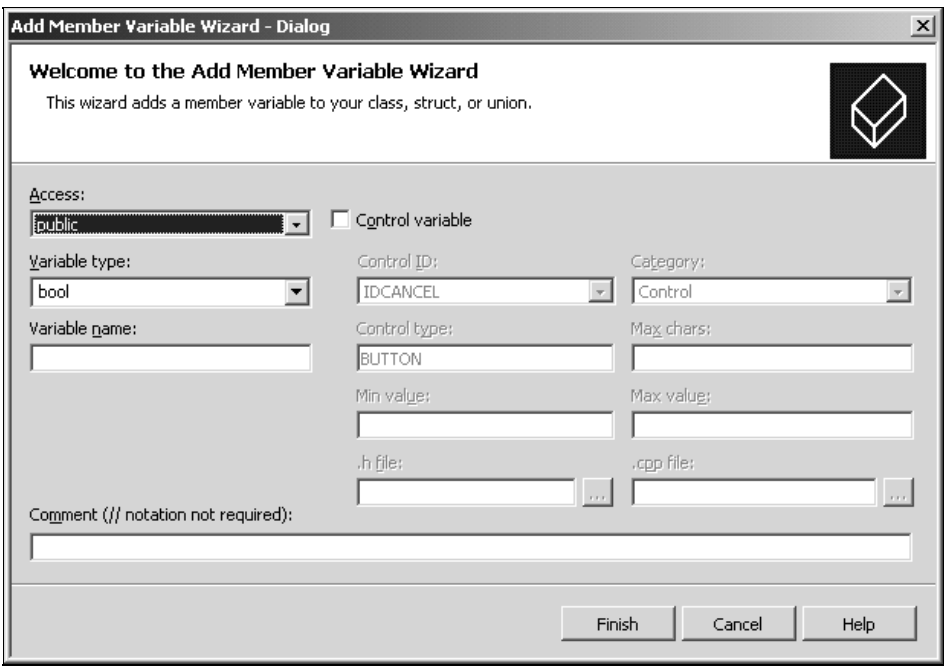


Рис. 5.5. Группа отраженных сообщений

16. Установите в нем флажок **Control variable** (Связь с элементом управления) и выделите в ставшем после этого раскрывающемся списке идентификатор ресурса `IDC_SPIN`.
17. Введите в текстовое поле раскрывающегося списка **Variable type** (Тип переменной) тип переменной `MySpinCtrl`, введите в текстовое поле **Variable name** (Имя переменной) имя переменной `m_Spin` и нажмите кнопку **Finish** (Готово). Данная переменная будет добавлена в класс.
18. Нажмите клавишу <F5> и запустите приложение на исполнение. В его работе ничего не изменится.

Суть проведенных операций заключается в том, что нами был создан пользовательский класс для работы с инкрементным регулятором и в нем была перегружена функция обработки сообщения `UDN_DELTAPOS`. Текст перегруженной функции полностью совпадает с текстом одноименной функции, помещенной нами в класс `CDialogDlg`. Поскольку теперь эта функция перегружается в классе элемента управления, она удаляется из класса диалогового окна.

На первый взгляд произведенные действия имеют мало смысла: удалив функцию в одном месте, мы тут же создали полностью идентичную функцию в другом месте, включив при этом в проект два дополнительных файла и один класс. Однако если поместить эти файлы с описанием класса в пользовательскую библиотеку, то, включив эту библиотеку в свое приложение и используя пользовательский класс для работы с инкрементным регулятором, можно забыть об особенностях этого элемента управления и не включать в каждое свое приложение функции, корректирующие его поведение.

Использование карты сообщений приложением

Каждое приложение содержит объект класса, производного от класса `CWinApp`, содержащего функцию — член класса `Run`. Эта функция вызывает функцию `CWinThread::Run`, содержащую свой собственный цикл сообщений, состоящий из вызовов функций `GetMessage`, `TranslateMessage` и `DispatchMessage`. Практически все объекты оконных классов используют в качестве базового класса класс `CWnd` и функцию `AfxWndProc`, ничем не отличающуюся от функции `WndProc`, используемую в старых программах.

Функция `WndProc` позволяет определить дескриптор окна, которому предназначено данное сообщение, а в библиотеке MFC содержится *карта дескрипторов* (handle map), содержащая таблицу дескрипторов окон и указателей на соответствующие им объекты. Совместное использование полученной информации позволяет определить указатель на объект класса `CWnd`, который должен осуществлять обработку данного сообщения. После этого в данном объекте вызывается виртуальная функция — член класса `WindowProc`. В каждом классе, производном от класса `CWnd`, производится перегрузка данной функции, позволяющая отразить все особенности обработки сообщений в данном конкретном классе окна.

Функция `WindowProc`, в свою очередь, вызывает функцию C++ `OnWndMessage`, которая, собственно, и обрабатывает сообщения. Сначала данная функция про-

веряет тип поступившего сообщения: является ли это собственно сообщением, командой или кодом извещения. Если поступило сообщение, то функция просматривает карту сообщений своего класса и находит в ней функцию обработки данного сообщения. Если в карте сообщения данного класса отсутствует макрос обработки данного сообщения, то просматривается карта сообщений родительского класса и так до тех пор, пока не будет найдена функция обработки данного сообщения. Механизм наследования карты сообщений работает аналогично механизму наследования C++, но полностью независим от него, что позволяет избежать многих проблем, возникающих при использовании виртуальных функций.

Использование библиотеки MFC позволяет существенно упростить для пользователя работу с картой сообщений, поскольку она берет на себя обработку многих стандартных сообщений и не отражает их в карте сообщений. Например, при выборе пользователем команды **File | Open** (Файл | Открыть) классы библиотеки MFC сами производят обработку посылаемого по этой команде сообщения и в результате данной обработки вызывают функцию `Serialize` соответствующего класса документа. При этом карта сообщений данного класса документа не содержит макроса обработки данного сообщения.

Создание функций обработки сообщений

Мы уже много раз описывали процедуру включения в проект функций обработки сообщений. Однако все сведения, касающиеся работы с ним, оказались разбросанными по разным главам. В данном разделе будет собрана основная информация, касающаяся этого вопроса.

Диалоговое окно *Properties*

Большинство операций, связанных с функциями обработки сообщений, производится в окне **Properties** (Свойства). Чтобы открыть окно **Properties** (если оно закрыто, что случается крайне редко), выберите команду **View | Properties Window** (Вид | Окно свойств), нажмите клавишу <F4> или щелкните правой кнопкой мыши на элементе, свойствами которого вы интересуетесь. В последнем случае на экране появится контекстное меню, показанное на рис. 5.6, в котором нужно выбрать команду **Properties** (Свойства).

Окно **Properties** (Свойства), изображенное на рис. 5.7, имеет несколько кнопок, раскрывающих в нем различные списки и управляющих выводом в них информации.

В окне, изображенном на рис. 5.7, нажаты кнопки **Categorized** (Разбиение по категориям), определяющая разбиение свойств по категориям, и **Properties** (Свойства), определяющая вывод списка основных свойств. При нажатии кнопки **Alphabetic** (Алфавитный), расположенной в той же группе, что и кнопка

Categorized (Разбиение по категорииям), свойства не разбиваются по категориям, а выводятся единым списком в алфавитном порядке.

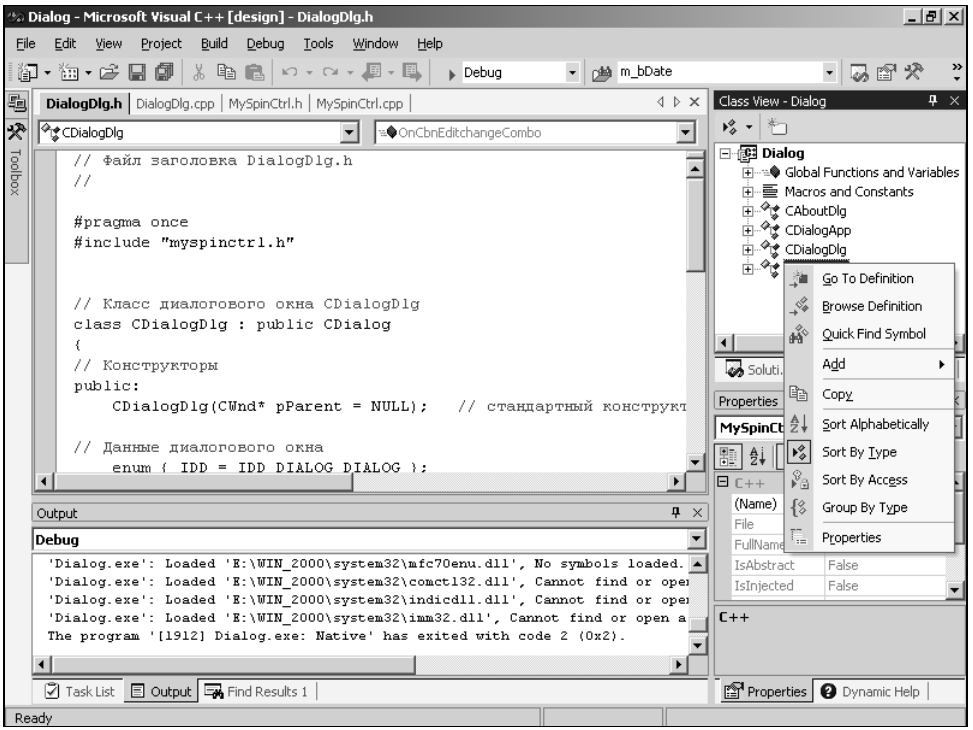


Рис. 5.6. Контекстное меню открытия окна **Properties**

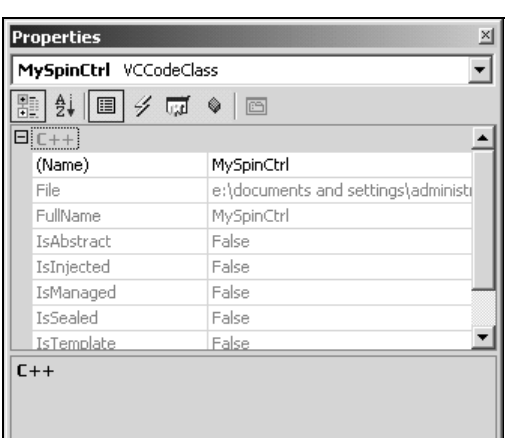


Рис. 5.7. Окно **Properties**

Для создания функций обработки сообщений используются кнопки **Event** (Событие) (**ControlEvents**) (События элементов управления) и **Messages** (Сообщения). Кнопка **ControlEvents** (События элементов управления) появляется в том случае, если пользователь выделил элемент управления в заготовке диалогового окна. В раскрывающемся при нажатии на эту кнопку списке содержатся идентификаторы событий, посылаемых данным элементом управления своему родительскому окну (рис. 5.8).

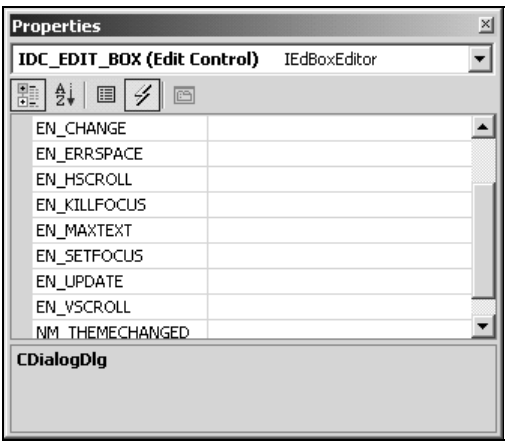


Рис. 5.8. Нажата кнопка **ControlEvents**

Чтобы добавить функцию обработки сообщения, необходимо выделить строку с идентификатором события, которое следует обработать. При этом в соответствующей строке появится значок раскрывающегося списка. Раскройте этот список. Окно **Properties** (Свойства) примет вид, изображенный на рис. 5.9.

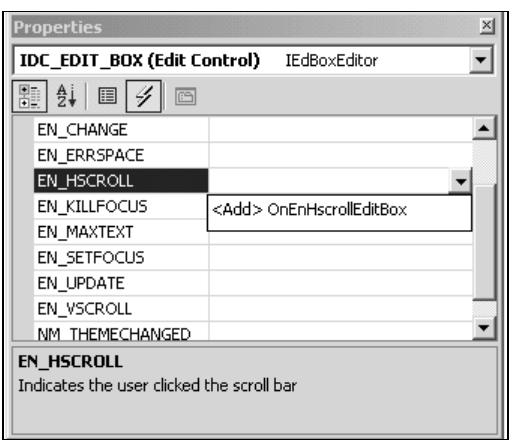


Рис. 5.9. Добавление функции обработки сообщения

Выделите в этом списке единственную строку. Функция обработки сообщения будет добавлена в приложение. При необходимости внести изменения в имя данной функции, это можно сделать непосредственно в текстовом поле раскрывающегося списка.

Для удаления функции обработки сообщения из приложения в окне **Properties** (Свойства) выделите строку, содержащую имя данной функции, и раскройте список, как показано на рис. 5.10.

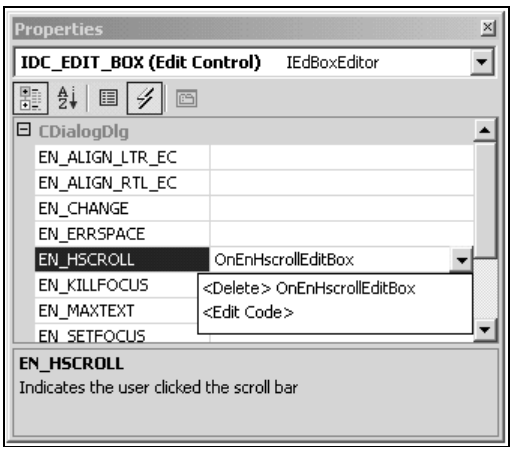


Рис. 5.10. Удаление функции обработки сообщения

Выделите в этом списке строку, начинающуюся с **<Delete>**. Функция обработки сообщения будет удалена из приложения. При этом в файле заголовка будет заремаркировано объявление данной функции, а в файле реализации будут заремаркированы соответствующий макрос карты сообщений и текст реализации данной функции.

Кнопка **Events** (События) (рис. 5.11) появляется в окне **Properties** (Свойства) в том случае, если пользователь выделил имя класса в окне **Class View** (Просмотр класса) или производит редактирование файла заголовка или файла реализации класса.

При нажатии этой кнопки в окне **Properties** (Свойства) раскрывается список идентификаторов элемента управления, события в которых могут обрабатываться в данном классе. Каждый из идентификаторов элементов управления в списке представляет собой папку, содержащую список событий, появляющихся при нажатии кнопки **ControlEvents** (События элементов управления), если в заготовке диалогового окна будет выделен соответствующий элемент управления. Работа со списком событий в папке полностью аналогична описанной выше работе с этим списком при выделении элемента управления.

Каждое окно может обрабатывать стандартные оконные сообщения. Кроме этого, каждому типу окон соответствует свой набор сообщений. Для доступа к этим сообщениям необходимо нажать кнопку **Messages** (Сообщения) в окне **Properties** (Свойства). Появляющийся при этом список сообщений показан на рис. 5.12.

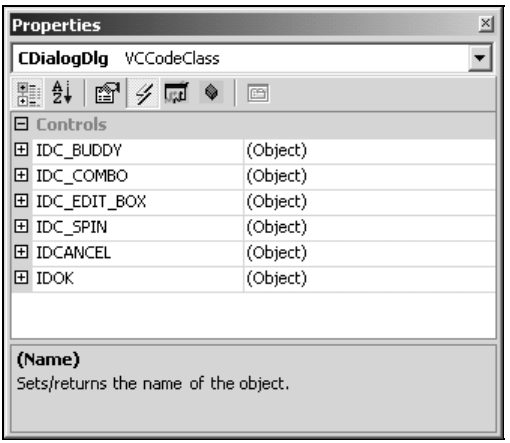


Рис. 5.11. Нажата кнопка **Events**

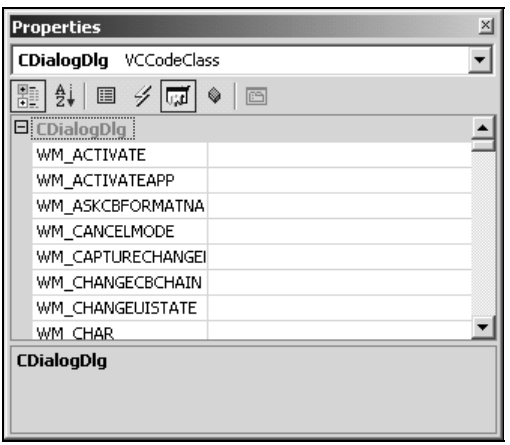


Рис. 5.12. Нажата кнопка **Messages**

Процедура создания и удаления функций обработки сообщений в списке **Messages** (Сообщения) аналогична процедуре создания и удаления функций обработки сообщений в списке **ControlEvents** (События элементов управления).

Список сообщений

В настоящее время в операционной системе Windows используется около 1000 различных сообщений. Конечно, в каждом классе имеется своя карта сообщений и количество обрабатываемых им сообщений обычно на несколько порядков меньше. Даже потенциально ни один класс не может обработать все имеющиеся сообщения, поскольку они разделяются на классы и, например, только потомок класса `CListBox` может получить сообщение `LB_SETSEL`, выделяющее в списке указанный в нем элемент.

По префиксу имени сообщения можно определить класс, объект которого послал данное сообщение, или объект класса окна, которому оно было направлено. В табл. 5.3 приведены префиксы некоторых сообщений и указано, какому окну они принадлежат.

Таблица 5.3. *Соответствия префиксов сообщений пославшему их окну*

Префикс	Описание
`BM`	Сообщение от кнопки
`BN`	Извещение от кнопки
`CB`	Сообщение от комбинированного списка
`CBN`	Извещение от комбинированного списка
`CDM`	Сообщение от стандартного диалогового окна
`CDN`	Извещение от стандартного диалогового окна
`CPL`	Сообщение от панели управления
`DM`	Сообщение о нажатии кнопки по умолчанию в диалоговом окне

Нет никакой необходимости продолжать этот список, поскольку за правильностью синтаксиса идентификаторов элементов управления следит среда программирования Visual C++, а окно, породившее данное сообщение ясно из контекста программы.

Обновление команд

Мы уже ознакомились с сообщениями и извещениями, но до сих пор не выяснили, что же представляют собой команды. Команды представляют собой извещения формата `WM_COMMAND`, посылаемые объектами классов пользовательского интерфейса, такими как меню, кнопки панели инструментов и клавиши акселераторов. В Windows 3.*x* сообщение `WM_COMMAND` формировалось как в ответ на выбор команды меню, так и в ответ на нажатие кнопки. В Win32 сообщение `WM_COMMAND` формируется только в ответ на выбор команды меню, а нажатие

кнопки или выделение элемента в списке порождает сообщение WM_NOTIFY с кодом извещения элемента управления.

Отличительной чертой команд является необходимость задания состояния связанных с ними элементов управления. Обычно, команды меню и кнопки панели инструментов могут находиться в различных состояниях. Например, команда меню в данной конкретной ситуации может быть недоступной. Команды меню могут, также, отмечаться флажками и переключателями. Все вышеперечисленное полностью относится и к кнопкам панели инструментов.

Для определения текущего состояния подобных элементов управления приложение должно при каждом обновлении своего состояния иметь информацию о том, в каком состоянии при этом должен находиться каждый из его элементов управления. Эту информацию может дать только сам программист. Логично предположить, что если команда меню посылает сообщение, обрабатываемое некоторым документом, то именно этот документ и должен определять состояние команды в меню, поскольку в данном документе должна содержаться вся необходимая для этого информация.

Если одной и той же команде соответствуют различные элементы пользовательского интерфейса (например, пункт меню и кнопка панели инструментов), то их состояние должно определяться одной и той же функцией обработки сообщения.

В библиотеке MFC предусмотрен удобный интерфейс управления состоянием подобных элементов управления. Пользователь может использовать для этого свои функции и методы, но обычно для этого бывает достаточно воспользоваться стандартными методами.

При раскрытии меню приложение запускает механизм проверки состояния команд в данном меню. Для этого приложение посылает сообщение об обновлении всем элементам управления, содержащимся в данном меню. Для каждой команды обновления меню находится соответствующий макрос в карте сообщений, имеющий форму ON_UPDATE_COMMAND_UI, и вызывается указанная в нем функция обработки сообщения. Таким образом, для меню, содержащего шесть команд, вызывается шесть функций обработки сообщения. Если пользователем указана функция обработки сообщения об обновлении состояния команды с данным идентификатором, то вызывается данная функция. В противном случае приложение проверяет, имеется ли в классе функция обработки для команды с данным идентификатором, и, в соответствии с результатами проверки, делает данную команду доступной или недоступной.

Если приложение не находит соответствующий макрос ON_UPDATE_COMMAND_UI, то оно делает команду доступной, если в карте сообщений активного класса данного приложения имеется макрос ON_COMMAND для идентификатора данной команды. Если же такой макрос отсутствует, то команда делается недоступной. Поэтому, чтобы сделать команду меню доступной, для нее нужно написать функцию обработки.

Типичная функция обработки сообщения об обновлении команды выглядит следующим образом:

```
afx_msg void OnUpdateMyControl (CCmdUI* pCmdUI);
```

Ее аргументом является объект класса `CCmdUI`. Функции — члены данного класса перечислены в табл. 5.4.

Таблица 5.4. *Функции — члены класса* `CCmdUI`

Функция	Описание
`Enable`	Делает элемент управления доступным или недоступным
`SetCheck`	Отмечает элемент управления флажком
`SetRadio`	Отмечает элемент управления переключателем
`SetText`	Устанавливает текст в элементе управления
`ContinueRouting`	Указывает механизму обработки сообщений, что данная функция не является конечной и что необходимо продолжить обработку данного сообщения

Глава 6

Вывод информации на экран

Заготовки многооконных и однооконных приложений, создаваемые мастером создания приложений MFC, при запуске их на исполнение выводят на экран пустое окно документа. Данная глава посвящена вопросу о том, каким образом это окно может быть заполнено полезной информацией.

Существует четыре принципиально разные возможности заполнить окно документа в приложении:

- использовать линии для рисования фигур;
- использовать кисти для заполнения замкнутых геометрических фигур;
- вывод тестовой информации;
- вывод битовых образов.

Однако прежде чем перейти к рассмотрению вопросов, связанных с каждой из этих возможностей, необходимо рассмотреть интерфейс графических устройств Windows (GDI), используемый приложением для вывода любой информации на экран.

Интерфейс графических устройств (GDI)

Программы, работающие под управлением MS DOS, должны самостоятельно управлять устройствами отображения информации, с которыми они работают. Так, например, любой текстовый редактор в MS DOS, обеспечивающий вывод редактируемой информации на печать, должен включать в себя драйверы всех принтеров, на которые предполагается выводить информацию.

Одной из основополагающих концепций Windows является независимость текстов программ от используемых аппаратных средств. Поэтому, для работы с устройствами отображения информации приложения Windows используют *интерфейс графических устройств* (Graphic Device Interface или GDI) Windows, а не обращаются непосредственно к устройствам.

Эта особенность программ Windows, с одной стороны, усложняет методику программирования операций отображения информации, но, с другой стороны, избавляет программиста от необходимости использования в программе драйверов всевозможных устройств и гарантирует ему, что его программа будет работать даже с теми устройствами, которые еще не созданы на момент написания программы. Управление устройствами осуществляется с использованием драй-

веров устройств, которые устанавливаются в процессе инсталляции операционной системы Windows на конкретный компьютер или поставляются вместе с вновь устанавливаемым на него оборудованием. Эти драйверы получают информацию из приложения, производят ее обработку, и формируют на основе результатов этой обработки сигналы управления для конкретного устройства.

Ясно, что цветной принтер и цветной монитор решают, в принципе, одну и ту же задачу: выводят двумерную графическую информацию в растровой форме. Поэтому для их управления могут быть использованы одни и те же команды, но, с другой стороны, никакой цветной принтер не обеспечит 24-битной глубины цвета и, пока что, никакой дисплей не имеет разрешающей способности 2400×1200 точек на дюйм. Поэтому при получении от приложения задания вывести одно и то же изображение драйвер принтера должен преобразовать его цвета в цвета, имеющиеся в палитре данного принтера, а драйвер монитора должен объединить соседние точки изображения для того, чтобы результат преобразования соответствовал разрешающей способности установленного на компьютере дисплея.

Поскольку программист при работе с устройствами отображения информации не должен отвлекаться на особенности конкретного устройства, Windows предоставляет ему возможность написания программ, работающих с некоторым виртуальным устройством, имеющим превосходные характеристики, которых не имеет ни одно реальное устройство. В тот момент, когда возникнет необходимость вывести изображение, созданное для этого идеального устройства на некоторое реальное устройство, приложение запросит его контекст, определит возможности данного устройства отображения информации и с минимальными потерями преобразует это изображение таким образом, чтобы его параметры полностью соответствовали возможностям реального устройства.

Контекст устройства

Контекст устройства представляет собой структуру данных, определенную в операционной системе Windows и содержащую информацию о наборе *графических объектов* и связанных с ними атрибутах таких устройств, а также о графических режимах вывода информации. Под графическими объектами подразумеваются перья для рисования линий, кисти для закрашивания замкнутых контуров, битовые образы для вывода на экран или прокрутки изображения в окне, палитры, определяющие набор доступных цветов, регионы для усечения изображения и подобных операций, а также пути для операций рисования.

Все отображение информации в Windows может осуществляться только с использованием объектов классов контекста устройств, содержащих методы Windows API, реализующие рисование линий, форм и вывод текста. Использование контекста устройства позволяет осуществлять независимый от устройства вывод графической информации в Windows. С его помощью графическая информация может быть выведена на экран, на принтер или в метафайл.

В классе `CPaintDC` реализованы основные принципы работы с контекстом устройств Windows. Конструктор данного класса вызывает функцию `CWnd::BeginPaint`, а его деструктор вызывает функцию `CWnd::EndPaint`. При вызове функции `CView::OnDraw` приложение само вызывает функцию `CView::OnPrepareDC` и передает пользователю указатель на объект класса `CDC`. Уничтожение этого объекта приложение также берет на себя.

Класс `CClientDC` содержит методы, позволяющие работать только в рабочей области окна. Конструктор данного класса вызывает функцию `CWnd::GetDC`, а его деструктор — функцию `CWnd::ReleaseDC`. Большинство процедур рисования располагается в функциях `OnDraw`, вызываемых при создании окна и при необходимости его перерисовки. Однако в некоторых случаях требуется вывести в окно некоторый графический объект, не прибегая к полной или частичной перерисовке окна. Именно в этом случае и следует использовать для задания контекста устройства объект класса `CClientDC`.

Класс `CWindowDC` содержит методы, позволяющие работать со всем окном, включая его рамку. Класс `CMetaFileDC` содержит методы, позволяющие сохранять графические объекты в метафайлах. В отличие от класса `CPaintDC`, указатель на который передается в функции `OnDraw`, данный класс не вызывает функцию `OnPrepareDC` и ее вызов должен осуществлять пользователь.

Как уже отмечалось ранее, почти все операции рисования в приложении осуществляются функцией `OnDraw`, перегружаемой в каждом производном от нее классе. В перегруженной функции `OnDraw` необходимо произвести следующие действия:

- получить данные из связанного с данным представлением документа;
- вывести эти данные в графическом виде с использованием объекта класса контекста устройства, указатель на который передается данной функции в качестве параметра.

В случае внесения каких-либо изменений в данные, содержащиеся в документе, эти изменения должны быть немедленно отображены на экране. Обычно все изменения вносятся в документ связанным с ним объектом класса представления. Если с данным документом связаны несколько объектов представления, то для внесения изменений во все эти объекты класс представления, через который вносятся изменения, получает указатель на связанный с ним объект документа и вызывает функцию `CDocument::UpdateAllViews`, имеющую три аргумента, первым из которых является указатель на объект класса `CView`, в котором произошли изменения, вызвавшие модификацию документа. Если первый аргумент данной функции равен нулю, то необходимо внести изменения во все объекты представления. Второй параметр функции `UpdateAllViews` имеет тип `LPARAM` и содержит информацию о необходимых изменениях. Третий параметр данной функции представляет собой указатель на объект класса `CObject`, в котором также хранится информация о необходимых изменениях.

Данная функция, обычно, вызывается после внесения изменений в документ и после вызова функции `CDocument::SetModifiedFlag`. Функция `UpdateAllViews` посы-

лает сообщения всем объектам класса представления, за исключением объекта на который указывает первый аргумент данной функции, о внесении изменений в связанный с ним документ. Для этого она вызывает функцию `CView::OnUpdate`, имеющую тот же набор аргументов, что и вызывающая функция.

Как только представление получает сообщение о необходимости перерисовки, Windows посылает сообщение `WM_PAINT`. В классе представления имеется функция обработки данного сообщения `CWnd::OnPaint`, создающая объект класса контекста устройств `CPaintDC` и вызывающая функцию `OnDraw` данного класса представления. Обычно пользователь не должен перегружать функцию `OnPaint`.

Типичный пример перегруженной функции `OnDraw` приведен ниже.

```
void CMyView::OnDraw(CDC* pDC)
{
  CMyDoc*  pDoc = GetDocument();
  CString  s = pDoc-> szDemoString;
  CRect rect;

  GetClientRect(&rect);
  pDC->SetTextAlign(TA_BASELINE  |  TA_CENTER);
  pDC->TextOut(rect.right / 2, rect.bottom / 2, s, s.GetLength());
}
```

В данном примере приложение получает указатель на объект документа, связанного с данным представлением. Затем, используя полученный указатель на объект класса документа, данная функция получает доступ к переменной — члену класса документа `szDemoString`. После этого полученная из документа строка выводится посередине экрана. Если данная функция осуществляет вывод на экран, то передаваемый ей указатель на объект класса `CDC` в действительности является указателем на объект класса `CPaintDC`, для которого уже вызвана функция `BeginPaint`. Для вызова функций работы с графическими объектами используются методы соответствующего класса контекста устройства.

Отображение текста

Человеку, знакомому с типографскими терминами, может показаться странным их трактовка в системе Windows, однако большинство терминологических различий между шрифтами Win32 и стандартными типографскими шрифтами отражают различие используемых технологий печати. Стандартные типографские термины связаны с технологическим процессом отливки матриц, в то время как используемые в Win32 термины связаны с компьютерной версткой текстов и печатью их на принтерах.

Шрифты

Под *шрифтом* понимается набор букв и символов, объединенных единым дизайном. Основными элементами, определяющими дизайн шрифта, являются начертание, стиль и размер.

Под термином *начертание* понимаются отличительные черты букв и символов данного шрифта, такие как ширина жирных и тонких штрихов, составляющих символы, а также наличие или отсутствие засечек. Засечка представляет собой короткую поперечную линию на открытых концах штрихов. Шрифт или начертание, не имеющее засечек, обычно называется "Sanserif".

Термин *стиль* определяет толщину линии знака и его наклон. Шрифты Win32 могут иметь следующие значения толщины линии (начиная с самой тонкой и кончая самой толстой):

- thin (самая тонкая);
- extralight (очень тонкая);
- light (тонкая);
- normal (нормальная);
- medium (средняя);
- semibold (полужирная);
- bold (толстая);
- extrabold (очень толстая);
- heavy (самая толстая).

Параметр наклона шрифта может принимать одно из трех значений: roman (без наклона), oblique (наклонный) и italic (курсив). Символы со значением параметра roman характеризуются отсутствием наклона. Параметр oblique означает искусственный наклон шрифта. Наклон достигается специальным начертанием шрифтов roman. Шрифты с параметром italic действительно наклонены.

Размер шрифта не может быть определен точно. Обычно он определяется как вертикальное расстояние между нижней точкой прописной буквы g и верхней точкой заглавной буквы M. Размер шрифта измеряется в пиках. Одна пика составляет 0,013837 дюйма. В соответствии с системой пик, разработанной Пьером Симоном Фурнье, под одной пикой обычно понимают 1/72 дюйма.

Под термином *семейство шрифтов* (font family) понимается совокупность шрифтов, имеющих единую ширину штрихов и засечек. Всего имеется пять семейств шрифтов. Шестое семейство предполагает использование в приложении шрифтов по умолчанию. Ниже перечислены семейства шрифтов и даны краткие их характеристики.

- Decorative — определяет нестандартный шрифт. Например, Old English.
- Dontcare — является общим именем семейства шрифтов. Это имя используется в том случае, когда отсутствует информация о шрифте или используе-

мый шрифт не имеет значения. В этом случае используется шрифт по умолчанию.

- Modern — определяет шрифт с одинаковой шириной букв, с засечками и без них. К таким шрифтам, как правило, относятся современные шрифты, такие, как Pica, Elite и Courier New.
- Roman — определяет пропорциональный шрифт с засечками. Примером такого шрифта является Times New Roman.
- Script — определяет шрифт, который выглядит как рукописный. Примером такого шрифта являются Script и Cursive.
- Swiss — определяет пропорциональный шрифт без засечек. Примером такого шрифта является Arial.

Данным шрифтам в файле заголовка WinGdi.h соответствуют предопределенные константы `FF_DECORATIVE`, `FF_DONTCARE`, `FF_MODERN`, `FF_ROMAN`, `FF_SCRIPT` и `FF_SWISS`. Приложение использует эти константы при создании и выборе шрифта, а также при получении информации о шрифте.

Приложения Win32 могут использовать четыре различных способа вывода шрифтов на экран или принтер: растровый, векторный, TrueType, и OpenType. Различия между этими шрифтами состоят в способе хранения данного символа в памяти компьютера или в соответствующем файле ресурса шрифта. В растровых шрифтах символ хранится в виде битового изображения, которое копируется приложением в ту позицию экрана или распечатки, в которую требуется поместить изображение данного символа. В векторных шрифтах символ представлен конечными точками отрезков линий, которые рисует приложение при выводе данного символа. В шрифтах TrueType и OpenType символ представляется набором команд рисования отрезков прямых линий и сегментов кривых, а также набором подсказок. Команды рисования отрезков линий используются для грубого задания начертания символа в шрифтах TrueType или OpenType. Система подсказок позволяет отредактировать положения начальных и конечных точек отрезков, используемых при рисовании символов. Эти подсказки и соответствующие им корректировки учитывают текущий масштаб отображения символа. Шрифт OpenType эквивалентен шрифту TrueType за исключением того, что шрифт OpenType допускает использование команд Postscript при формировании начертания шрифта.

Поскольку битовые образы каждого символа растрового шрифта разрабатывались с учетом разрешающей способности устройства, на котором они должны отображаться, то растровые шрифты считаются шрифтами, зависящими от устройства. С другой стороны, векторные шрифты не зависят от устройства, на которое они будут выводиться, поскольку каждый их символ представлен легко масштабируемым набором конечных точек отрезков. Однако вывод векторных шрифтов требует больших вычислительных затрат, чем вывод растровых шрифтов или шрифтов TrueType и OpenType. Шрифты TrueType и OpenType обеспечивают достаточно высокую скорость вывода символов и полную независимость от устройства. Использование подсказок, содержащихся в описании символов

шрифта, позволяет легко масштабировать символы шрифтов TrueType и OpenType без потери качества их отображения.

Как уже говорилось, описание символов шрифта хранится в файле ресурса шрифта. Файл ресурса шрифта фактически представляет собой библиотеку динамической компоновки, которая содержит данные, но не содержит методов их обработки. Для растровых и векторных шрифтов данная библиотека состоит из двух разделов: заголовка, описывающего метрику шрифта, и данных, описывающих символы данного шрифта. Файл ресурса шрифта для растрового или векторного шрифта имеет расширение fon. Каждому из шрифтов TrueType или OpenType соответствуют два файла: первый файл содержит относительно короткий заголовок, а второй — содержит данные о символах данного шрифта. Первый файл имеет расширение fot, а второй — расширение ttf.

Каждый шрифт характеризуется набором включенных в него символов. Набор символов включает в себя знаки препинания, символы верхнего и нижнего регистра, а также все другие печатаемые символы. Каждому элементу символьного набора соответствует свой номер.

Большинство символьных наборов использует набор символов ASCII, определенный стандартом США, содержащий 96 символов, которым соответствуют номера от 32 до 127. Имеются пять основных групп символьных наборов:

- Windows;
- Unicode;
- OEM;
- Symbol;
- специальные наборы.

В приложениях Win32 обычно используется набор символов Windows. Этот набор практически полностью эквивалентен набору символов ANSI. Первым символом в наборе символов Windows является знак пробела. Он имеет шестнадцатеричное значение 0x20 (десятичное число 32). Последний символ в наборе символов Windows имеет шестнадцатеричное значение 0xFF (десятичное число 255).

Во многих шрифтах определяется символ по умолчанию. Всякий раз, когда данный шрифт получает код запроса, которому не соответствует ни один из его символов, в ответ на этот запрос шрифт посылает символ по умолчанию. Многие шрифты, использующие набор символов Windows, определяют в качестве символа по умолчанию точку (.). Шрифты TrueType и OpenType обычно используют в этом качестве символ открытого поля.

Многие шрифты используют символ абзаца для выделения слов и выравнивания текста. Большинство шрифтов, использующих набор символов Windows, определяют в качестве знака абзаца символ пробела.

В Microsoft Windows версии 3.1 к кодовой странице Windows добавлено 24 символа, обозначающие символы валют, крестики, кавычки, тире и другие служебные символы.

Символьный набор Windows использует для идентификации каждого символа байт или 8 битов, следовательно, максимальное количество символов, которое может содержать данный набор, составляет 256. Этого количества символов обычно бывает достаточно для западных языков, включая представление диакритических знаков, используемых во французском, немецком, испанском и других языках. Однако восточные языки используют тысячи различных символов, которые не могут быть закодированы с применением одного байта. Стремительный рост компьютерной торговли привел к необходимости создания двухбайтовых схем кодирования, позволяющих идентифицировать символы по 8-разрядным, 16-разрядным, 24-разрядным или 32-разрядным последовательностям. Эти схемы требуют сложных алгоритмов дешифрации, но даже они не обеспечивают того, что один и тот же текст будет одинаково воспроизведен на двух различных компьютерах.

Для решения этой задачи был разработан стандарт Unicode, позволяющий кодировать символы 16-разрядным кодом. В результате стало возможно кодировать до 65 536 символов, что достаточно для работы со всеми шрифтами, используемыми в настоящее время, а также со всеми знаками препинания, математическими символами и, кроме того, еще осталось достаточно свободных кодов для дальнейшего расширения. Unicode устанавливает уникальный код для каждого символа, что гарантирует правильное декодирование символа любого языка.

Набор символов OEM обычно используется программами, работающими в операционной системе MS-DOS. Символам с кодами от 32 до 127 обычно соответствуют те же самые символы, что и в стандарте США ASCII, и в наборе символов Windows. Другим кодам в наборе символов OEM (от 0 до 31 и от 128 до 255) соответствуют символы, используемые при отображении текста в программах MS-DOS. Эти символы отличаются от символов Windows.

Набор символов Symbol содержит специальные символы, обычно используемые в математических и научных формулах.

Многие принтеры и другие устройства вывода работают со шрифтами, использующими символьные наборы, отличающиеся от символьных наборов Windows и OEM. Примером такого символьного набора может служить расширенный набор двоично-десятичных символов (EBCDIC). Чтобы использовать подобный символьный набор, драйвер принтера должен обеспечивать трансляцию набора символов Windows в данный специализированный набор.

Шрифт может быть уже установлен на используемом приложением устройстве вывода информации. В противном случае для вывода текста необходимо загрузить данный шрифт в системную таблицу шрифтов. Таблица шрифтов представляет собой внутренний массив, идентифицирующий загружаемые в устройство шрифты, используемые данным приложением Win32. Приложение может получать имена шрифтов, загруженных на данный момент в устройство и хранящихся во внутренней таблице шрифтов, используя функции `EnumFontFamilies` и `ChooseFont`.

Функция `EnumFontFamilies`, первым аргументом которой является дескриптор контекста устройства, позволяет получить информацию для каждого шрифта

семейства, указанного во втором ее аргументе, и передать эту информацию функции обратного вызова, указатель на которую является ее третьим аргументом, а указатель на передаваемые функции обратного вызова данных содержится в четвертом аргументе функции `EnumFontFamilies`. Данная функция возвращает последнее значение, возвращенное функцией обратного вызова.

Функция `ChooseFont` вызывает диалоговое окно **Font** (Шрифт), изображенное на рис. 6.1. Данное окно позволяет пользователю выбрать атрибуты логического шрифта. Эти атрибуты включают в себя имя начертания, стиль (полужирный, наклонный или обычный), размер шрифта в пиках и эффекты (подчеркивание, зачеркивание и цвет текста). Единственным параметром данной функции является указатель на структуру `CHOOSEFONT`.

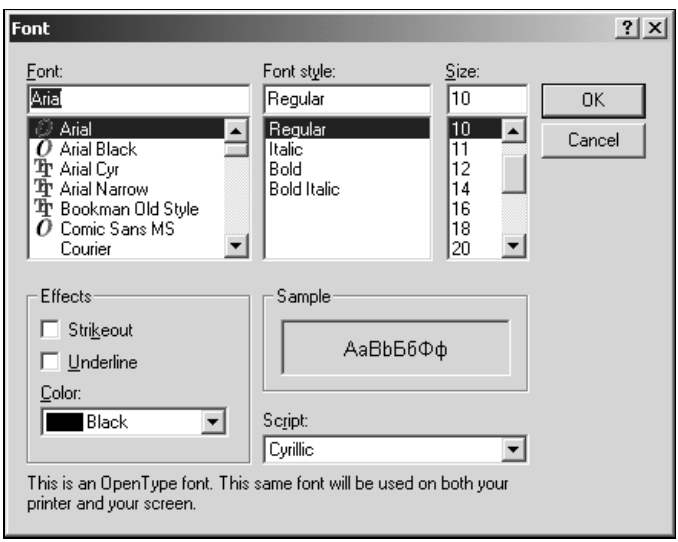


Рис. 6.1. Диалоговое окно **Font**

Приложение может устанавливать шрифт, используя вызов функции `AddFontResource`, единственным параметром которой является имя файла, содержащего ресурс данного шрифта. Данный файл может иметь расширение fon, fnt, ttf или fot.

При использовании шрифтов TrueType и OpenType иногда необходим дополнительный шаг прежде, чем шрифт может быть установлен в таблицу шрифтов. Некоторые разработчики шрифтов поставляют для этих шрифтов только файлы данных (имеющие расширение ttf). Прежде чем система сможет загрузить подобный файл, требуется создать соответствующий файл заголовка (имеющий расширение fot). Чтобы создать этот файл, приложение должно вызвать функцию `CreateScalableFontResource` и передать ей имя файла данных шрифта в качестве третьего аргумента. Первый аргумент данной функции определяет, бу-

дет ли этот шрифт впоследствии модифицироваться. Второй аргумент содержит указатель на имя создаваемого файла заголовка, а четвертый — путь к этому файлу. Когда файл заголовка создан, приложение может устанавливать шрифт, для чего необходимо вызвать функцию `AddFontResource` и передать ей в качестве аргумента имя нового файла заголовка.

Когда приложение заканчивает использовать установленный шрифт, оно должно удалить этот шрифт, вызывая функцию `RemoveFontResource`. Указав ей в качестве аргумента то же самое имя файла, что было указано в качестве аргумента функции `AddFontResource`.

Всякий раз, когда приложение вызывает функции, добавляющие или удаляющие ресурсы шрифта, оно должно также вызвать функцию `SendMessage` и послать сообщение `WM_FONTCHANGE` всем главным окнам приложений в системе. Это сообщение извещает другие приложения о том, что внутренняя таблица шрифтов была изменена приложением, которое добавило или удалило шрифт.

После того как пользователь выбирает шрифт в диалоговом окне **Font** (Шрифт) и нажимает кнопку **OK**, функция `ChooseFont` инициализирует элементы структуры `LOGFONT` с атрибутами запрошенного шрифта. Чтобы использовать этот шрифт для действий с текстовым выводом, приложение должно сначала создать логический шрифт, а затем выбирать этот шрифт в контекст устройства. Под логическим шрифтом понимается содержащееся в приложении описание идеального шрифта. Пользователь может создать логический шрифт с использованием функций `CreateFont` или `CreateFontIndirect`. В большинстве случаев целесообразнее вызывать функцию `CreateFontIndirect`, поскольку функция `CreateFont` имеет несколько аргументов, функция `CreateFontIndirect` использует указатель на объект структуры `LOGFONT`.

Прежде чем приложение сможет выводить текст с использованием логического шрифта, оно должно найти ему наиболее близкое соответствие среди шрифтов, хранящихся во внутренней памяти устройств, и шрифтов, чьи ресурсы были загружены в операционную систему. Шрифты, хранящиеся в устройстве или в операционной системе, называются физическими шрифтами. Процесс нахождения физического шрифта, максимально соответствующего определенному логическому шрифту, называется подстановкой шрифтов. Этот процесс происходит при вызове из прикладной программы функции `SelectObject`. Подстановка шрифтов выполняется внутренней процедурой операционной системы, сравнивающей атрибуты запрошенного логического шрифта и атрибуты доступных физических шрифтов.

Функция `SetMapperFlags` определяет необходимость учета при сопоставлении логического и физических шрифтов коэффициента сжатия, характерного для данного физического устройства. Коэффициент сжатия устройства определяется как отношение ширины и высоты элемента изображения в данном устройстве.

При передаче документов на другой компьютер часто используется технология внедрения шрифтов в документ. Внедрение шрифта гарантирует, что шрифт, определенный в передаваемом файле документа, будет присутствовать на компьютере, получающем данный файл. Не все шрифты могут перемещаться с од-

ного компьютера на другой, поскольку большинство шрифтов имеют лицензию на использование только на одном компьютере. Внедряться могут только шрифты TrueType и OpenType.

Приложения могут внедрять шрифты в документ только по требованию пользователя. Дистрибутив приложения не может содержать документы с внедренными в них шрифтами, и само приложение не может содержать внедренный шрифт. Всякий раз, когда совместно с приложением поставляются шрифты в любом формате, должны быть подтверждены права собственности владельца шрифта.

Несоблюдение приведенных выше правил может быть расценено как нарушение прав собственности поставщика шрифта или лицензионного соглашения пользователя. Лицензия может содержать разрешение на чтение и запись шрифта, который будет установлен на компьютере адресата. Лицензия может содержать разрешение только на чтение. Подобное разрешение допускает просмотр и печать (но не изменение) документа на компьютере адресата. Документы с внедренными шрифтами, имеющими лицензию только для чтения, сами являются документами только для чтения. Внедренные шрифты, имеющие лицензию только для чтения, не могут быть извлечены из документа и установлены на компьютере адресата.

Приложение может определять состояние лицензии путем вызова функции `GetOutlineTextMetrics` и исследуя переменную `otmfsType`, являющуюся членом структуры `OUTLINETEXTMETRIC`, объект которой является третьим аргументом данной функции. Если бит 1 переменной `otmfsType` установлен, то для данного шрифта внедрение запрещено. В противном случае этот шрифт может быть внедрен. Если в данной переменной установлен бит 2, то данный шрифт имеет лицензию только на чтение.

Для внедрения шрифта приложение может использовать функцию `GetFontData`, производящую чтение файла шрифта. Для того чтобы приложение прочитало данный файл полностью, необходимо установить значения аргументов данной функции `dwTable` и `dwOffset` в 0L, а значение аргумента `cbData` — в −1L.

После того как приложение получает всю необходимую информацию о шрифте, оно может сохранить ее вместе с документом, используя любой подходящий формат. Большинство приложений формирует в документе специальный каталог для шрифтов, перечисляя в нем все внедренные шрифты и режимы работы с ними. Приложение может использовать для идентификации шрифта переменные `otmpStyleName` и `otmFamilyName`, являющиеся объектами структуры `OUTLINETEXTMETRIC`.

Если для внедренного шрифта установлен бит только для чтения, приложение должно зашифровать данные шрифта перед сохранением его в документе. Метод шифрования не должен быть слишком сложным. Для этого можно, например, использовать операцию "исключающего ИЛИ" между данными шрифта и некоторой константой, определенной в приложении.

Вывод текста

Вывод текста является наиболее распространенным типом вывода графической информации в рабочую область окон приложений Win32. Текстовая информация используется в системах подготовки документов, позволяющих задать достаточно сложные параметры форматирования текста при работе с электронными таблицами, базами данных, в системах автоматизированного проектирования и в других приложениях.

Win32 API обеспечивает функционально полный набор методов, позволяющих форматировать и рисовать текст в рабочей области окна приложения и выводить его на принтер. Эти функции могут быть разделены на две категории: функции форматирования текста (или подготовки его к выводу) и функции фактического вывода текста на устройство. Функции форматирования осуществляют выравнивание текста, устанавливают межсимвольные интервалы, цвет текста и цвет фона и выравнивают текст. Функции вывода текста осуществляют рисование отдельных его символов или всей строки текста.

Форматирование текста

Функции форматирования текста могут быть условно разделены на три группы:

- функции установки атрибутов форматирования текста в контексте устройства;
- функции определения ширины символа;
- функции определения ширины и высоты строки текста.

Группа установки атрибутов форматирования текста в контексте устройства состоит из шести функций: `SetBkColor`, `SetBkMode`, `SetTextAlign`, `SetTextCharacterExtra`, `SetTextColor` и `SetTextJustification`. Эти функции устанавливают параметры юстировки текста, межсимвольный интервал, выравнивание текста, а также цвет текста и фона. Кроме того, существуют еще шесть функций, используемых для извлечения текущих установок форматирования текста из любого контекста устройства: `GetBkColor`, `GetBkMode`, `GetTextAlign`, `GetTextCharacterExtra`, `GetTextColor` и `GetTextExtentPoint32`.

Функция `SetTextAlign` позволяет определить, каким образом приложение должно позиционировать символы в строке текста при выводе его на устройство. Эта функция может использоваться для юстировки заголовков, номеров страниц, ссылок и т. д. Система юстирует строку текста, выравнивая репер на виртуальном прямоугольнике, охватывающем данную строку. В зависимости от значения второго аргумента данной функции репер устанавливается в различных точках данного прямоугольника, например, в его левом нижнем углу. По умолчанию репер устанавливается в верхнем левом углу этого прямоугольника. Приложение может получить информацию о текущих установках параметров юстировки текста в контексте устройства с помощью вызова функции `GetTextAlign`.

Функция `SetTextCharacterExtra` позволяет изменить межсимвольный интервал при выполнении всех операций вывода текста с использованием данного кон-

текста устройства. По умолчанию для любого контекста устройства устанавливается нулевое значение межсимвольного интервала. Приложение может получить информацию о текущих установках параметров межсимвольного интервала в контексте устройства для данного контекста устройства с помощью вызова функции `GetTextCharacterExtra`.

Функции `GetTextExtentPoint32` и `SetTextJustification` позволяют установить параметры выравнивания строки текста. Операция текстового выравнивания широко применяется во всех настольных издательских системах и в большинстве приложений подготовки текстов. Функция `GetTextExtentPoint32` вычисляет ширину и высоту строки текста. На основании информации о ширине и высоте строки текста функция `SetTextJustification` определяет размеры дополнительных интервалов между словами текста в строке, позволяющими выровнять эту строку по ширине.

Для установки цвета текста, выводимого в рабочую область окна приложения или на цветной принтер, используется функция `SetTextColor`. Для установки цвета фона, на котором выводится отдельный символ, используется функция `SetBkColor`, а для установки цвета фона всей рабочей области окна — функция `SetBkMode`.

По умолчанию для вывода текста в контексте устройства выбирается черный цвет, а для отображения фона — белый. При этом выбирается режим непрозрачного фона. Приложение может получить информацию о текущих установках цветов текста в контексте устройства с помощью вызова функции `GetTextColor`. Соответствующая информация о цвете фона символа и рабочей области окна выбирается с использованием функций `GetBkColor` и `GetBkMode`.

Приложения должны получать информацию о ширине символов при размещении строк текста на странице или в окне. Для этой цели приложение может использовать одну из четырех функций. Две из этих функций позволяют определить ширину символа с межсимвольным интервалом, а две другие — фактическую ширину символа.

Функции `GetCharWidth32` и `GetCharWidthFloat` позволяют определить ширину символа с межсимвольным интервалом. Под шириной символа с межсимвольным интервалом понимается расстояние, на которое курсор на дисплее или печатающая головка принтера должны переместиться перед печатью следующего символа в строке текста. Функция `GetCharWidth32` возвращает ширину символа с межсимвольным интервалом в формате целого числа. Если требуется большая точность, приложение может использовать функцию `GetCharWidthFloat`, позволяющую получить эту величину в формате числа с плавающей запятой.

Функции `GetCharABCWidths` и `GetCharABCWidthsFloat` позволяют определить ширину символа. Функция `GetCharABCWidthsFloat` работает со всеми шрифтами, а функция `GetCharABCWidths` работает только со шрифтами TrueType и OpenType.

Ширина с межсимвольным интервалом может быть больше или меньше фактической ширины символа вследствие того, что символы в строке могут нависать друг над другом.

Кроме информации о ширине каждого символа в строке, приложение также нуждается в информации о ширине и высоте всех строк. Эта информация может быть получена при вызове одной из двух функций: `GetTextExtentPoint32` или `GetTabbedTextExtent`. Если строка не содержит символов табуляции, то приложение может использовать для определения высоты и ширины строки функцию `GetTextExtentPoint32`. В противном случае оно должно вызывать функцию `GetTabbedTextExtent`.

Кроме того, Win32 API содержит функцию `GetTextExtentExPoint`, способную работать с приложениями, обеспечивающими автоматический перенос слов. Эта функция позволяет получить количество символов в определенной строке, попадающее в определенный диапазон.

Некоторые приложения устанавливают различный межстрочный интервал между различными текстовыми строками в зависимости от максимального надстрочного элемента символа и минимального подстрочного элемента используемого в них шрифта. Для этого приложение должно вызвать функцию `GetTextMetrics`, а затем использовать значения переменных `tmAscent` и `tmDescent`, являющихся членами структуры `TEXTMETRIC`.

В некоторых приложениях величины максимального надстрочного и минимального подстрочного элементов отличаются от соответствующих типографских величин данных параметров. В шрифтах TrueType и OpenType типографские величины надстрочного и подстрочного элементов обычно определяются как верхняя точка символа "f" и нижняя точка символа "g" соответственно. Приложение может получить величину типографских надстрочного и подстрочного элементов для шрифтов TrueType или OpenType с помощью вызова функции `GetOutlineTextMetrics` и проверки значений переменных `otmMacAscent` и `otmMacDescent`, являющихся членами структуры `OUTLINETEXTMETRIC`.

Для определения физических размеров шрифтов TrueType или OpenType приложение должно вызвать функцию `GetOutlineTextMetrics`. Для получения подобной информации по любому другому шрифту необходимо вызвать функцию `GetTextMetrics`. Для определения возможностей вывода текста на конкретном устройстве приложение может вызывать функцию `GetDeviceCaps`, возвращающую как физические, так и логические размеры шрифтов.

Логический дюйм представляет собой меру длины, используемую системой для формирования на экране удобочитаемых шрифтов на экране, и приблизительно на 30—40% больший, чем физический дюйм. Использование логических дюймов приводит к отсутствию точного соответствия между текстом, выведенным на экран и на принтер. Разработчики приложений должны понимать, что при одновременном выводе текстовых и графических объектов эти объекты имеют различный масштаб при выводе на экран и на принтер.

Вывод текста

После того как приложение выбрало соответствующий шрифт, установило для него требуемые параметры форматирования и вычислило ширину символов и

размеры строки текста, оно может приступить к выводу данного текста на устройство отображения информации. Для этого может быть использована одна из четырех функций. Функции `DrawText` и `TabbedTextOut` представляют собой методы Windows Manager и хранятся в библиотеке USER.DLL, а остальные две функции являются частью GDI и содержатся в библиотеке GDI.DLL.

При вызове из прикладной программы одной из этих функций операционная система передает запрос графической подсистеме, которая, в свою очередь, передает этот запрос соответствующему драйверу устройства. На уровне драйвера устройства все эти обращения преобразуются в одно или несколько обращений к функциям — членам класса драйвера `ExtTextOut` или `TextOut`. Наиболее быстро задача отображения текста решается с использованием функции `ExtTextOut`, сразу же преобразуется в функцию `ExtTextOut` запрашиваемого устройства. Однако в некоторых случаях приложению выгоднее вызвать одну из оставшихся трех функций. Например, вывод многострочного текста в прямоугольную область на экране наиболее эффективно выполняется функцией `DrawText`. Для создания таблицы, имеющей несколько столбцов форматированного текста, лучше всего воспользоваться функцией `TabbedTextOut`.

Программа вывода текста

Для демонстрации вывода текста было создано приложение Text, которое можно найти в одноименной папке на дискете, прилагаемой к данной книге. Чтобы самостоятельно создать данное приложение:

1. Создайте заготовку многооконного приложения Windows по методике, описанной в *главе 1*, и назовите его Text.
2. Откройте окно **Class View** (Просмотр классов), а в нем раскройте папку **Text**.
3. Щелкните правой кнопкой мыши на папке `CTextView` и выберите в раскрывшемся контекстном меню команду **Properties** (Свойства). Раскроется окно **Properties** (Свойства).
4. В окне **Properties** (Свойства) нажмите кнопку **Messages** (Сообщения). Раскроется список сообщений, обрабатываемых данным классом.
5. В этом списке раскройте список `WM_LBUTTONDOWN` и добавьте в класс функцию обработки данного сообщения `OnLButtonDown`.
6. Раскройте папку `CTextView` и дважды щелкните левой кнопкой мыши на имени функции `OnLButtonDown`. Откроется окно редактирования файла TextView.cpp, а текстовый курсор будет располагаться в заготовке данной функции.
7. Измените функцию `OnLButtonDown` в соответствии с текстом листинга 6.1.

Листинг 6.1. Функция `CTextView::OnLButtonDown`

```
// Вывод координат на экран

void CTextView::OnLButtonDown(UINT nFlags, CPoint point)
{
  CClientDC  clientDC(this);
  LOGFONT  logFont;
  CFont    font;
  CString  Out;

  // Заполнение объекта структуры LOGFONT
  logFont.lfHeight    = 12;
  logFont.lfWidth    = 0;
  logFont.lfEscapement    = 0;
  logFont.lfOrientation  = 0;
  logFont.lfWeight    = FW_NORMAL;
  logFont.lfItalic    = 0;
  logFont.lfUnderline    = 0;
  logFont.lfStrikeOut    = 0;
  logFont.lfCharSet    = ANSI_CHARSET;
  logFont.lfOutPrecision  = OUT_DEFAULT_PRECIS;
  logFont.lfClipPrecision  = CLIP_DEFAULT_PRECIS;
  logFont.lfQuality    = PROOF_QUALITY;
  logFont.lfPitchAndFamily  = VARIABLE_PITCH  |  FF_ROMAN;
  strcpy(logFont.lfFaceName, "MS Sans Serif");

  // Получение координат указателя мыши
  sprintf(Out.GetBuffer(16),"[%d,%d]", point.x, point.y);

  // Очистка буфера
  Out.ReleaseBuffer();

  // Создание шрифта
  font.CreateFontIndirect(&logFont);

  // Ввод шрифта в контекст устройства
  CFont*  oldFont = clientDC.SelectObject(&font);

  // Вывод текста на экран
  clientDC.TextOut(point.x, point.y, Out);
```

```
  // Восстановление старого шрифта в контексте устройства
  clientDC.SelectObject(oldFont);

  CView::OnLButtonDown(nFlags, point);
}
```

8. Нажмите клавишу <F5> и запустите приложение на исполнение. Появится пустое окно.
9. Несколько раз щелкните левой кнопкой мыши в рабочей области окна. Оно примет вид, изображенный на рис. 6.2.

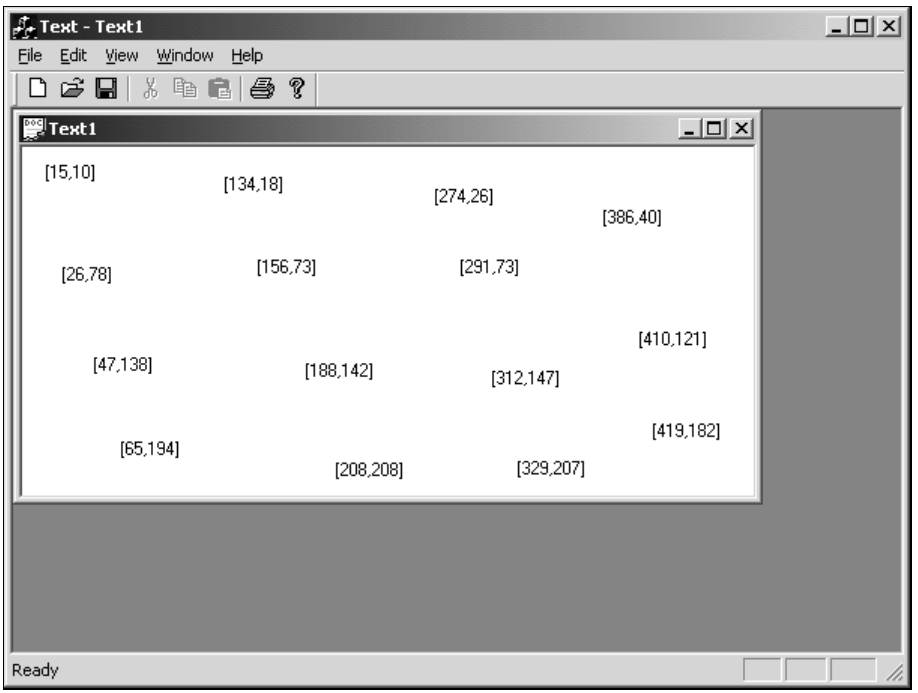


Рис. 6.2. Результат работы приложения Text

Рассмотрим текст функции обработки сообщения `OnLButtonDown`. В данной функции создаются объекты классов контекста устройства, структуры логического шрифта, класса шрифта и строкового класса. После этого производится заполнение переменных в объекте структуры логического шрифта а в объект строкового класса записывается информация о текущем положении курсора мыши. После этого, на основе объекта структуры логического шрифта создается объект класса шрифта, который затем выбирается в контекст устройства с использованием функции `CDC::SelectObject`, причем указатель на объект клас-

са старого шрифта сохраняется при этом в переменной `oldFont`. После этого, с использованием функции `CDC::TextOut`, производится вывод информации, содержащейся в объекте строкового класса, на экран. Последние два оператора данной функции возвращают в контекст устройства старый шрифт и вызывают соответствующую функцию обработки сообщения базового класса.

Перерисовка окна

Попробуйте изменить размер окна в описанном выше приложении. Вся отображавшаяся в нем информация исчезнет. Дело в том, что, как указывалось выше, при изменении размеров или положения окна вызывается функция `OnDraw`, производящая все необходимые для этого операции. Поскольку мы не внесли никаких изменений в текст этой функции, то при ее вызове выполняется процедура, выбираемая по умолчанию, состоящая в очистке содержимого окна.

Чтобы внести необходимые изменения в приложение Text:

1. В окне **Class View** (Просмотр классов) щелкните правой кнопкой мыши на папке `CTextDoc` и выберите в раскрывшемся контекстном меню команду **Go To Definition** (Перейти к заголовку класса). Откроется окно редактирования файла заголовка TextAppDoc.h.
2. В текст описания класса `CTextDoc` в раздел `// Attributes` после модификатора прав доступа `public:` вставьте следующий текст:

```
CUIntArray  aX;
CUIntArray  aY;
```

3. Откройте окно редактирования файла TextView.cpp и вставьте в текст функции `OnLButtonDown` после объявления всех объектов класса `CString` следующий текст:

```
CTextDoc*  lpDoc = GetDocument();


// Запись в массив
lpDoc-> aX.Add(point.x);
lpDoc-> aY.Add(point.y);
```

4. Измените функцию `OnDraw` в соответствии с текстом листинга 6.2.

Листинг 6.2. Функция `CTextView::OnDraw`

```
// Вывод информации на экран в классе CTextView

void CTextView::OnDraw(CDC* pDC)
{
  // Получение указателя на связанный объект документа
  // и проверка его корректности
```

```
  CTextDoc* pDoc = GetDocument();
  ASSERT_VALID(pDoc);

  LOGFONT  logFont;
  CFont    font;
  CString  Out;

  // Заполнение объекта структуры LOGFONT
  logFont.lfHeight    = 12;
  logFont.lfWidth        = 0;
  logFont.lfEscapement    = 0;
  logFont.lfOrientation  = 0;
  logFont.lfWeight    = FW_NORMAL;
  logFont.lfItalic    = 0;
  logFont.lfUnderline    = 0;
  logFont.lfStrikeOut    = 0;
  logFont.lfCharSet    = ANSI_CHARSET;
  logFont.lfOutPrecision  = OUT_DEFAULT_PRECIS;
  logFont.lfClipPrecision  = CLIP_DEFAULT_PRECIS;
  logFont.lfQuality    = PROOF_QUALITY;
  logFont.lfPitchAndFamily  = VARIABLE_PITCH  |  FF_ROMAN;
  strcpy(logFont.lfFaceName, "MS Sans Serif");

  // Создание шрифта
  font.CreateFontIndirect(&logFont);

  // Выбор шрифта в контекст устройства
  CFont*  oldFont = pDC-> SelectObject(&font);

  // Вывод сохраненной информации
  for(int i=0; i < pDoc-> aX.GetSize(); i++){

    sprintf(Out.GetBuffer(16),"[%d,%d]", pDoc-> aX[i], pDoc-> aY[i]);
    Out.ReleaseBuffer();
    pDC-> TextOut(pDoc-> aX[i], pDoc-> aY[i], Out);
  }

  // Возврат старого шрифта в контекст устройства
  pDC-> SelectObject(&oldFont);
}
```

5. Нажмите клавишу <F5> и запустите приложение на исполнение.

На первый взгляд в приложении ничего не изменилось, однако теперь при изменении размеров окна или его перемещении информация не пропадает. Сравнение текстов функций `OnLButtonDown` и `OnDraw` выявляет почти полную их идентичность, за исключением того, что в них используются различные классы контекста устройств и в функции `OnDraw` используются записанные в специальные массивы координаты предыдущих нажатий левой кнопки мыши.

Использование перьев

Работа с перьями намного проще работы со шрифтами, поскольку в данном случае не нужно использовать сложные структуры данных, подобные структуре `LOGFONT`. Перо имеет всего три параметра: стиль начертания линии, толщина и цвет. Для демонстрации принципов работы с перьями было создано приложение Line, которое можно найти в одноименной папке на дискете, прилагаемой к данной книге. Чтобы самостоятельно создать данное приложение:

1. Создайте заготовку многооконного приложения Windows по методике, описанной в *главе 1*, и назовите ее Line.
2. Откройте окно **Class View** (Просмотр класса), а в нем раскройте папку **Line**.
3. Щелкните правой кнопкой мыши на папке `CLineView` и выберите в раскрывшемся контекстном меню команду **Properties** (Свойства). Раскроется окно **Properties** (Свойства).
4. В окне **Properties** (Свойства) нажмите кнопку **Messages** (Сообщения). Раскроется список сообщений, обрабатываемых данным классом.
5. В этом списке раскройте список `WM_LBUTTONDOWN` и добавьте в класс функцию обработки данного сообщения `OnLButtonDown`.
6. Повторите п. 5 для сообщений `WM_LBUTTONUP`, `WM_RBUTTONDOWN`, `WM_RBUTTONDBLCLK` и `WM_MOUSEMOVE`.
7. В окне **Class View** (Просмотр класса) щелкните правой кнопкой мыши на папке `CLineView` и выберите в раскрывшемся контекстном меню команду **Go To Definition** (Перейти к заголовку класса). Откроется окно редактирования файла заголовка LineView.h.
8. В текст описания класса `CLineView` после макроса `DECLARE_MESSAGE_MAP()` вставьте следующий текст:

```
CClientDC*  lpDC;
CPen*    lpPen;
CPen*    oldPen;
int      nWidth;
bool     isDown;
```

9. В окне **Class View** (Просмотр класса) раскройте папку `CLineView` и дважды щелкните левой кнопкой мыши на имени конструктора данного класса. Откроется окно редактирования файла LineView.cpp, а текстовый курсор будет располагаться в заготовке конструктора класса `CLineView`.
10. Измените конструктор класса в соответствии с текстом листинга 6.3.

Листинг 6.3. Конструктор класса `CLineView`

```
// Конструкторы и деструкторы класса CLineView

CLineView::CLineView()
{
  lpDC  = NULL;
  lpPen  = NULL;
  oldPen  = NULL;
  nWidth  = 1;
  isDown  = false;
}
```

11. Измените функции обработки сообщений в соответствии с текстом листинга 6.4.

Листинг 6.4. Функции обработки сообщений класса `CLineView`

```
// Функции обработки сообщений класса CLineView

// Начало рисования линии
void CLineView::OnLButtonDown(UINT nFlags, CPoint point)
{
  lpDC  = new CClientDC(this);  // Получение контекста устройства
  // Создание нового пера
  lpPen  = new CPen(PS_SOLID, nWidth, RGB(0,0,0));
  // Выбор пера в контекст устройства
  oldPen  = lpDC-> SelectObject(lpPen);
  isDown  = true;

  lpDC-> MoveTo(point.x, point.y);  // Установка начальной точки линии
  CView::OnLButtonDown(nFlags, point);
}
```

```
// Завершение рисования линии
void CLineView::OnLButtonUp(UINT nFlags, CPoint point)
{
  lpDC-> SelectObject(oldPen);  // Выбор в контекст старого пера

  delete  lpPen;  // Уничтожение объекта пера
  delete  lpDC;   // Уничтожение объекта контекста устройства
  isDown  = false;
  CView::OnLButtonUp(nFlags, point);
}

// Увеличение толщины линии
void CLineView::OnRButtonDown(UINT nFlags, CPoint point)
{
  nWidth++;
  CView::OnRButtonDown(nFlags, point);
}

// Уменьшение толщины линии
void CLineView::OnRButtonDblClk(UINT nFlags, CPoint point)
{
  if(nWidth > 2)
    nWidth -= 2;
  else
    nWidth = 1;
  CView::OnRButtonDblClk(nFlags, point);
}

// Рисование линии
void CLineView::OnMouseMove(UINT nFlags, CPoint point)
{
  if(isDown)
    lpDC->LineTo(point.x, point.y);
  CView::OnMouseMove(nFlags, point);
}
```

12. Нажмите клавишу <F5> и запустите приложение на исполнение. Появится пустое окно.
13. Нажмите левую кнопку мыши в рабочей области окна и, удерживая ее нажатой, переместите мышь. В окне появится линия, соответствующая траектории перемещения указателя мыши.

14. Отпустите левую кнопку мыши, нажмите правую и повторите операцию. Линия станет толще.
15. Дважды щелкните правой кнопкой мыши и нарисуйте еще одну линию. Она станет тоньше. Результат подобных операций приведен на рис. 6.3.

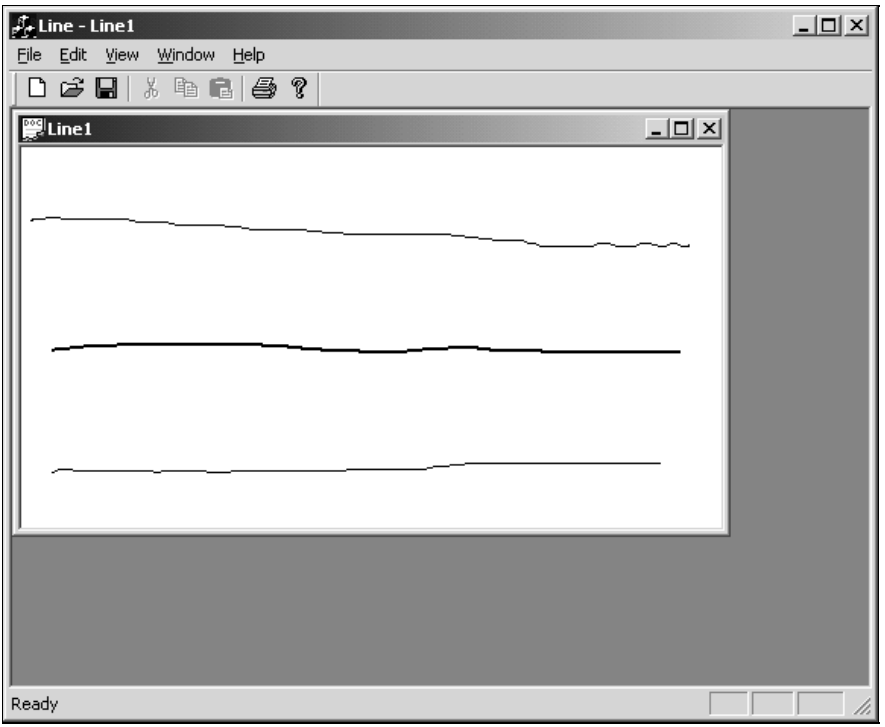


Рис. 6.3. Результат работы приложения Line

При нажатии левой кнопки мыши в рабочей области окна вызывается функция обработки сообщения `OnLButtonDown`, в которой создаются новые объекты классов контекста устройства и пера, и указатели на них присваиваются переменным — членам класса представления. Объект класса пера выбирается в контекст устройства, а старый объект данного класса сохраняется в переменной — члене класса представления. После этого логической переменной присваивается значение, указывающее на то, что нажата левая кнопка мыши и текущая позиция перемещается в точку, соответствующую текущему положению курсора мыши.

При перемещении мыши в рабочей области окна вызывается функция обработки сообщения `OnMouseMove`, в которой производится проверка того, нажата ли в данный момент левая кнопка мыши, и в случае положительного исхода проверки на экран выводится линия от текущей позиции до текущего положения кур-

сора, причем текущая позиция перемещается при этом в конец нарисованной линии.

При отпускании левой кнопки мыши в рабочей области окна вызывается функция обработки сообщения `OnLButtonUp`, в которой в контекст устройства выбирается сохраненный объект класса пера и уничтожаются объекты класса контекста устройства и класса пера. Логической переменной присваивается значение, указывающее на то, что левая кнопка мыши отпущена.

При нажатии правой кнопки мыши в рабочей области окна вызывается функция обработки сообщения `OnRButtonDown`, производящая увеличение параметра ширины линии на один дискрет.

При двойном нажатии правой кнопки мыши в рабочей области окна вызывается функция обработки сообщения `OnRButtonDblClk`. Эта функция имеет более сложную структуру, чем функция `OnRButtonDown`. Это связано, прежде всего, с тем, что величина, определяющая ширину линии, должна быть положительной, поэтому прежде чем уменьшать, ее следует проверить, не находится ли она уже на нижнем пределе. Вообще-то нижним пределом ширины линии является нулевая величина, но она в данном варианте вызова конструктора класса `CPen` эквивалентна единичной ширине, что может вызвать вопросы об отсутствии видимых изменений при щелчке мыши при нулевом значении ширины линии. Поэтому в качестве нижнего предела была выбрана единичная ширина линии.

С другой стороны, необходимо помнить, что двойной щелчок кнопкой мыши состоит из двух одиночных щелчков, и система не может не послать сообщение о первом нажатии кнопки на соответствующую обработку, поскольку она не обладает телепатическими способностями и не может знать заранее: собирается ли пользователь сделать одиночный или двойной щелчок или он будет обрабатывать сообщения о перемещении мыши с нажатой кнопкой. Все это приводит к тому, что в нашем случае при двойном щелчке правой кнопкой мыши сначала вызывается функция обработки сообщения `OnRButtonDown`, производящая увеличение параметра ширины линии на один дискрет, и только потом функция обработки сообщения `OnRButtonDblClk`, которая должна уменьшить эту величину на один дискрет. Поэтому в данной функции ширина линии уменьшается на два дискрета. Если ширина линии уже находилась на нижнем пределе, то данная функция должна только восстановить исходное значение.

Работа с кистью

Если перья используются для рисования линий и фигур, то кисти предназначены для заполнения замкнутых пространств определенными цветами или текстурами. Кисть может заполнять область равномерным фоном любого цвета, использовать стандартные трафареты (pattern) или определенные пользователем битовые поля.

Для демонстрации принципов работы с кистью было создано приложение Brush, которое можно найти в одноименной папке на дискете, поставляемой с этой книгой. Чтобы самостоятельно создать данное приложение:

1. Создайте заготовку многооконного приложения Windows по методике, описанной в *главе 1*, и назовите ее Brush.
2. Откройте окно **Class View** (Просмотр классов), а в нем раскройте папку **Brush**.
3. Щелкните правой кнопкой мыши на папке `CBrushView` и выберите в раскрывшемся контекстном меню команду **Properties** (Свойства). Раскроется окно **Properties** (Свойства).
4. В окне **Properties** (Свойства) нажмите кнопку **Messages** (Сообщения). Раскроется список сообщений, обрабатываемых данным классом.
5. В этом списке раскройте список `WM_LBUTTONDOWN` и добавьте в класс функцию обработки данного сообщения `OnLButtonDown`.
6. Повторите п. 5 для сообщения `WM_RBUTTONDOWN`.
7. В окне **Class View** (Просмотр класса) щелкните правой кнопкой мыши на папке `CBrushView` и выберите в раскрывшемся контекстном меню команду **Go To Definition** (Перейти к заголовку класса). Откроется окно редактирования файла заголовка BrushView.h.
8. В текст описания класса `CBrushView` после макроса `DECLARE_MESSAGE_MAP()` вставьте следующий текст:

```
int    nBrushStyle;
```

9. В окне **Class View** (Просмотр класса) раскройте папку `CBrushView` и дважды щелкните левой кнопкой мыши на имени конструктора данного класса. Откроется окно редактирования файла BrushView.cpp, а текстовый курсор будет располагаться в заготовке конструктора класса `CBrushView`.
10. Измените конструктор класса в соответствии с приведенным ниже текстом.

```
// Конструкторы и деструкторы класса CBrushView
   CBrushView::CBrushView()
   {
     nBrushStyle    = 0;
   }
```

11. Измените текст функций обработки сообщений в соответствии с текстом листинга 6.5.

Листинг 6.5. Функции обработки сообщений класса `CBrushView`

```
// Функции обработки сообщений класса CBrushView
// Вывод образца кисти на экран
void CBrushView::OnLButtonDown(UINT nFlags, CPoint point)
```

```
{
  CClientDC  dc(this);            // Получение контекста устройства
  CBrush*  lpBrush;
  CPen    pen(PS_SOLID, 0, RGB(0,0,255));  // Создание нового пера
  // Выбор пера в контекст устройства
  CPen*    oldPen = dc.SelectObject(&pen);

  // Создание кисти заданного стиля
  if(nBrushStyle)
    lpBrush = new CBrush(nBrushStyle — 1, RGB(0,0,0));
  else
    lpBrush = new CBrush(RGB(255,0,0));

  // Выбор кисти в контекст устройства
  CBrush*    oldBrush = dc.SelectObject(lpBrush);

  // Вывод образца
  dc.Rectangle(point.x, point.y, point.x + 40, point.y + 40);
  dc.SelectObject(oldBrush);
  dc.SelectObject(oldPen);

  // Уничтожение объекта кисти
  delete  lpBrush;
  CView::OnLButtonDown(nFlags, point);
}
// Выбор стиля кисти
void CBrushAppView::OnRButtonDown(UINT nFlags, CPoint point)
{
  nBrushStyle++;
  nBrushStyle  %= 7;
  CView::OnRButtonDown(nFlags, point);
}
```

12. Нажмите клавишу <F5> и запустите приложение на исполнение. Появится пустое окно.
13. Попеременно нажимайте левую и правую кнопки мыши в рабочей области окна. В окне будут появляться квадраты, содержащие образцы стандартных трафаретов и сплошной фон, как это показано на рис. 6.4.

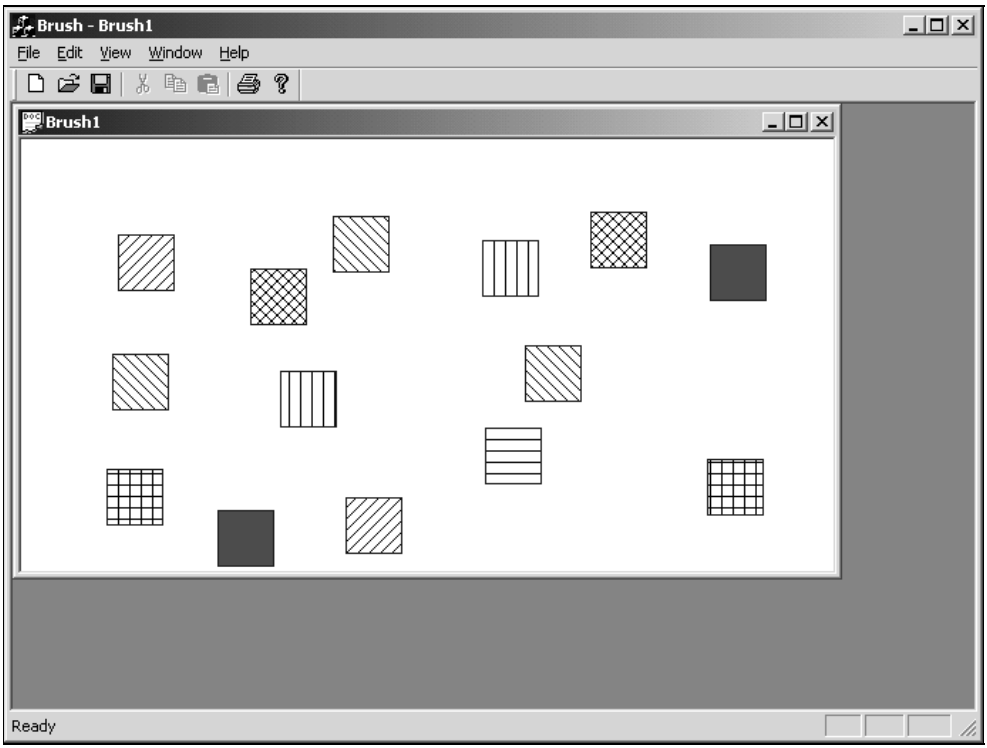


Рис. 6.4. Результат работы приложения Brush

При нажатии левой кнопки мыши в рабочей области окна вызывается функция обработки сообщения `OnLButtonDown`, в которой создаются новые объекты классов контекста устройства и пера. Объект класса пера выбирается в контекст устройства, а старый объект данного класса сохраняется. После этого в зависимости от значения переменной, характеризующей стиль кисти, создается новый объект кисти с соответствующим стилем. Если стиль равен нулю, то создается кисть с равномерным красным фоном. В противном случае создаются кисти с одним из стандартных трафаретов. Поскольку нумерация трафаретов начинается с нуля, то при передаче его номера в конструктор кисти из него вычитается единица. Созданная кисть выбирается в контекст устройства, а старый объект кисти сохраняется. После этого в рабочем поле окна рисуется прямоугольник, границы которого нарисованы выбранным в контекст пером, а внутренняя поверхность заполнена выбранной в контекст устройства кистью. Координаты верхнего левого угла данного прямоугольника соответствуют текущему положению курсора мыши.

При нажатии правой кнопки мыши в рабочей области окна вызывается функция обработки сообщения `OnRButtonDown`, производящая циклическое изменение параметра стиля кисти.

Использование диалогового окна для настройки параметров

В предыдущих примерах изменение параметров отображения информации на экран производилось путем нажатия правой кнопки мыши. В реальном приложении такой способ задачи параметров недопустим, поскольку в нем имеется слишком много параметров, принимающих значения в достаточно широком диапазоне, чтобы все изменения этих параметров производить одной-единственной кнопкой.

В реальных приложениях для изменения значений практически всех параметров приложения используются диалоговые окна. В *главе 3* мы уже рассмотрели этот вид окон, но для его исследования использовались специальные диалоговые приложения, в которых диалоговое окно вызывалось непосредственно при вызове приложения и являлось его главным окном. В большинстве случаев диалоговые окна вызываются из функций обработки соответствующих сообщений. Подробно процесс создания диалогового окна рассмотрен в разделе описания класса `Cdialog`, размещенного на дискете. В данном разделе на базе приложения Brush будет создан простейший пример использования диалогового окна для изменения параметров кисти.

Чтобы включить в данное приложение работу с диалоговым окном:

1. Выберите команду **File | Open Solution** (Файл | Открыть приложение). Появится диалоговое окно **Open Solution** (Открыть приложение), изображенное на рис. 6.5.

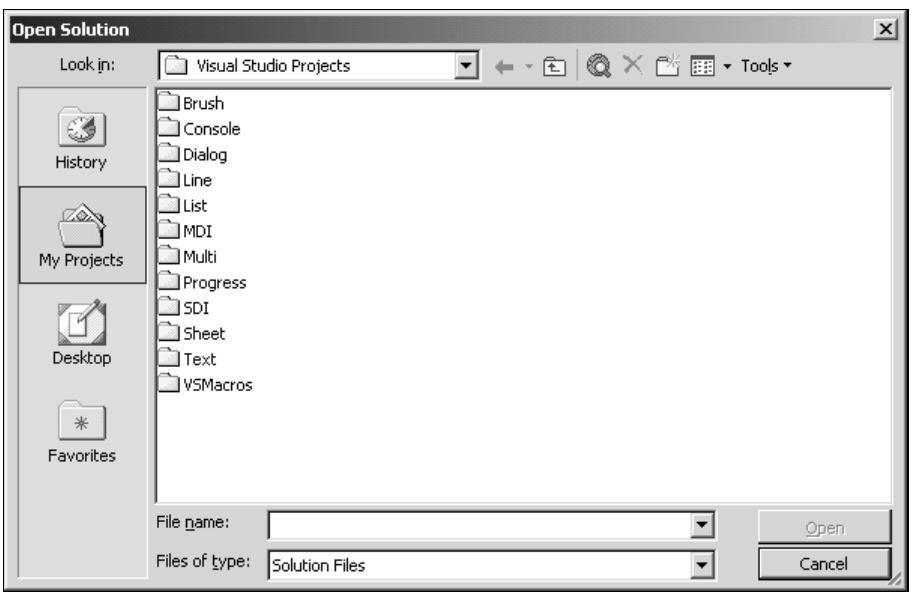


Рис. 6.5. Диалоговое окно **Open Solution**

2. Раскройте папку **Brush** (Кисть) и дважды щелкните левой кнопкой мыши по значку **Brush** (Кисть). Откроется соответствующий проект.
3. Раскройте окно **Resourse View** (Просмотр ресурсов), щелкните правой кнопкой мыши на папке **Dialog** Диалог) и выберите в появившемся контекстном меню команду **Insert Dialog** (Добавить ресурс диалогового окна). Раскроется окно редактирования диалогового ресурса, в котором будет находиться заготовка диалогового окна.
4. В окне **Resourse View** (Просмотр ресурсов) раскройте папку **Dialog** (Диалог), выделите в ней идентификатор ресурса `IDD_DIALOG1` и нажмите клавишу <F4>. Откроется окно **Properties** (Свойства).
5. Введите в текстовое поле раскрывающегося списка **ID** (Идентификатор ресурса) открывшегося окна **Properties** (Свойства) идентификатор ресурса `IDD_BRUSH_DIALOG`.
6. В раскрывающемся списке **Language** (Язык) того же диалогового окна выделите строку **Russian** (Русский) (если она уже не отображается в текстовом поле данного диалогового окна).
7. Щелкните левой кнопкой мыши в области диалогового окна и введите в текстовое поле **Caption** (Заголовок) окна **Properties** (Свойства) заголовок "Настройка кисти".
8. Закройте окно **Output** (Окно вывода) и растяните заготовку диалогового окна.
9. В окне **Toolbox** (Инструментарий) выберите элемент управления **Group Box** (Рамка группы), перетащите его в левую часть диалогового окна и растяните на все свободное пространство по высоте.
10. Введите в текстовое поле **Caption** (Заголовок) окна **Properties** (Свойства) заголовок "Трафареты кисти".
11. В окне **Toolbox** (Инструментарий) выберите элемент управления **Radio Button** (Переключатель) и перетащите его в верхнюю часть групповой рамки.
12. Введите в текстовое поле **Caption** (Заголовок) окна **Properties** (Свойства) заголовок "Сплошной фон".
13. В раскрывающемся списке **Group** (Группа), расположенном в группе **Misc** (Общие свойства) окна **Properties** (Свойства), выделите строку **True** (Истина).
14. Повторите п.п. 11 и 12 для шести других переключателей, сопоставляя им заголовки: "Горизонтальный", "Вертикальный", "Диагональный, наклон влево", "Диагональный, наклон вправо", "Прямая клетка" и "Косая клетка". При необходимости увеличьте горизонтальные и вертикальные размеры заготовки диалогового окна и групповой рамки. Заготовка диалогового окна примет вид, изображенный на рис. 6.6.
15. Щелкните правой кнопкой мыши на заготовке диалогового окна и выберите в появившемся контекстном меню команду **Add Class** (Добавить класс). Появится диалоговое окно **Add Class - Brush** (Добавить класс), изображенное на рис. 6.7.

Рис. 6.6. Заготовка диалогового окна

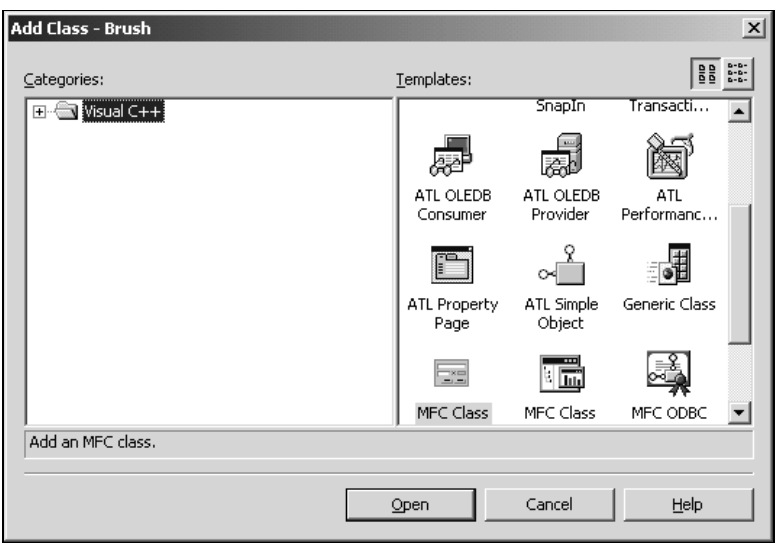


Рис. 6.7. Диалоговое окно **Add Class - Brush**

16. В окне списка **Templates** (Шаблоны) выделите значок **MFC Class** (Класс MFC) и нажмите кнопку **Open** (Открыть). Появится диалоговое окно **MFC Class Wizard** (Мастер создания класса MFC), изображенное на рис. 6.8.

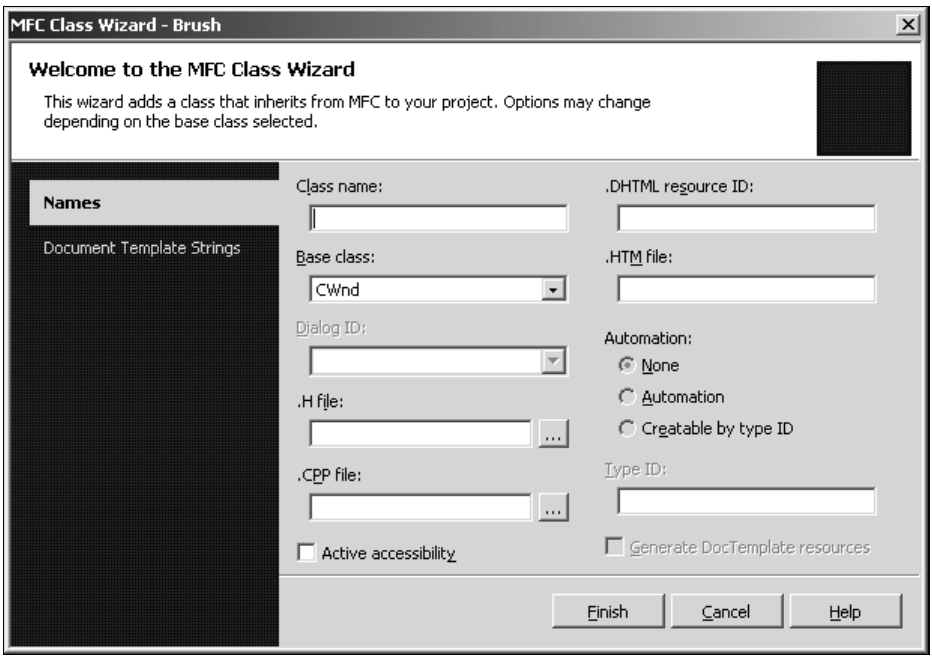


Рис. 6.8. Диалоговое окно **MFC Class Wizard - Brush**

17. В текстовое поле **Class name** (Имя класса) введите имя класса `CBrushDialog`.
18. В раскрывающемся списке **Base class** (Базовый класс) выберите имя базового класса `CDialog`.
19. В ставшем после этого доступным раскрывающемся списке **Dialog ID** (Идентификатор ресурса диалогового окна) выберите идентификатор диалогового окна `IDD_BRUSH_DIALOG` (другого выбора у вас все равно не будет).
20. Оставьте выбранное по умолчанию имя файла BrushDialog.cpp и нажмите кнопку **Finish** (Готово). Окно **MFC Class Wizard - Brush** (Мастер создания класса MFC) закроется.
21. Откройте окно **Class View** (Просмотр класса), а в нем — папку **Brush**.
22. Щелкните правой кнопкой мыши на папке `CBrushDialog` и выберите в появившемся меню команду **Add | Add Variable** (Добавить | Добавить переменную). Появится диалоговое окно **Add Member Variable Wizard - Brush** (Мастер добавления переменной в класс), изображенное на рис. 6.9.

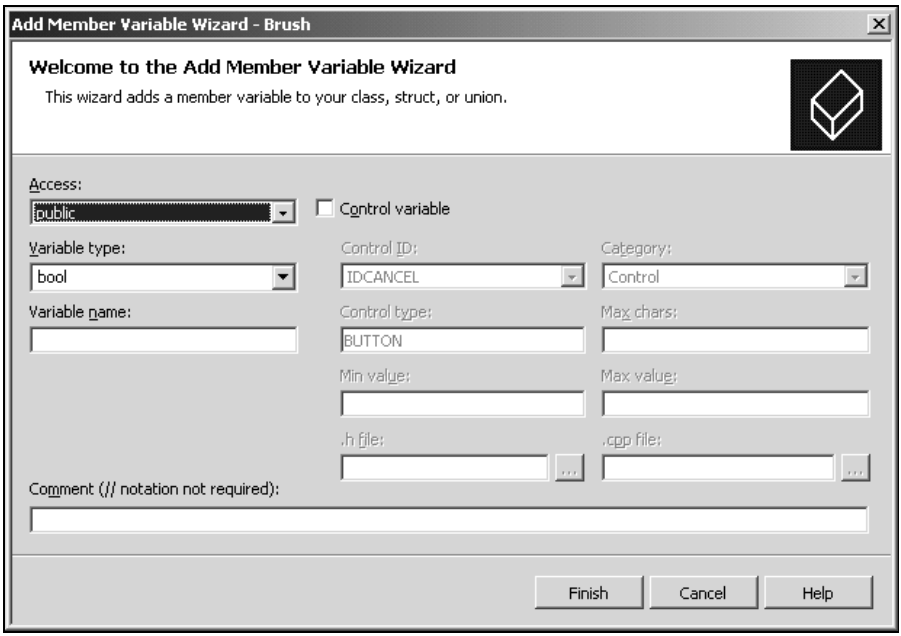


Рис. 6.9. Диалоговое окно **Add Member Variable Wizard - Brush**

23. Установите флажок **Control variable** (Связь с элементом управления).
24. В ставшем после этого доступным раскрывающемся списке **Control ID** (Идентификатор элемента управления) выделите идентификатор ресурса `IDC_RADIO1`.
25. В раскрывающемся списке **Category** (Категория) выделите строку **Value** (Переменная).
26. Введите в текстовое поле раскрывающегося списка **Variable type** (Тип переменной) тип переменной `int`.
27. Введите в текстовое поле **Variable name** (Имя переменной) имя переменой `m_nBrush` и нажмите кнопку **Finish** (Готово).
28. Откройте окно редактирования файла BrushView.cpp и после строки `#include "BrushView.h"` вставьте в него строку `#include "BrushDialog.h"`.
29. Измените функцию `OnRButtonDown` в соответствии с текстом листинга 6.6.

Листинг 6.6. Функция `CBrushView::OnRButtonDown`

```
// Выбор стиля кисти

void CBrushView::OnRButtonDown(UINT nFlags, CPoint point)
{
```

```
//  nBrushStyle++;
//  nBrushStyle  %= 7;

  CBrushDialog  dlg;  // Создание объекта класса диалогового окна
  dlg.m_nBrush  = nBrushStyle;  // Указание текущего стиля
  // Вывод диалогового окна и сохранение выбора пользователя
  if(dlg.DoModal() == IDOK)
    nBrushStyle  = dlg.m_nBrush;
  CView::OnRButtonDown(nFlags, point);
}
```

30. Запустите приложение на исполнение.
31. Щелкните правой кнопкой мыши в рабочей области окна. Появится диалоговое окно **Настройка кисти**, показанное на рис. 6.10.

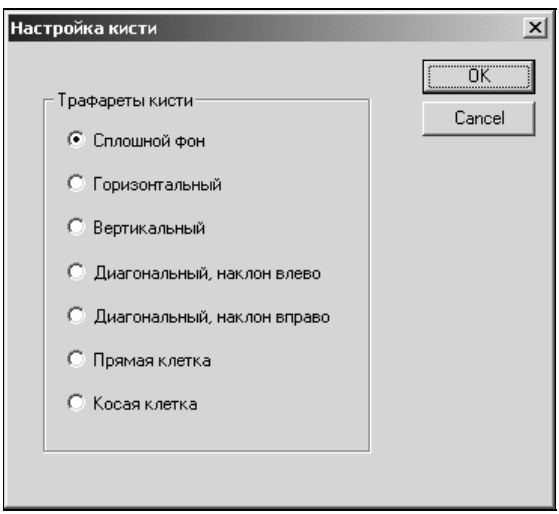


Рис. 6.10. Диалоговое окно **Настройка кисти**

32. Установите переключатель **Трафареты кисти** в положение **Косая клетка** и нажмите кнопку **OK**. Диалоговое окно **Настройка кисти** закроется.
33. Щелкните левой кнопкой мыши в рабочей области окна. На месте щелчка появится квадрат, заполненный косой клеткой.
34. Повторите п.п. 31—33 для других трафаретов кисти.

Первый оператор функции `OnRButtonDown` создает объект класса `CBrushDialog`. Переменной данного объекта `m_nBrush` присваивается текущее значение пере-

менной `nBrushStyle` и вызывается функция `CDialog::DoModal`, осуществляющая весь обмен информацией с объектом диалогового окна и возвращающая функции `OnRButtonDown` управление только после закрытия диалогового окна. Если диалоговое окно было закрыто нажатием кнопки **OK**, переменной `nBrushStyle` присваивается значение переменной — члена класса диалога `m_nBrush`.

Присваивание переменной `m_nBrush` значения переменной `nBrushStyle` позволяет установить переключатель диалогового окна в положение, соответствующее текущей установке трафарета кисти. Обратное присваивание позволяет передать и сохранить в классе представления новую установку трафарета кисти.

Работа с битовыми образами

Термин *битовый образ* иногда неправильно трактуется некоторыми программистами. Многие из них понимают под ним любое изображение, которое может выводиться на экран. Хотя это определение имеет право на существование, необходимо помнить, что существуют две разновидности битовых образов, из которых только один подходит под данное определение. Первая разновидность называется *аппаратно-независимыми битовыми образами* (device-independent bitmaps или DIB). Эти битовые образы хранятся в файлах с расширением bmp и именно эту разновидность имеют в виду большинство пользователей, когда говорят о битовых образах. Второй разновидностью битовых образов являются *аппаратно-зависимые битовые образы* (device-dependent bitmap или DDB). Эти битовые образы существуют только в памяти компьютера и представляют собой изображение, создаваемое приложением Windows для вывода на экран.

Аппаратно-зависимые битовые образы

Аппаратно-независимые битовые образы являются таковыми, поскольку содержат информацию о цвете, необходимую для их вывода на любое устройство. Аппаратно-зависимые битовые образы не содержат таблиц цветов и поэтому привязаны к конкретному устройству. Эти битовые образы создаются в памяти компьютера непосредственно перед выводом изображения на экран и удаляются из памяти вместе с создавшим их приложением.

Этот тип битовых образов обычно используется для выполнения чисто утилитарных задач. Например, для хранения образца текстуры, используемой для заполнения некоторого пространства. Другой, достаточно распространенной, областью применения аппаратно-зависимых битовых образов является копирование части экрана с целью переноса его в новое место. При этом выделенная часть экрана копируется в память в формате битового образа, а затем этот битовый образ выводится в указанном месте экрана.

Для демонстрации принципов работы с аппаратно-зависимыми битовыми образами было создано приложение DDB, которое можно найти в одноименной

папке на дискете, прилагаемой к данной книге. Чтобы самостоятельно создать данное приложение:

1. Создайте приложение с именем DDB по методике, описанной в *главе 1*. Оставьте все установки по умолчанию без изменения.
2. В окне **Class View** (Просмотр класса) раскройте папку **DDB**.
3. Щелкните правой кнопкой мыши на папке `CDDBView` и выберите в раскрывшемся контекстном меню команду **Add | Add Variable** (Добавить | Добавить переменную). Появится диалоговое окно **Add Member Variable Wizard** (Мастер добавления переменной в класс).
4. В текстовое поле **Variable Type** (Тип переменой) введите тип переменной `CRect`, а в текстовое поле **Variable name** (Имя переменной) введите имя `m_Rect` и нажмите кнопку **Finish** (Готово).
5. Повторите п.п. 3 и 4 для включения в этот класс переменной `was_copied`, имеющей тип `bool`.
6. Щелкните правой кнопкой мыши на папке `CDDBView` и выберите в раскрывшемся контекстном меню команду **Properties** (Свойства).
7. В раскрывшемся окне **Properties** (Свойства) нажмите на кнопку **Messages** (Сообщения). Раскроется список сообщений, обрабатываемых классом `CDDBView`.
8. Добавьте в этот список функцию обработки сообщения `WM_LBUTTONDBLCLK`.
9. Щелкните правой кнопкой мыши на папке `CDDBView` и выберите в раскрывшемся контекстном меню команду **Go To Definition** (Перейти к заголовку класса). Раскроется окно редактирования файла заголовка DDBView.h.
10. Под строкой `bool was_copied;` вставьте строку

    ```
    BYTE m_Buffer[32768];
    ```

11. В окне **Class View** (Просмотр класса) раскройте папку `CDDBView`, щелкните левой кнопкой мыши на имени функции обработки сообщения `OnLButtonDblClk` и выберите в раскрывшемся контекстном меню команду **Go To Definition** (Перейти к заголовку класса). Раскроется окно редактирования файла реализации DDBView.cpp, а текстовый курсор будет располагаться в заготовке данной функции.
12. Измените функцию `OnLButtonDblClk` в соответствии с текстом листинга 6.7.

Листинг 6.7. Функция `CDDBView::OnLButtonDblClk`

```
// Функции обработки сообщений класса CDDBView


// Сохранение и вывод битовых образов
void CDDBView::OnLButtonDblClk(UINT nFlags, CPoint point)
```

```
{
  CClientDC  pDC(this);  // Получение контекста устройства
  CDC    memDC;          // Создание контекста устройства памяти
  CBitmap  m_Bitmap;     // Создание объекта битового образа

  int      wd  = m_Rect.Width() >> 2;

  // Создание совместимого битового образа и контекста устройства памяти
  m_Bitmap.CreateCompatibleBitmap(&pDC, wd, m_Rect.Height());
  memDC.CreateCompatibleDC(&pDC);

  // Сохранение старого битового образа
  CBitmap*  oldBitmap = memDC.SelectObject(&m_Bitmap);
  if(m_Rect.PtInRect(point))  // Копирование экрана
  {
    int  nRect = (point.x — m_Rect.left)/wd;
    memDC.BitBlt(0, 0, wd, 40, &pDC, m_Rect.left + nRect*wd, 10, SRCCOPY);
    m_Bitmap.GetBitmapBits(32768, m_Buffer);
    was_copied = true;
  }
  else  // Вывод на экран
  {
    CRect  rect;
    GetClientRect(rect);
    if(was_copied && (point.x < (rect.right — wd)) && (point.y <
(rect.bottom — 40))){
      m_Bitmap.SetBitmapBits(32768, m_Buffer);
      pDC.BitBlt(point.x, point.y, wd, 40, &memDC, 0, 0, SRCCOPY);
    }
  }
  // Выбор старого битового образа
  memDC.SelectObject(&oldBitmap);
  CView::OnLButtonDblClk(nFlags, point);
}
```

13. Измените функцию `OnDraw` в соответствии с текстом листинга 6.8.

Листинг 6.8. Функция `CDDBView::OnDraw`

```
// Вывод информации на экран в классе CDDBView
void CDDBView::OnDraw(CDC* pDC)
```

```
{
  // Получение указателя на связанный объект документа
  // и проверка его корректности
  CDDBDoc* pDoc = GetDocument();
  ASSERT_VALID(pDoc);

  // Получение координат рабочей области окна
  CRect  rect;

  GetClientRect(rect);
  if((rect.Width() > 24) && (rect.Height() > 80))  // Места хватит
  {
    // Вывод исходного изображения
    m_Rect  = CRect(10, 10, rect.right — 10, 50);
    int  wd  = m_Rect.Width() >> 2;

    rect = m_Rect;
    rect.right = rect.left + wd;

    pDC->FillRect(rect, &CBrush(RGB(0, 0, 0)));

    rect.left  = rect.right;
    rect.right  += wd;

    pDC->FillRect(rect, &CBrush(RGB(255, 0, 0)));

    rect.left  = rect.right;
    rect.right  += wd;
    pDC->FillRect(rect, &CBrush(RGB(0, 255, 0)));

    rect.left  = rect.right;
    rect.right  += wd;

    pDC->FillRect(rect, &CBrush(RGB(0, 0, 255)));
  }
  was_copied  = false;
}
```

14. Нажмите клавишу <F5> и запустите приложение на исполнение. Окно приложения будет иметь вид, изображенный на рис. 6.11.

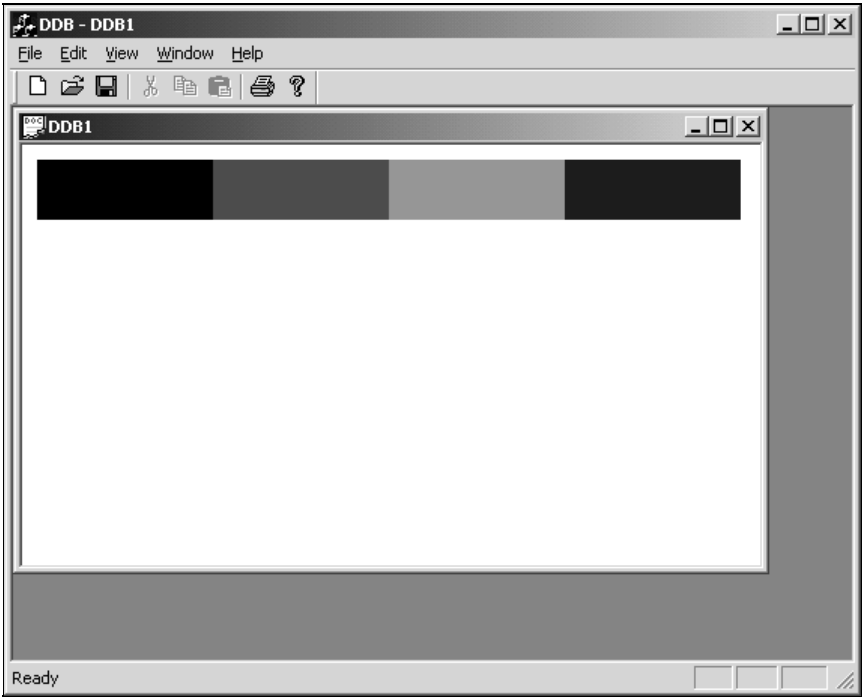


Рис. 6.11. Исходное окно приложения

15. Дважды щелкните левой кнопкой мыши по одному из прямоугольников с основными цветами, расположенных в верхней части окна документа. Переместите курсор мыши на свободное место под этими прямоугольниками и дважды щелкните левой кнопкой мыши. На указанном месте появится прямоугольник выбранного цвета.
16. Повторите эту операцию с другими цветами. Результат подобных действий приведен на рис. 6.12.
17. Закройте приложение.

Для демонстрации принципов работы с аппаратно-зависимыми битовыми образами в окне документа выводятся четыре прямоугольника, окрашенные в четыре основных цвета (чистый тон каждого из основных цветов и отсутствие цветов или черный цвет). Поскольку эти прямоугольники не должны выходить за пределы окна при его перерисовке, они генерируются непосредственно в функции `OnDraw`. После получения указателя на объект класса документа, связанный с данным объектом класса представления, и проверки его допустимости, производится вызов функции `CWnd::GetClientRect`, копирующей пользовательские координаты рабочей области окна, связанного с объектом класса `CWnd`, в структуру, на которую указывает ее аргумент. Если размеры окна позволяют разместить в нем четыре прямоугольника и при этом в окне остается еще доста-

точно места для их копирования, в окне выводятся эти четыре прямоугольника. Для этого вычисляются координаты области размещения прямоугольников и сохраняются в переменной `m_Rect`. Кроме того, определяется значение переменной `wd`, в которой хранится ширина прямоугольников. После этого производится вывод прямоугольников на экран с использованием функции `CDC::FillRect`, заполняющей прямоугольник указанной кистью. Перед выходом из функции `OnDraw` независимо от того, были в ней выведены цветные прямоугольники или нет, переменной `was_copied` присваивается значение `FALSE`, говорящее о том, что информация, хранящаяся в буфере, не может использоваться для формирования битового образа.

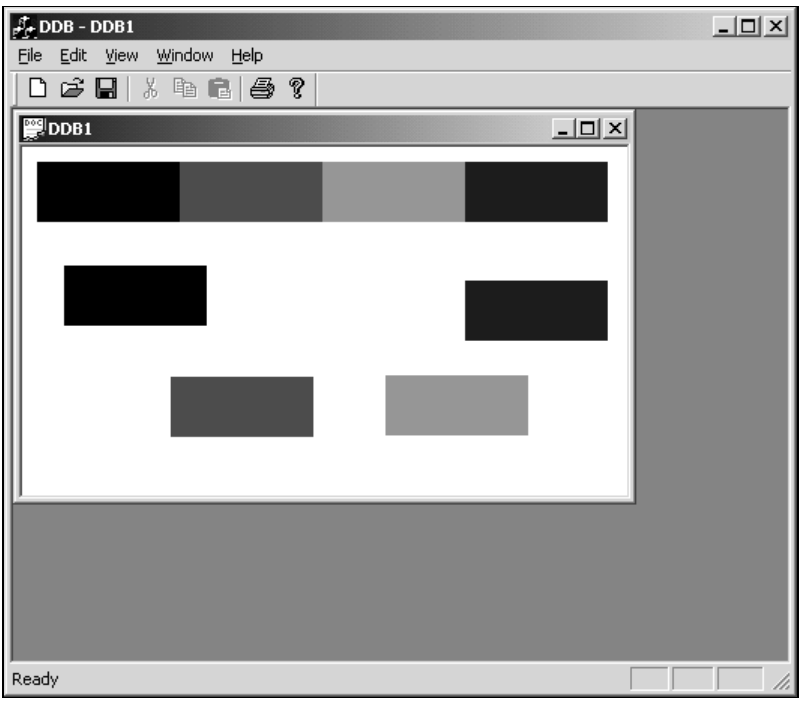


Рис. 6.12. Окно приложения после работы с ним

Непосредственная работа с битовым образом производится в функции `OnLButtonDblClk`. Для этого в ней создаются объекты классов `CDC` и `CBitmap`. Переменная `memDC`, являющаяся объектом класса `CDC`, служит для хранения контекста устройства памяти, представляющего собой область памяти, эмулирующую виртуальный дисплей. Контекст устройства памяти обычно используется для сборки изображений перед их выводом на экран.

Созданный объект класса `CBitmap` инициализируется функцией `CBitmap::CreateCompatibleBitmap`, первым аргументом которой является указа-

тель на объект класса контекста устройства, вторым аргументом — ширина битового образа, а третьим — его высота. Использование данной функции устанавливает для инициализируемого битового образа такой формат, который позволит выбирать его в указанный контекст устройства в качестве текущего битового образа. Объект класса `CDC` инициализируется функцией `CDC::CreateCompatibleDC`, устанавливающей контекст устройства памяти и делающей его совместимым с контекстом устройства, объект класса которого задается ей в качестве аргумента. Инициализированные объекты классов используются для выбора в контекст устройства памяти пользовательского объекта класса `CBitmap`. Указатель на старый объект класса `CBitmap` сохраняется в специальной переменной. Для осуществления этой операции вызывается функция `CDC::SelectObject`.

После проведения подготовительных операций производится проверка того, находится ли указатель мыши в прямоугольнике с эталонными цветами. Эта проверка производится с использованием функции `CRect::PtInRect`. Если указатель мыши при двойном щелчке находится в пределах прямоугольника, производится вычисление номера цветного прямоугольника, в котором находится указатель мыши и вызывается функция `CDC::BitBlt` для объекта класса контекста устройства памяти. Данная функция имеет восемь аргументов. Первые два аргумента определяют координаты левого верхнего угла области вывода. Следующие два аргумента определяют размеры этой области. Пятый аргумент является указателем на объект класса контекста устройства, из которого производится копирование информации. Следующие два аргумента определяют координаты левого верхнего угла копируемого изображения, а последний аргумент определяет режим копирования информации в область вывода.

После этого вызывается функция `CBitmap::GetBitmapBits`, копирующая битовый образ из объекта класса `CBitmap` в буфер, на который указывает ее второй аргумент. Ее первый аргумент определяет число копируемых байтов. Поскольку этот битовый образ будет впоследствии копироваться в область вывода, имеющую те же размеры и формат, что и исходная область, информация может быть скопирована с запасом, поскольку для заполнения области вывода будет использоваться только нужная для этого информация. Единственное требование при этом, чтобы вся необходимая информация поместилась в этом буфере. Используемое в данном случае значение удовлетворяет требованиям, предъявляемым при работе в большинстве реальных режимов отображения информации на дисплее. Последней операцией в данной ветви условного оператора является присваивание переменной `was_copied` значения `TRUE`, свидетельствующего о том, что буфер заполнен и его можно использовать для вывода битового образа на экран.

Если двойной щелчок левой кнопки мыши был произведен за пределами прямоугольника с эталонными цветами, с помощью функции `CWnd::GetClientRect` определяются координаты рабочей области окна и производится проверка: содержит ли буфер информацию о битовом образе и не выйдет ли она за пределы рабочей области окна. В том случае, если эти условия выполняются, производится вывод информации в окно. Для этого вызывается функция

`CBitmap::SetBitmapBits`, устанавливающая биты в текущем битовом образе контекста устройства памяти в соответствии со значениями, содержащимися в ее втором аргументе. Затем вызывается функция `CDC::BitBlt`, копирующая битовый образ из объекта класса контекста устройства памяти в контекст устройства дисплея.

По завершении всех операций, независимо от того, где был произведен двойной щелчок левой кнопкой мыши, в контекст устройства памяти выбирается его старый битовый образ, освобождая при этом выбранный в него до этого битовый образ.

В завершение этого раздела следует сделать замечание, не относящееся непосредственно к рассматриваемой теме. Вы, наверно, уже обратили внимание, что скалярные члены класса `CDDBView` были включены в него с использованием мастера, а массив `m_Buffer` был включен непосредственно в файл заголовка. Это связано с любовью корпорации Microsoft устраивать своим пользователям мелкие пакости. В Visual C++ 6.0 диалоговое окно **Add Member Variable** (Добавить переменную в класс) без проблем позволяло включать в класс массивы. Однако чтобы жизнь малиной не казалась, в Visual C++ 7.0 эта возможность была изъята.

Аппаратно-независимые битовые образы

Как уже говорилось выше, аппаратно-зависимые битовые образы, как следует из их названия, жестко привязаны к устройству, с которого считывается или на который выводится графическая информация. Однако что же делать в том случае, если вам необходимо вывести некоторое растровое изображение в создаваемом вами приложении. Вы не можете заранее знать в каком режиме будет работать дисплей пользователя вашего приложения. Поэтому вы не можете использовать аппаратно-зависимые битовые образы. Наилучшим выходом из этого положения является использование аппаратно-независимых битовых образов. Эти битовые образы позволяют определить используемую в них палитру, которая стандартными средствами преобразуется к текущей палитре графического адаптера.

Для демонстрации принципов работы с аппаратно-независимыми битовыми образами было создано приложение DIB, которое можно найти в одноименной папке на дискете, прилагаемой к данной книге. Чтобы самостоятельно создать данное приложение:

1. Создайте приложение с именем DIB по методике, описанной в *главе 1*. Оставьте все установки по умолчанию без изменения.
2. В окне **Class View** (Просмотр класса) раскройте папку **DIB**.
3. Щелкните правой кнопкой мыши на папке `CDDBView` и выберите в раскрывшемся контекстном меню команду **Add | Add Variable** (Добавить | Добавить переменную). Появится диалоговое окно **Add Member Variable Wizard** (Мастер добавления переменной в класс).

4. В текстовое поле **Variable Type** (Тип переменной) введите тип переменной `BITMAPINFO*`, а в текстовое поле **Variable name** (Имя переменной) введите имя `m_Info` и нажмите кнопку **Finish** (Готово).
5. В окне **Class View** (Просмотр класса) раскройте папку `CDIBView`, щелкните левой кнопкой мыши на конструкторе класса и выберите в раскрывшемся контекстном меню команду **Go To Definition** (Перейти к заголовку класса). Раскроется окно редактирования файла реализации DIBView.cpp, а текстовый курсор будет располагаться в заготовке конструктора класса `CDIBView`.
6. Измените конструктор и деструктор класса `CDIBView` в соответствии с текстом листинга 6.9.

Листинг 6.9. Конструктор и деструктор класса `CDIBView`

```
// Конструктор и деструктор класса CDIBView

CDIBView::CDIBView()
{
  // Выделение памяти под объект структуры BITMAPINFO
  m_Info      = (BITMAPINFO*) new char[ sizeof(BITMAPINFOHEADER) +
(sizeof(RGBQUAD) << 8) ];

  // Создание структуры для хранения информации о битовом образе
  m_Info-> bmiHeader.biSize      = sizeof(BITMAPINFOHEADER);
  m_Info-> bmiHeader.biWidth      = 4;
  m_Info-> bmiHeader.biHeight      = 0;
  m_Info-> bmiHeader.biPlanes      = 1;
  m_Info-> bmiHeader.biBitCount    = 8;
  m_Info-> bmiHeader.biCompression = BI_RGB;
  m_Info-> bmiHeader.biSizeImage    = 0;
  m_Info-> bmiHeader.biXPelsPerMeter  = 0;
  m_Info-> bmiHeader.biYPelsPerMeter  = 0;
  m_Info-> bmiHeader.biClrUsed        = 0;
  m_Info-> bmiHeader.biClrImportant   = 0;

  // Заполнение палитры
  for(int i=0; i < 256; i++)
  {
    m_Info-> bmiColors[i].rgbBlue    = (BYTE) i;
    m_Info-> bmiColors[i].rgbGreen  = (BYTE) i;
    m_Info-> bmiColors[i].rgbRed    = (BYTE) i;
```

```
    m_Info-> bmiColors[i].rgbReserved  = 0;
  }
}

CDIBView::~CDIBView()
{
  delete  m_Info;
}
```

7. Измените функцию `OnDraw` в соответствии с текстом листинга 6.10.

Листинг 6.10. Функция `CDIBView::OnDraw`

```
// Вывод информации на экран в классе CDIBView

void CDIBView::OnDraw(CDC* pDC)
{
  // Получение указателя на связанный объект класса документа
  // и проверка его корректности
  CDIBDoc* pDoc = GetDocument();
  ASSERT_VALID(pDoc);

  int  i, k, q;
  CRect  Draw_Rect;

  // Получение координат рабочей области окна
  GetClientRect(Draw_Rect);

  // Настройка объекта структуры BITMAPINFO на область вывода
  int  Rect_Height  = Draw_Rect.Height();
  int  Rect_Width  = Draw_Rect.Width();
  HDC  hDc    = pDC-> GetSafeHdc();
  m_Info-> bmiHeader.biHeight    = Rect_Height;
  m_Info-> bmiHeader.biSizeImage  = Rect_Height << 2;
  m_Info-> bmiHeader.biXPelsPerMeter  = (::GetDeviceCaps(hDc,
HORZRES)*1000)/::GetDeviceCaps(hDc, HORZSIZE);
  m_Info-> bmiHeader.biYPelsPerMeter  = (::GetDeviceCaps(hDc,
VERTRES)*1000)/::GetDeviceCaps(hDc, VERTSIZE);

  char    Frg[8192];
  for(i=0; i < Rect_Width; i++)
  {
```

```
    // Вывод битового образа на экран
    q = i & 3;
    for(k=0; k < Rect_Height; k++)
      Frg[ (k << 2) + q] = (char) (i+k);
    if(q == 3)
      ::SetDIBitsToDevice(hDc, i — 3, 0, 4, Rect_Height, 0, 0, 0,
Rect_Height, Frg, m_Info, DIB_RGB_COLORS);
  }

  // Вывод последних столбцов битового образа
  if(q != 3)
    ::SetDIBitsToDevice(hDc, i — q, 0,  q+1, Rect_Height, 0, 0, 0,
Rect_Height, Frg, m_Info, DIB_RGB_COLORS);
}
```

8. Нажмите клавишу <F5> и запустите приложение на исполнение. Окно приложения будет иметь вид, изображенный на рис. 6.13. (Если, конечно вы себя уважаете и работаете в нормальной палитре.)

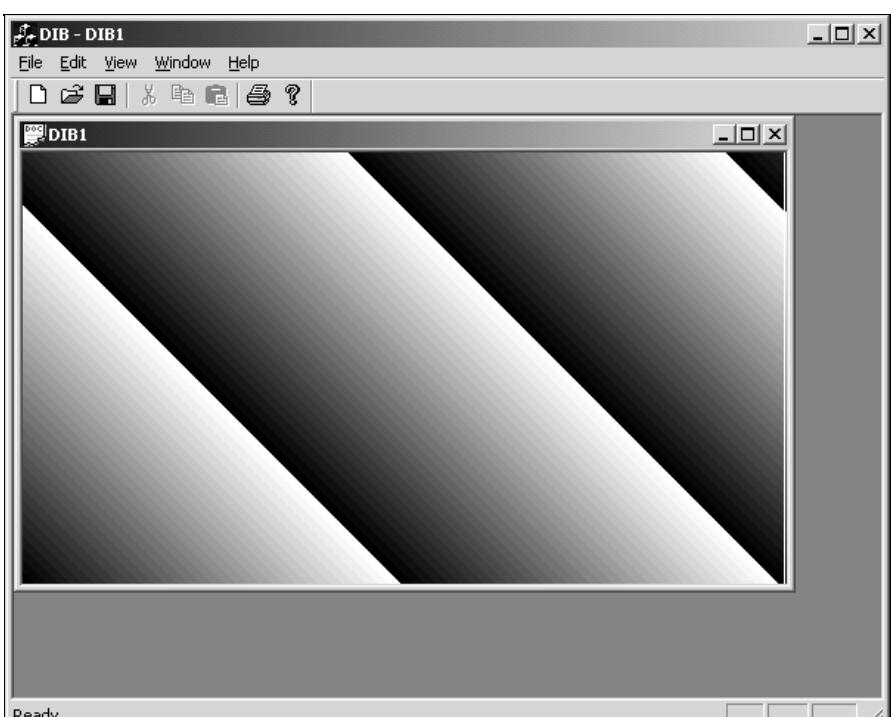


Рис. 6.13. Вывод аппаратно-независимого битового образа (режим High Color)

9. Установите системную палитру 256 цветов. Окно приложения изменится и будет выглядеть как на рис. 6.14.

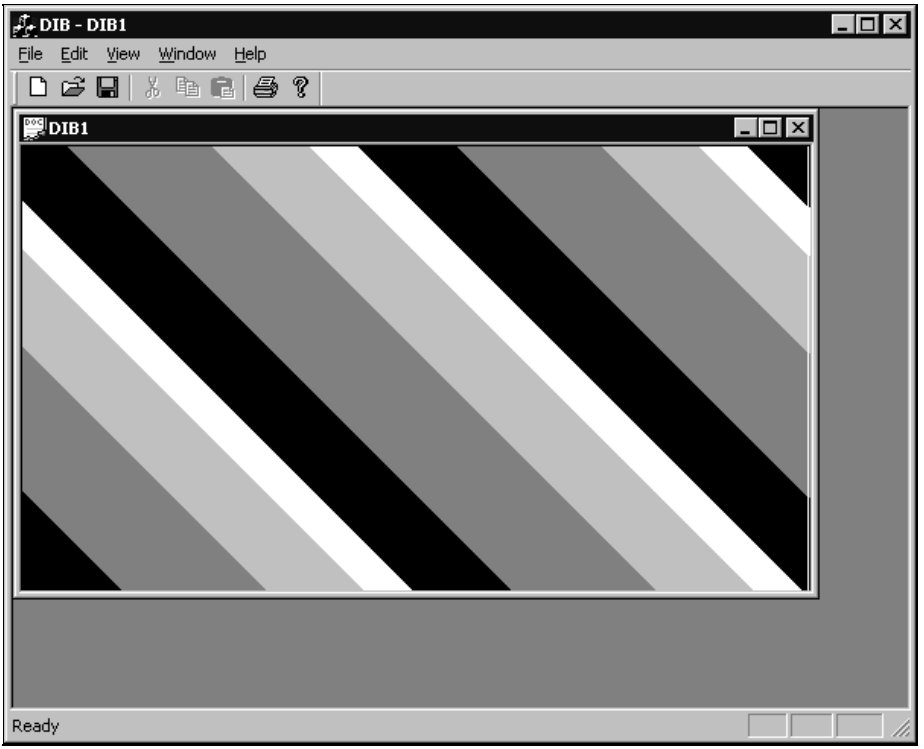


Рис. 6.14. Вывод аппаратно-независимого битового образа (режим 256 цветов)

10. Закройте приложение.

Прежде всего, в конструкторе класса `CDIBView` создается объект структуры `BITMAPINFO`, используемый для хранения информации о размере и палитре аппаратно-независимого битового образа (DIB) и производится его заполнение. Объект этой структуры является динамическим, поскольку его размер зависит от размера содержащейся в ней таблицы цветов.

Информация о размере и других характеристиках битового образа хранится в переменной `bmiHeader`, представляющей собой объект структуры `BITMAPINFOHEADER`. Данный объект структуры заполняется не полностью, поскольку содержимое некоторых его переменных зависит от характеристик устройства вывода графической информации. Этим переменным присваивается нулевое значение. После заполнения переменной `bmiHeader` производится заполнение массива `bmiColors`, элементы которого представляют собой объекты

структуры `RGBQUAD`, содержащие описание цветов палитры, используемой данным битовым образом.

Поскольку объект структуры `BITMAPINFO` является динамическим, он должен быть уничтожен в деструкторе класса `CDIBView`.

Для непосредственного вывода изображения на экран используется функция `CDIBView::OnDraw`. После получения указателя на объект класса `CDIBDoc` и проверки корректности полученного указателя (который, впрочем, нигде не будет задействован) с использованием функции `GetClientRect` определяется размер рабочей области окна, в которую и будет выводиться битовый образ. Полученная информация используется для инициализации соответствующих переменных объекта структуры `BITMAPINFOHEADER`.

После этого производится вывод черно-белого битового образа, цвет точки в котором определяется как младший байт суммы ее вертикальной и горизонтальной координат.

В соответствии с требованиями, предъявляемыми к аппаратно-независимому битовому образу размер строки выводимого изображения должен быть кратен размеру двойного слова, т. е. четырем байтам. Поскольку выводимая информация имеет размер 1 байт, то для более эффективной работы функции в ней выводится сразу четыре столбца изображения. Для этого вызывается функция `SetDIBitsToDevice`. Если размер области вывода оказывается не кратным четырем битам, то при выводе последних столбцов указывается их количество и все размеры изменяются соответствующим образом.

Глава 7

Работа с файлами документов

В предыдущей главе уже был рассмотрен вопрос о том, как сохранить изображение в окне при изменении его размера. Другим очень важным вопросом является вопрос о том, как восстановить изображение в окне, существовавшее до его закрытия, при новом открытии этого окна. Сходная проблема возникает и при открытии окна в новом приложении. Данный вопрос решается в рамках концепции Документ/Представление, описанной в *главе 2*. Согласно ей вся информация о выводимом в окне изображении хранится в объекте класса документа, связанного с объектом класса представления, осуществляющего вывод этого изображения на экран или на принтер.

Работа с архивом

Как уже отмечалось в *главе 2*, для чтения и записи информации, содержащейся в документе, при создании и уничтожении объекта класса этого документа используется функция `CObject::Serialize`, имеющая своим аргументом объект класса `CArchive`. Данная функция осуществляет инициализацию объекта класса документа при его создании и сохранение имеющейся в нем информации при уничтожении данного объекта. Для хранения данных объекта используется файл архива на диске. Данный файл не содержит специальных записей, характеризующих его как файл архива, поэтому в этом качестве может выступать любой файл, даже, созданный средствами операционной системы MS-DOS.

Изменим созданное нами приложение Text таким образом, чтобы выводимая в окне информация сохранялась в файле архива и появлялась бы на экране после открытия этого файла.

Чтобы внести в приложение Text изменения, позволяющие сохранять информацию, выводимую в окно, в файле архива:

1. Выберите команду **File | Open Solution** (Файл | Открыть решение) и в открывшемся диалоговом окне **Open Solution** (Открыть решение) раскройте папку **Text** (Текст). Диалоговое окно **Open Solution** (Открыть решение) примет вид, изображенный на рис. 7.1.
2. В окне списка выделите значок **Text** (Текст) и нажмите кнопку **Open** (Открыть).

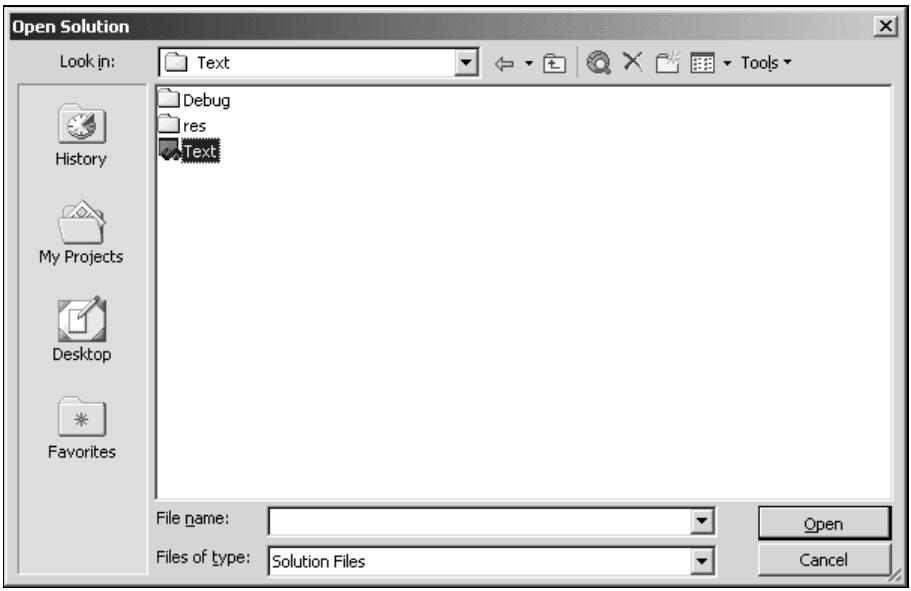


Рис. 7.1. Диалоговое окно **Open Solution**

3. Откройте окно редактирования файла TextView.cpp и вставьте в текст функции `OnLButtonDown` после строки `lpDoc-> aY.Add(point.y);` строку

```
// Установка флага изменения
lpDoc-> SetModifiedFlag();
```

4. Откройте окно редактирования файла TextDoc.cpp и измените функцию `Serialize` в соответствии с текстом листинга 7.1.

Листинг 7.1. Функция `CTextDoc::Serialize`

```
// Сохранение документа в потоке и чтение из него в классе CTextDoc

void CTextDoc::Serialize(CArchive& ar)
{
  int    i, n;
  if (ar.IsStoring())
  {
    // Сохранение документа в потоке
    n = aX.GetSize();
    ar << n;
    for(i=0; i < n; i++)
    {
```

```
      ar << aX[i];
      ar << aY[i];
    }
  }
  else
  {
    // Чтение документа из потока
    ar >> n;
    aX.SetSize(n);
    aY.SetSize(n);
    for(i=0; i < n; i++)
    {
      ar >> aX[i];
      ar >> aY[i];
    }
  }
}
```

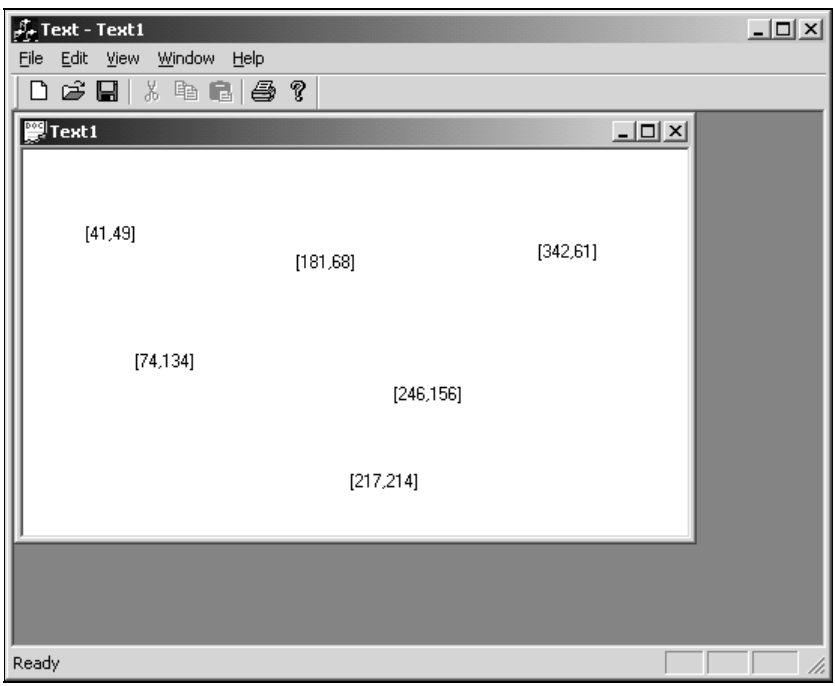


Рис. 7.2. Окно приложения Text

5. Нажмите клавишу <F5> и запустите приложение на исполнение. Появится окно приложения Text.
6. Несколько раз щелкните левой кнопкой мыши в рабочей области окна. Окно примет вид, изображенный на рис. 7.2.
7. Выберите команду меню **File | Close** (Файл | Закрыть) или нажмите кнопку **Close** (Закрыть) в окне документа. Появится диалоговое окно **Text** (Текст), изображенное на рис. 7.3.
8. Нажмите кнопку **Yes** (Да), подтвердив необходимость сохранения изменений, внесенных в документ Text1. Появится стандартное диалоговое окно **Save As** (Сохранить как), изображенное на рис.7.4.

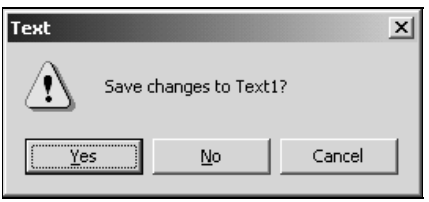


Рис. 7.3. Диалоговое окно **Text**

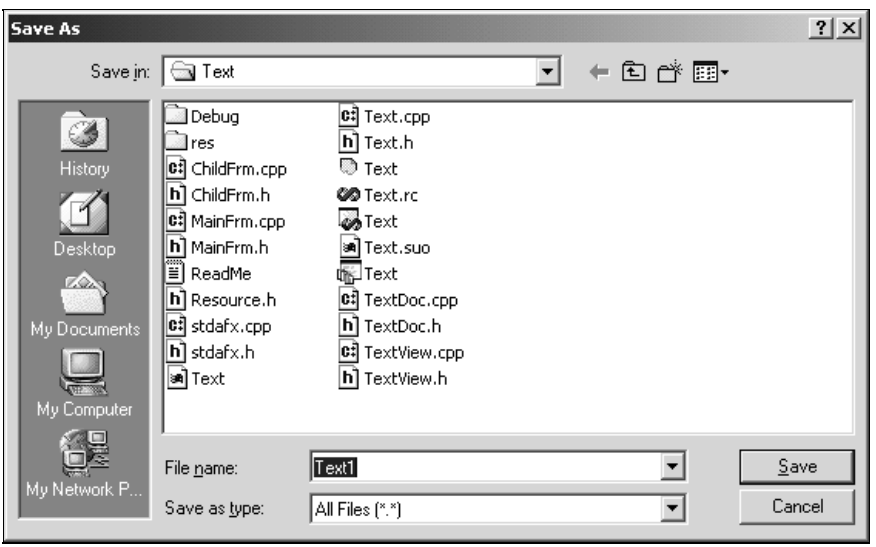


Рис. 7.4. Диалоговое окно **Save As**

9. Нажмите кнопку **Save** (Сохранить). Окно документа закроется.

10. Выберите команду меню **File | Open** (Файл | Открыть) или нажмите кнопку **Open** (Открыть) на панели инструментов. Появится стандартное диалоговое окно **Open** (Открыть), изображенное на рис. 7.5.

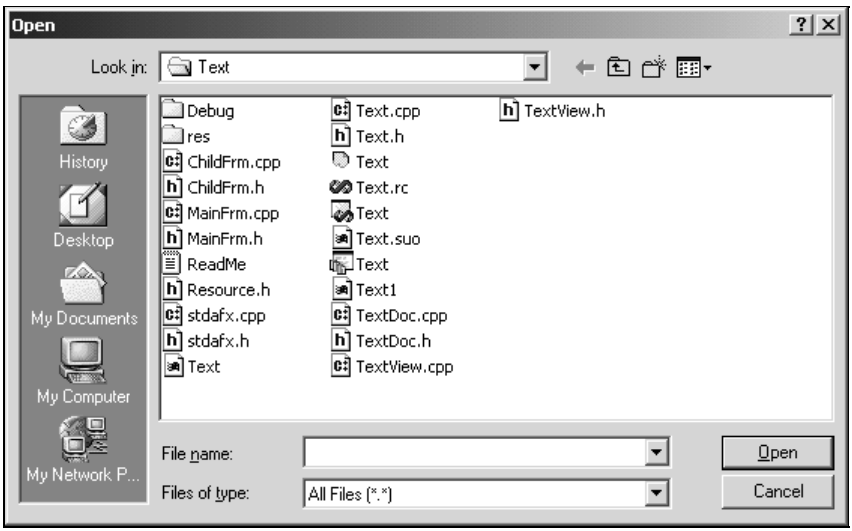


Рис. 7.5. Диалоговое окно **Open**

11. В окне списка выделите значок **TextApp1** и нажмите кнопку **Open** (Открыть). Откроется окно документа, в котором будет сохранена вся информация, отображавшаяся в нем до его закрытия.

Использование в данном приложении функции `CDocument::SetModifiedFlag`, вызываемой после записи в массив очередных координат щелчка левой кнопкой мыши позволяет приложению определить, что данный документ не является пустым и, следовательно, нуждается в сохранении. Следствием вызова данной функции является появление диалогового окна, изображенного на рис. 7.3 и связанные с ним действия по сохранению документа.

Функция `CObject::Serialize` осуществляет непосредственную работу по инициализации и сохранению документа. Вызов функции `CArchive::IsStoring` позволяет определить, какая именно операция производится в данный момент. При сохранении информации, хранящейся в документе, в архив сначала заносится размер массивов координат точек, а затем сохраняются сами координаты в виде пар координат точек. Для сохранения объекта в потоке используется переопределенный в классе `CArchive` оператор `<<`. При инициализации объекта класса документа сохраненной в архиве информацией сначала считывается размер массивов координат и на основании этой информации устанавливаются размеры массивов, в которые будут записываться эти координаты. А затем производится заполнение этих массивов информацией, хранящейся в архиве. Для

чтения объекта из потока используется переопределенный в классе `CArchive` оператор >>.

Данное приложение имеет два основных недостатка:

- документ сохраняется в файле без расширения, что приводит к необходимости отображать в окне **Open** (Открыть) и в окне **Save As** (Сохранить как) все имеющиеся в данном каталоге файлы;
- файл записывается в рабочий каталог программы, что в большинстве случаев не совсем удобно.

Чтобы устранить перечисленные выше недостатки:

1. Раскройте окно **Resource View** (Просмотр ресурсов).
2. Раскройте папку **Text.rc**, а в ней — папку **String Table** (Таблица строковых ресурсов) и дважды щелкните левой кнопкой мыши на значке **String Table**. В окне редактирования появится таблица строковых ресурсов.
3. Найдите в ней идентификатор ресурса `IDR_TextTYPE` и щелкните на содержимом столбца **Caption** (Заголовок) данной строки. Появится поле для редактирования, изображенное на рис. 7.6.

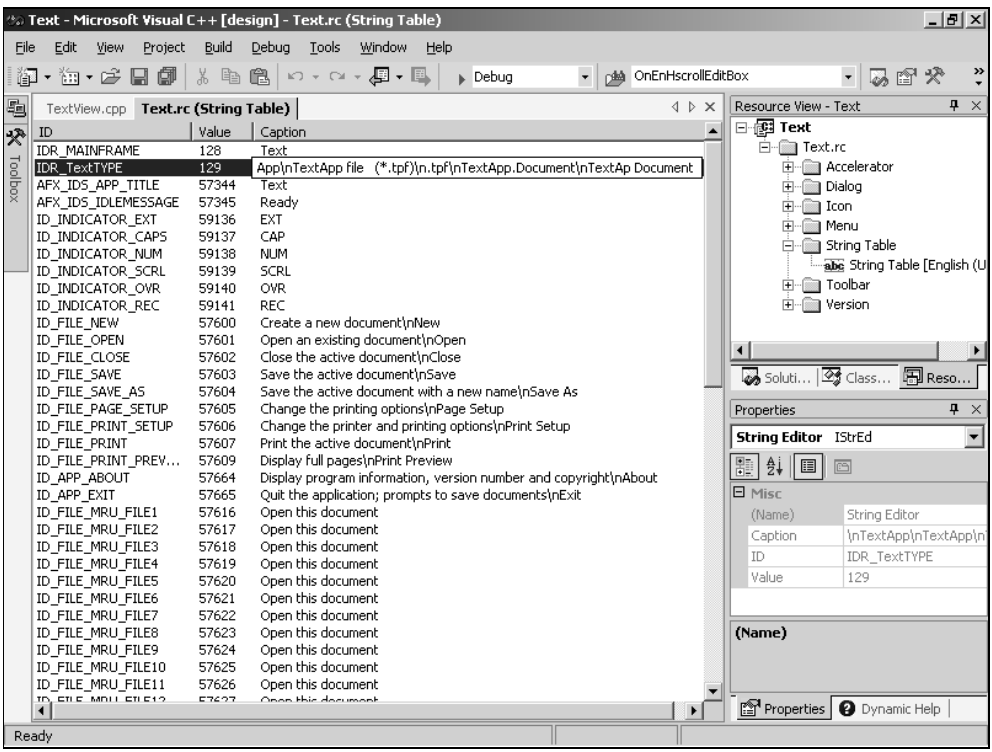


Рис. 7.6. Редактирование строкового ресурса

4. Введите в это поле текст "\nTextApp\nTextApp\nTextApp file (*.tpf)\n.tpf\nTextApp.Document\nTextAp Document".
5. Откройте окно редактирования файла Text.cpp и вставьте в текст функции `InitInstance` перед оператором `return TRUE;` следующий текст:

```
// Чтение рабочего каталога из системного реестра
CString    Work_Dir;

Work_Dir = GetProfileString("CONTROL","DIR","c:\\");
SetCurrentDirectory(Work_Dir);
```

6. Откройте окно редактирования файла TextDoc.cpp и измените функцию `Serialize` в соответствии с текстом листинга 7.2.

Листинг 7.2. Функция `CTextDoc::Serialize`

```
// Сохранение документа в потоке и чтение из него в классе CTextDoc

void CTextDoc::Serialize(CArchive& ar)
{
  int    i, n;
  CWinApp*  pApp = AfxGetApp();
  if (ar.IsStoring())
  {
    // Сохранение документа в потоке
    n = aX.GetSize();
    ar << n;
    for(i=0; i < n; i++)
    {
      ar << aX[i];
      ar << aY[i];
    }
    // Сохранение текущего каталога в системном реестре
    char  Cur_Dir[_MAX_DIR];
    GetCurrentDirectory(_MAX_DIR, Cur_Dir);
    ASSERT(pApp-> WriteProfileString("CONTROL","DIR",Cur_Dir));
  }
  else
  {
    // Чтение документа из потока
    ar >> n;
    aX.SetSize(n);
```

```
    aY.SetSize(n);
    for(i=0; i < n; i++)
    {
      ar >> aX[i];
      ar >> aY[i];
    }
  }
}
```

7. Нажмите клавишу <F5> и запустите приложение на исполнение.
8. Несколько раз щелкните левой кнопкой мыши в рабочей области окна.
9. Выберите команду меню **File | Save** (Файл | Сохранить). Появится диалоговое окно **Save As** (Сохранить как), изображенное на рис. 7.7.

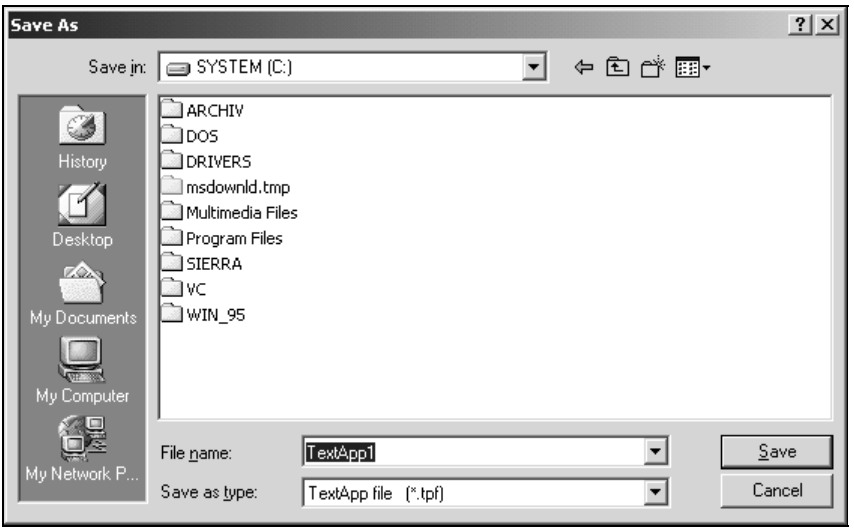


Рис. 7.7. Диалоговое окно **Save As**

10. Раскройте папку, в которой вы собираетесь сохранить документ, и нажмите кнопку **Save** (Сохранить). Диалоговое окно **Save As** (Сохранить как) закроется.
11. Закройте окно документа и выберите команду **File | Open** (Файл | Открыть) или нажмите кнопку **Open** (Открыть) на панели инструментов. Появится стандартное диалоговое окно **Open** (Открыть), изображенное на рис. 7.8.
12. Выделите в окне списка значок **TextApp1.tpf** и нажмите кнопку **Open** (Открыть). Откроется окно документа, в котором будет сохранена вся информация, отображенная в нем до закрытия.

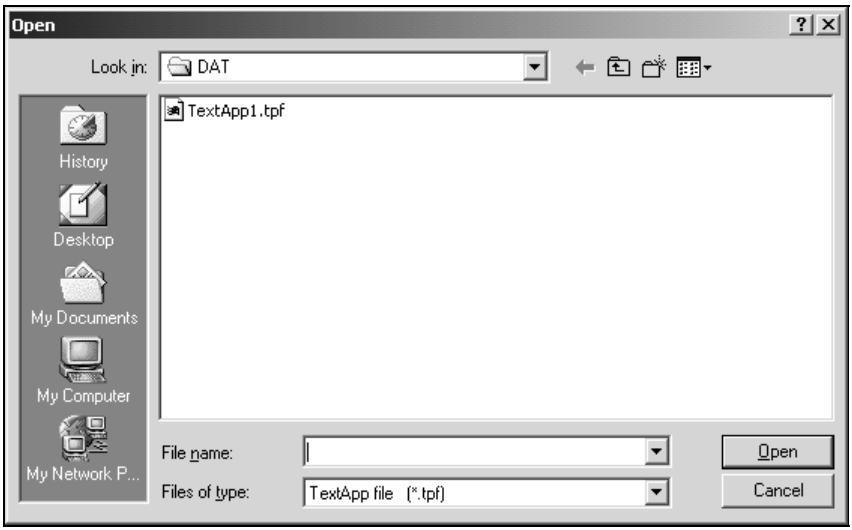


Рис. 7.8. Диалоговое окно **Open**

Основным отличием диалогового окна **Save As** (Сохранить как), приведенного на рис. 7.7, от аналогичного окна, приведенного на рис. 7.4, является то, что в текстовом поле его раскрывающегося списка **Save as type** (Тип сохраняемого файла) размещается строка "TextApp file (*.tpf)", в текстовом поле раскрывающегося списка **Save in** (Сохранить в) размещается строка "SYSTEM (C:)", а в окне списка выведены только папки, поскольку в корневом каталоге диска C: отсутствуют файлы с расширением tpf.

Основным отличием диалогового окна **Open** (Открыть), приведенного на рис. 7.8, от аналогичного окна, приведенного на рис. 7.5, является то, что в текстовом поле его раскрывающегося списка **Files of type** (Тип файлов) размещается строка "TextApp file (*.tpf)", в текстовом поле раскрывающегося списка **Look in** (Искать в) содержится имя каталога, а в окне списка файлов содержится единственное имя файла, имеющее расширение tpf.

Текст, введенный в столбце **Caption** (Заголовок) окна редактирования строкового ресурса в п. 4, является строковым ресурсом, связанным с шаблоном документа, использующего данный идентификатор ресурса. Этот строковый ресурс может содержать до семи текстовых полей, разделенных символом перевода строки (`\n`) (в данном случае символ используется для обеспечения возможности указания числа пустых полей, если после некоторого поля все остальные поля являются пустыми, то нет необходимости указывать для них символы перевода строки). Подробно эти поля даны в описании функции `CDocTemplate::GetDocString`. Здесь отметим только, что первое поле в многооконных документах не используется, а заданные нами четвертое и пятое поля содержат строку, выводящую описание типа документа и его расширения, используемого для поиска доку-

ментов данного типа (эта строка отображается в раскрывающемся списке **Files of type** (Тип файлов) диалогового окна **Open** (Открыть) и в раскрывающемся списке **Save as type** (Тип сохраняемого файла) диалогового окна **Save As** (Сохранить как)).

Для того чтобы приложение запоминало рабочий каталог, установленный при предыдущем запуске приложения, он сохраняется в системном реестре, представляющем собой файл, в который записываются сведения о параметрах всех приложений, зарегистрированных в операционной системе данного компьютера. Регистрация приложения является обязательной процедурой для его нормального функционирования в системе. Каждому зарегистрированному приложению в системном реестре соответствует запись, в которой хранится вся информация, необходимая приложению для нормального возобновления работы. Характер данной информации определяется исключительно самим приложением. Более подробно системный реестр будет рассмотрен в заключительном разделе данной главы.

Для записи текстовой строки в системный реестр используется функция `CWinApp::WriteProfileString`, первый аргумент которой определяет секцию, в которую необходимо записать текстовую строку, указанную в третьем аргументе данной функции, а второй — содержит идентификатор записываемой строки в секции. Данная функция вызывается из функции `CTextDoc::Serialize` после того, как пользователь сохранил в файле информацию о содержимом окна. Поскольку сохранение информации производится в текущем каталоге процесса, после сохранения информации в файле вызывается функция `GetCurrentDirectory`, сохраняющая в переменной `Cur_Dir` текстовую строку, содержащую полный путь в текущий каталог приложения. После этого вызывается функция `WriteProfileString`, сохраняющая данную строку в системном реестре. Указатель на объект класса `CWinApp`, вызывающий данную функцию, получается посредством вызова функции `AfxGetApp`.

Для чтения текстовой строки из системного реестра используется функция `CWinApp::GetProfileString`, первый аргумент которой представляет собой текстовую строку, определяющую секцию, в которой содержится текстовая строка, возвращаемая данной функцией, второй аргумент является текстовой строкой, содержащей идентификатор возвращаемой строки в секции, а третий аргумент — текстовой строкой, возвращаемой функцией в случае отсутствия указанной записи в системном реестре. Данная функция вызывается из функции `CText::InitInstance`, поскольку именно в этой функции задаются основные параметры создаваемого объекта класса документа. Одним из таких параметров можно считать его рабочий каталог. Поскольку функция `InitInstance` является членом класса, производного от класса `CWinApp`, в данном случае нет необходимости получать указатель на объект данного класса. Возвращаемое данной функцией значение используется в качестве аргумента функции `SetCurrentDirectory`, устанавливающей новый рабочий каталог приложения.

Непосредственное чтение и запись файлов

Использование архива позволяет достаточно просто решить вопрос сохранения на диске и чтения с диска отдельных документов. Однако основной особенностью документов Windows является то, что они рассматриваются как единое целое. Это значит, что при создании документа в него записывается вся необходимая для его работы информация, а после завершения работы вся эта информация сохраняется на диске. Последовательный принцип доступа к файлу требует чтения, а тем более, записи всей сохраняемой информации в документе. Кроме того, в классе документа Windows не предусмотрены методы для работы с диском в процессе работы с документом.

Предположим, данное приложение работает с мультимедийными объектами, такими как файлы фонограмм или файлы изображений. Требование загрузить всю информацию, связанную с данными объектами в оперативную память, может оказаться невыполнимым. Поэтому при работе с данным типом объектов приходится постоянно обращаться к расположенному на диске файлу.

Предположим, что в приложении используется шаблон документа, предназначенный для вывода на экран фонограммы звукового файла. Размер этого файла может достигать нескольких мегабайтов и, в принципе, не ограничен. В данном случае можно использовать функцию `CDocument::Serialize` для создания промежуточного массива, хранящего достаточно информации для того, чтобы корректно отображать фонограмму при любом разрешении и любом размере окна. Казалось бы, вопрос решен и нет необходимости в дальнейших обращениях к исходному файлу. Однако пользователь может потребовать увеличить масштаб отображения или произвести некоторые операции редактирования. В этом случае информации, хранящейся в созданном функцией `Serialize` массиве, может быть недостаточно для корректного отображения фонограммы, а операции редактирования требуют непосредственного доступа ко всему файлу. Данные проблемы решаются при использовании объектов класса `CFile`.

Использование объектов класса *CFile* при работе с классом *CArchive*

Класс `CFile` является базовым классом библиотеки MFC для работы с файлами. Он обеспечивает произвольный доступ к двоичному файлу. Данный класс используется совместно с классом `CArchive` для сохранения объектов классов из библиотеки MFC. Файл на диске автоматически открывается конструктором класса `CFile` и автоматически закрывается его деструктором. Подробное описание данного класса содержится в *приложении 2*. Здесь будут рассмотрены только некоторые аспекты работы с объектами данного класса.

В приведенном выше примере возникла необходимость работы с файлом после заполнения документа функцией `CDocument::Serialize`. Поскольку данная

функция является неотъемлемой частью класса документа и структура приложения оптимизирована под эту функцию, нет никого смысла игнорировать эту функцию при работе с файлом в приложении. Целесообразнее использовать ее для инициализации класса документа информацией, хранящейся в файле, и сохранения дополнительной информации, необходимой для повторного открытия файла. Как видно из описания конструктора данного класса, для повторного открытия файла необходимо задать только его имя и атрибуты открытия.

Модифицируем приложение Text, чтобы непосредственно обращаться из него к файлу в процессе работы с документом. Для этого:

1. Выберите команду **File | Open Solution** (Файл | Открыть решение) и в открывшемся диалоговом окне **Open Solution** (Открыть решение) раскройте папку **Text** (Текст), выделите значок **Text** и нажмите кнопку **Open** (Открыть). Или выберите команду **File | Recent Projects** (Файл | Последние проекты) и в открывшемся контекстном меню выберите файл соответствующего проекта. Проект станет активным и в окнах Visual C++ восстановится информация, содержавшаяся в них до последнего закрытия данного проекта.
2. Откройте окно **Class View** (Просмотр класса), а в нем — папку **Text** (Текст) и щелкните правой кнопкой мыши на папке `CTextView`.
3. В раскрывшемся контекстном меню выберите команду **Properties** (Свойства). Откроется окно **Properties** (Свойства).
4. В окне **Properties** (Свойства) нажмите кнопку **Messages** (Сообщения). Раскроется список сообщений, обрабатываемых данным классом.
5. В открывшемся списке раскройте список `WM_RBUTTONDOWN` и выделите в нем единственную строку. В класс `CTextView` будет добавлена функция обработки сообщения `OnRButtonDown`.
6. В окне **Class View** (Просмотр класса) раскройте папку `CTextView` и дважды щелкните левой кнопкой мыши на функции обработки сообщения `OnRButtonDown`. Откроется окно редактирования файла TextView.cpp, а текстовый курсор будет помещен в заготовку данной функции.
7. Измените функцию `OnRButtonDown` в соответствии с текстом листинга 7.3.

Листинг 7.3. Функция `CTextView::OnRButtonDown`

```
// Восстановление информации из файла

void CTextView::OnRButtonDown(UINT nFlags, CPoint point)
{
  // Получение указателя на связанный объект класса документа
  CTextDoc*    lpDoc = GetDocument();
  CFileException  fe;
  int      i, n;
  if(lpDoc-> GetPathName() == "")  // Файл документа отсутствует
```

```
  {
    CView::OnRButtonDown(nFlags, point);
    return;
  }

  // Создание объекта класса CFile
  CFile*  lpFile = lpDoc-> GetFile(lpDoc-> GetPathName(),
     CFile::modeRead  |  CFile::shareDenyNone, &fe);

  if(fe.m_cause)  // Возникла ошибка
  {
    ::MessageBox(0, "Ошибка при создании файла", "Ошибка",
          MB_OK  |  MB_SYSTEMMODAL);
    CView::OnRButtonDown(nFlags, point);
    return;
  }

  // Чтение длины файла
  lpFile->Read(&n, sizeof(int));

  // Установка размеров массива
  lpDoc-> aX.SetSize(n);
  lpDoc-> aY.SetSize(n);

  // Чтение сохраненной информации
  for(i=0; i < n; i++)
  {
    lpFile->Read(&lpDoc-> aX[i], sizeof(UINT));
    lpFile->Read(&lpDoc-> aY[i], sizeof(UINT));
  }

  // Освобождение файла
  lpDoc->ReleaseFile(lpFile, true);

  // В файле отсутствует новая информация
  lpDoc-> SetModifiedFlag(FALSE);

  // Перерисовка окна
  Invalidate();

  CView::OnRButtonDown(nFlags, point);
}
```

8. Нажмите клавишу <F5> и запустите приложение на исполнение.
9. Нажмите кнопку **Open** (Открыть) на панели инструментов, в появившемся диалоговом окне выделите файл с расширением tpf и нажмите кнопку **Open** (Открыть). (Если файл с таким расширением отсутствует, создайте его по методике, описанной в предыдущем разделе.)
10. Появится окно документа с сохраненной в файле информацией.
11. Несколько раз щелкните левой кнопкой мыши и добавьте в окно новые записи.
12. После этого щелкните правой кнопкой мыши. Все новые записи исчезнут.

Для возврата окна к сохраненному состоянию в функции `OnRButtonDown` используется функция `CDocument::GetFile`. Поскольку первым аргументом данной функции является путь к открываемому файлу, а вторым ее аргументом являются флаги открытия файла, можно сделать вывод, что в объекте класса `CDocument` не хранится связанный с ним объект класса `CFile`. Единственным параметром, связывающим данный документ с файлом, является хранящееся в объекте данного класса полное имя файла. Оно может быть получено с помощью вызова функции `CDocument::GetPathName`. Как видно из описания данной функции, приведенного на дискете, возвращаемая данной функцией строка может быть пустой. При передаче пустой строки в качестве первого аргумента функции `GetFile` возникает ошибка. Чтобы избежать ее, возвращаемое функцией `GetPathName` значение проверяется на допустимость, и в том случае, если оно является пустой строкой, выполнение функции `OnRButtonDown` прерывается и после вызова метода базового класса производится выход из функции.

В случае положительного исхода данной проверки с помощью функции `CDocument::GetFile` создается указатель на объект класса `CFile`. Поскольку последний аргумент данной функции содержит код возможной ошибки, то этот код проверяется и, в случае возникновения ошибки при выполнении данной функции выдается соответствующее сообщение. Для этого используется функция `MessageBox`, нулевое значение первого аргумента и флаг `MB_SYSTEMMODAL` говорит о том, что работа со всеми приложениями, запущенными в данном сеансе работы с операционной системой, будет невозможна, пока пользователь не нажмет кнопку **OK** в появившемся окне сообщения. Поскольку само по себе окно сообщения не предпринимает каких-либо действий, после его закрытия вызывается метод базового класса и функция прекращает свою работу.

После этого работа функции напоминает работу соответствующей секции функции `CTextDoc::Serialize` за тем исключением, что класс `CFile`, в отличие от класса `CArchive`, не содержит перегруженного оператора `>>` и поэтому операция ввода реализуется функцией `CFile::Read`. Первый аргумент данной функции представляет собой указатель на буфер, в который будет записана считанная из файла информация. В данном случае производится чтение целочисленных переменных, поэтому в качестве первого аргумента используется ссылка на соответствующую статическую переменную. Второй аргумент данной функции определяет длину буфера в байтах. Поскольку в качестве первого аргумента

используется ссылка на статическую переменную, в качестве второго аргумента используется оператор `sizeof`, возвращающий размер переменных заданного типа.

После чтения информации из файла производится его закрытие с использованием функции `CDocument::ReleaseFile`. Затем следует вызов функции `CWnd::Invalidate`, производящей перерисовку окна после того, как в документ будут внесены изменения. Последним оператором данной функции является вызов метода базового класса.

Данный пример включает в себя основные методы работы с файлом, связанным с данным шаблоном документа. Естественно, что при необходимости промежуточного сохранения информации в данном файле необходимо использовать функцию `CFile::Write`. В данном примере она не использовалась, но работа с ней ничем принципиально не отличается от работы с функцией `CFile::Read`.

При непосредственной работе с файлом необходимо всегда помнить, что он был открыт с помощью архива, и система считает, что и сохраняться он должен с помощью архива. Это обстоятельство часто забывают при работе с большими документами, которые нецелесообразно полностью записывать в память. В этом случае все изменения в документе сразу же записываются в связанный с ним файл, и операция сохранения документа теряет смысл. Поэтому некоторые пользователи считают, что секцию сохранения документа в функции `Serialize` можно не заполнять программным кодом. Но приложение об этом "не знает" и при вызове команды **File | Save** (Файл | Сохранить) инициирует процедуру сохранения документа, который, как она полагает, хранится в памяти компьютера. Не найдя операторов сохранения файла, она создает файл нулевой длины, т. е. уничтожает всю хранившуюся в документе информацию. Одним из способов избежать данной ситуации является создание в классе представления функции обработки сообщения `UPDATE_COMMAND_UI` для идентификатора ресурса `ID_FILE_SAVE`, единственным оператором которой должен быть вызов функции `CCmdUI::Enable` с аргументом `FALSE`, делающим соответствующую команду недоступной. Однако, если в данном приложении используется команда **File | Save As** (Файл | Сохранить как), в секции сохранения документа в функции `Serialize` необходимо поместить операторы, позволяющие переписать содержимое файла данного документа в другой файл.

Автономное использование класса *CFile*

Создание в данном приложении объекта класса `CFile` в качестве дублера класса `CArchive` не означает, что объект данного класса не может использоваться самостоятельно. Единственное, что нужно для его создания, это указать имя файла, с которым будет работать данный класс, а это можно сделать и не прибегая к услугам класса `CArchive`.

Объекты класса `CFile`, кроме дублирования архива, обычно используются для работы с файлами, обработка которых производится вне рамок концепции Документ/Представление и для создания пользовательского буфера обмена боль-

шого объема. Каждая из этих областей применения имеет свою специфику *(см. главу 2)*.

Чтобы включить в приложение Text объекты класса `CFile`, используемые для создания пользовательского буфера обмена и работы с файлом вне рамок концепции Документ/Представление:

1. Выберите команду **File | Open Solution** (Файл | Открыть решение) и в открывшемся диалоговом окне **Open Solution** (Открыть решение) раскройте папку **Text** (Текст), выделите значок **Text** и нажмите кнопку **Open** (Открыть). Или выберите команду **File | Recent Projects** (Файл | Последние проекты) и в открывшемся контекстном меню выберите файл соответствующего проекта. Проект станет активным и в окнах Visual C++ восстановится информация, содержавшаяся в них до последнего закрытия данного проекта.
2. Откройте окно **Class View** (Просмотр класса), а в нем — папку **Text** (Текст) и щелкните правой кнопкой мыши на папке `CTextDoc`.
3. В раскрывшемся контекстном меню выберите команду **Add | Add Variable** (Добавить | Добавить переменную). Появится диалоговое окно **Add Member Variable Wizard - Text** (Мастер добавления переменной в класс).
4. Введите в текстовое поле **Variable type** (Тип переменной) тип указателя на класс `CFile*`, в текстовое поле **Variable name** (Имя переменной) введите идентификатор переменной `lpClipFile` и нажмите кнопку **Finish** (Готово).
5. Повторите п.п. 2—4 для переменной `lpOutFile`, имеющей тип `CFile*`.
6. Откройте окно редактирования файла TextView.cpp и вставьте в текст функции `OnLButtonDown` после сроки `lpDoc-> aY.Add(point.y);` следующие строки:

```
// Запись во временный файл
lpDoc-> lpClipFile-> Write(&point.x, sizeof(LONG));
lpDoc-> lpClipFile-> Write(&point.y, sizeof(LONG));
```

7. Измените функцию `OnRButtonDown` в соответствии с текстом листинга 7.4.

Листинг 7.4. Функция `CTextView::OnRButtonDown`

```
// Восстановление информации из файла

void CTextView::OnRButtonDown(UINT nFlags, CPoint point)
{
  int        i, n;
  CString    File_Name;

  // Получение указателя на связанный объект класса документа
  CTextDoc*  lpDoc = GetDocument();
```

```
// Получение указателя на объект класса приложения и проверка
CWinApp*  pApp = AfxGetApp();
VERIFY(pApp-> m_pDocManager != NULL);

// Вывод диалогового окна для получения имени выходного файла
if(pApp-> m_pDocManager-> DoPromptFileName(File_Name,
    AFX_IDS_SAVEFILE,
    OFN_HIDEREADONLY  |  OFN_PATHMUSTEXIST, FALSE, NULL))
{
  // Сохранение текущего каталога в системном реестре
  char  Cur_Dir[_MAX_DIR];
  GetCurrentDirectory(_MAX_DIR, Cur_Dir);
  VERIFY(WriteProfileString("CONTROL","DIR",Cur_Dir));

  // Создание выходного файла
  try
  {
    lpDoc-> lpOutFile  = new CFile(File_Name,
       CFile::modeCreate  |  CFile::modeReadWrite  |
       CFile::shareDenyNone);
  }
  // Обработка ошибок, возникших при создании файла
  catch(...)
  {
    ::MessageBox(0, "Ошибка при открытии выходного файла",
                 "Ошибка", MB_OK | MB_SYSTEMMODAL);
    exit(1);
  }

  // Копирование содержимого временного файла в выходной файл
  UINT  nTemp;
  n = lpDoc-> lpClipFile-> GetLength()/(2*sizeof(UINT));
  lpDoc-> lpOutFile-> Write(&n, sizeof(int));
  lpDoc-> lpClipFile-> SeekToBegin();
  while(lpDoc-> lpClipFile-> Read(&nTemp, sizeof(LONG)))
    lpDoc-> lpOutFile-> Write(&nTemp, sizeof(LONG));
}

// Восстановление информации на экране
CFileException  fe;
if(lpDoc-> GetPathName() == "")  // Файл документа отсутствует
```

```
  {
    CView::OnRButtonDown(nFlags, point);
    return;
  }

  // Создание объекта класса CFile
  CFile*  lpFile = lpDoc-> GetFile(lpDoc-> GetPathName(), CFile::modeRead  |
CFile::shareDenyNone, &fe);

  if(fe.m_cause)  // Возникла ошибка
  {
    ::MessageBox(0, "Ошибка при создании файла",
                 "Ошибка", MB_OK  |  MB_SYSTEMMODAL);
    CView::OnRButtonDown(nFlags, point);
    return;
  }

  // Чтение длины файла
  lpFile->Read(&n, sizeof(int));

  // Установка размеров массива
  lpDoc-> aX.SetSize(n);
  lpDoc-> aY.SetSize(n);

  // Чтение сохраненной информации
  for(i=0; i < n; i++)
  {
    lpFile->Read(&lpDoc-> aX[i], sizeof(UINT));
    lpFile->Read(&lpDoc-> aY[i], sizeof(UINT));
  }

  // Освобождение файла
  lpDoc->ReleaseFile(lpFile, true);

  // В файле отсутствует новая информация
  lpDoc-> SetModifiedFlag(FALSE);

  // Перерисовка окна
  Invalidate();
  CView::OnRButtonDown(nFlags, point);
}
```

8. Откройте окно редактирования файла TextDoc.cpp и после строки `#include "Text.h"` вставьте строку

   ```
   #include <io.h>
   ```

9. Измените конструктор и деструктор класса `CTextDoc` в соответствии с текстом листинга 7.5.

Листинг 7.5. Конструктор и деструктор класса `CTextDoc`

```
// Конструктор и деструктор класса CTextDoc

CTextDoc::CTextDoc()
: lpClipFile(NULL)
, lpOutFile(NULL)
{
  struct  _finddata_t  c_file;
  long      hFile;
  CString    File_Name = "C:\\Файл пользовательского буфера обмена. Не уничтожайте этот файл";

  // Проверка наличия временного файла на диске
  if((hFile = _findfirst(File_Name, &c_file)) != -1L )
  {
    ::MessageBox(0, "Данное приложение не может иметь двух файлов буфера обмена", "Ошибка", MB_OK | MB_SYSTEMMODAL);
    exit(1);
  }
  _findclose(hFile);

  // Создание временного файла
  try
  {
    lpClipFile  = new CFile(File_Name, CFile::modeCreate  |
        CFile::modeReadWrite  |  CFile::shareDenyNone);
    SetFileAttributes(File_Name, FILE_ATTRIBUTE_HIDDEN);
    lpOutFile  = NULL;
  }

  // Обработка ошибок при создании временного файла
  catch(...)
  {
```

```
    ::MessageBox(0, "Ошибка при открытии файла буфера обмена",
                    "Ошибка", MB_OK | MB_SYSTEMMODAL);
    exit(1);
  }
}

CTextDoc::~CTextDoc()
{
  // Сохранение имени временного файла
  CString    F_Name = lpClipFile-> GetFilePath();

  // Закрытие временного файла
  lpClipFile-> Close();

  // Закрытие выходного файла
  if(lpOutFile)
    lpOutFile-> Close();

  // Уничтожение объектов файлов
  delete  lpClipFile;
  delete  lpOutFile;

  // Удаление с диска временного файла
  CFile::Remove(F_Name);
}
```

10. Откройте окно редактирования файла Text.cpp и уничтожьте в функции `CText::InitInstance` следующие строки:

    ```
    // Dispatch commands specified on the command line. Will return FALSE if
    // app was launched with /RegServer, /Register, /Unregserver or /Unregister.
    if (!ProcessShellCommand(cmdInfo))
      return FALSE;
    ```

11. Нажмите клавишу <F5> и запустите приложение на исполнение.
12. Выберите команду меню **File | Open** (Файл | Открыть) или нажмите кнопку **Open** (Открыть) в панели инструментов.
13. В окне списка файлов появившегося диалогового окна **Open** (Открыть) выделите имя файла TextApp1.tpf, созданного на предыдущем этапе рассмотрения данного приложения, и нажмите кнопку **Open** (Открыть).

14. В приложении откроется новое окно документа, в котором будет содержаться информация, хранившаяся в файле TextApp1.tpf.
15. Несколько раз щелкните в нем левой кнопкой мыши. В окне появятся координаты новых точек.
16. Щелкните правой кнопкой мыши в окне документа. Появится диалоговое окно **Save As** (Сохранить как). Постарайтесь сделать короткий щелчок. противном случае, если диалоговое окно **Save As** (Сохранить как) появится на месте щелчка, поверх него будет раскрыто контекстное меню, как это показано на рис. 7.9.

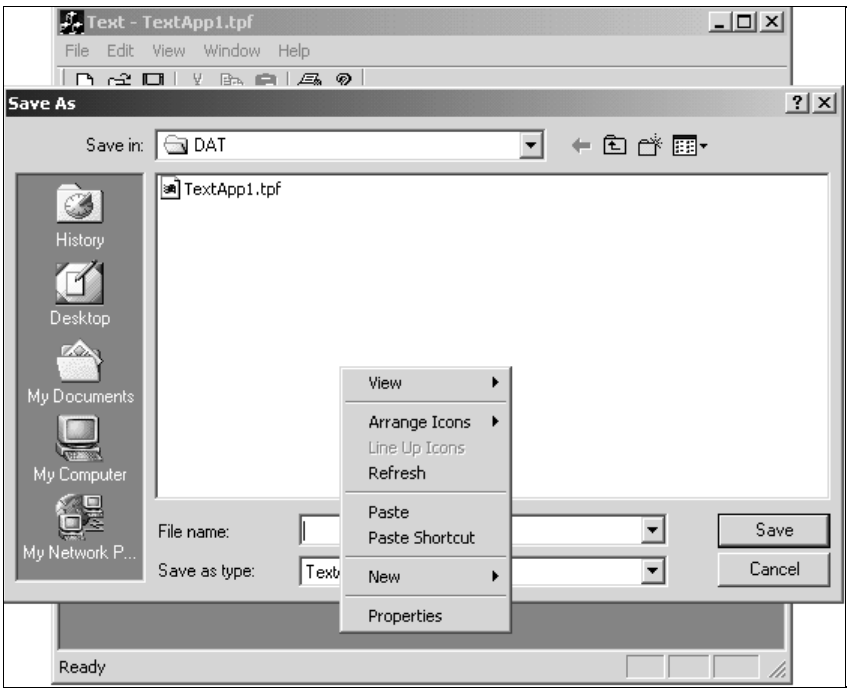


Рис. 7.9. Диалоговое окно **Save As**

17. Введите в текстовое поле **File name** (Имя файла) имя файла TextApp2 и нажмите кнопку **Save** (Сохранить). Из диалогового окна исчезнет вся информация, введенная в него в последнем сеансе пользователем.
18. Закройте окно документа и снова выберите команду меню **File | Open** (Файл | Открыть).
19. В окне списка файлов появившегося диалогового окна **Open** (Открыть) выделите имя файла TextApp2.tpf и нажмите кнопку **Open** (Открыть). В окне документа появятся изменения, внесенные в документ в последнем сеансе работы с ним.

Модификация функции `OnLButtonDown` заключается в том, что в нее добавлены операторы записи координат точек в файл пользовательского буфера обмена. Для этого используется функция `CFile::Write`, первый аргумент которой содержит указатель на созданный пользователем буфер, содержимое которого будет записано в файл, а второй ее аргумент содержит размер этого буфера в байтах.

В функцию `OnRButtonDown` внесены более серьезные изменения, поскольку на нее возлагается задача сохранения информации, внесенной пользователем в документ в текущем сеансе работы.

Прежде всего, данная функция выводит на экран стандартное диалоговое окно **Save As** (Сохранить как). Для этого она вызывает функцию `CDocManager::DoPromptFileName`. Переменная `m_pDocManager`, содержащая указатель на объект класса `CDocManager`, является членом класса `CWinApp`. Для получения указателя на связанный с данным приложением объект класса `CWinApp` используется глобальная функция `AfxGetApp`. Функция `CDocManager::DoPromptFileName` не документирована. Первым ее аргументом является ссылка на объект класса `CString`, в который будет записано имя файла, выбранное в вызываемом диалоговом окне. Вторым аргументом — идентификатор строкового ресурса заголовка диалогового окна. Заголовок диалогового окна **Open** (Открыть) содержится в строковом ресурсе `AFX_IDS_OPENFILE`, а заголовок диалогового окна **Save As** (Сохранить как) — в строковом ресурсе `AFX_IDS_SAVEFILE`. В третьем аргументе данной функции содержатся флаги, определяющие вид и поведение диалогового окна. Наиболее часто используемые значения этих флагов приведены ниже:

- `OFN_HIDEREADONLY` — в диалоговом окне не отображаются файлы, имеющие атрибут "только для чтения";
- `OFN_PATHMUSTEXIST` — выдает сообщение об ошибке в случае задания в диалоговом окне несуществующего каталога;
- `OFN_FILEMUSTEXIST` — выдает сообщение об ошибке в случае задания в диалоговом окне несуществующего файла.

Четвертый аргумент данной функции имеет тип `BOOL` и определяет тип выводимого диалогового окна. Если этот аргумент имеет значение `TRUE`, то на экран выводится диалоговое окно **Open** (Открыть), в противном случае выводится диалоговое окно **Save As** (Сохранить как). Последний, пятый, аргумент данной функции представляет собой указатель на объект класса `CDocTemplate`, который должен использоваться при открытии файлов. Если этот аргумент имеет ненулевое значение, то в текстовом поле раскрывающегося списка **Files of type** (Тип файлов) (в диалоговом окне **Open** (Открыть)) или **Save as type** (Тип сохраняемого файла) (в диалоговом окне **Save As** (Сохранить как)) выводится фильтр данного шаблона. Кроме указанного фильтра данный раскрывающийся список будет содержать только универсальный фильтр **All Files ("*.*)** (Все файлы). Если пятый аргумент имеет нулевое значение, то в текстовое поле раскрывающихся списков **Files of type** (Тип файлов) или **Save as type** (Тип сохраняемого файла) выводится фильтр первого зарегистрированного шаблона документа, а сами раскрывающиеся списки содержат фильтры всех зарегистрированных шаблонов документов и универсальный фильтр *(см. главу 2)*.

Если пользователь при работе с диалоговым окном **Save As** (Сохранить как) нажал кнопку **Save** (Сохранить), функция `CDocManager::DoPromptFileName` возвращает ненулевое значение, а в ее первом аргументе содержится имя файла, выбранное пользователем в диалоговом окне **Save As** (Сохранить как). Из этого имени выделяется текущий каталог и сохраняется в системном реестре, как это было описано при рассмотрении функции `Serialize`.

После внесения изменений в системный реестр функция `OnRButtonDown` создает объект класса `CFile` и запоминает указатель на этот объект в переменной `lpOutFile`. В случае возникновения ошибок при создании объекта данного класса его конструктор вызывает исключение. При обработке данного исключения вызывается глобальная функция `MessageBox`, выводящая сообщение об ошибке, и работа данного приложения прерывается вызовом функции `exit`.

В случае успешного завершения операции по созданию объекта класса `CFile` в него записывается количество точек, содержащихся в пользовательском буфере обмена. При этом для файла пользовательского буфера обмена вызывается функция `CFile::GetLength`, позволяющая получить логическую длину файла в байтах. Эта величина делится на размерность элемента данных и на их количество в структуре.

Поскольку при записи информации в файл текущая позиция находится в конце файла, для файла пользовательского буфера обмена вызывается функция `CFile::SeekToBegin`, устанавливающая текущую позицию в начало файла.

После этого производится копирование информации из файла пользовательского буфера обмена в выходной файл. Для этого с использованием функции `CFile::Read` производится чтение из файла пользовательского буфера обмена и последующая его запись в выходной файл с использованием функции `CFile::Write`. Поскольку функция `CFile::Read` возвращает количество прочитанных ею байтов, то процесс копирования будет продолжаться до тех пор, пока данная функция не возвратит нулевое значение, т. е. пока не будет скопирован весь файл пользовательского буфера обмена.

После этого работа функции `OnRButtonDown` аналогична работе ее исходной версии.

В конструкторе класса `CTextDoc` производится инициализация указателей на объекты класса `CFile`. Прежде всего, в конструкторе данного класса проверяется, не существует ли на диске файл с именем, используемым для пользовательского буфера обмена. Если такой файл существует, то пользователю выдается сообщение об этом и приложение завершает свою работу. Необходимость данной проверки связана с тем, что файл пользовательского буфера обмена создается в объекте класса `CTextDoc`, который не является уникальным для данного приложения. Наличие нескольких буферов обмена, использующих для хранения информации один и тот же файл, может привести к некорректной работе приложения, не говоря уже о том, что операционная система не позволяет уничтожать открытые файлы.

В реальном приложении, использующем один пользовательский буфер обмена, соответствующий объект класса `CFile` обычно создается и уничтожается в клас-

се `CMainFrame` или в классе приложения, что гарантирует его уникальность для данного приложения. При этом можно не производить данную проверку на существование файла.

Имя данного файла должно быть достаточно экстравагантным, чтобы его случайно не использовал пользователь. Для этого можно применить длинное имя файла, включив в него знаки препинания (например, запятые). Данное имя должно содержать хотя бы одну точку, но в конце данного имени ставить точку нежелательно, поскольку символы, стоящие после последней точки, будут рассматриваться как расширение файла. Таким образом, данный файл будет иметь не только экстравагантное имя, но и экстравагантное расширение.

В том случае, если на диске отсутствует файл с именем, зарезервированным для пользовательского буфера обмена, приложение создает объект класса `CFile` с использованием данного имени и, с использованием глобальной функции `SetFileAttributes`, устанавливает для данного файла атрибут скрытого файла. Использование скрытого файла для пользовательского буфера обмена повышает "дуракоустойчивость" приложения, поскольку у пользователя не возникнет вопроса: "Что это за файл появился в его каталоге?", и желания его уничтожить.

В деструкторе класса `CTextDoc` производится уничтожение объектов класса `CFile`, созданных в конструкторе данного класса. Хотя мы сами задали имя файла пользовательского буфера обмена и можем просто присвоить его соответствующей переменной, для его получения в деструкторе используется функция `CFile::GetFilePath`, возвращающая полный путь к файлу. Это особенно важно в том случае, если при задании имени файла пользовательского буфера обмена использовалось только имя файла с расширением. В этом случае, при изменении пользователем текущего каталога в процессе работы с приложением, при использовании функции `GetFileName` приложение не сможет уничтожить данный файл, поскольку будет искать его в другом каталоге.

После этого файл пользовательского буфера обмена закрывается с использованием функции `CFile::Close`, но связанный с ним объект не уничтожается. Если для файла пользовательского буфера обмена заранее известно, что он был создан, то для выходного файла такая информация отсутствует. Если объект класса `CFile` не был создан, то любые операции с ним приводят к возникновению ошибок, поэтому перед вызовом функции `CFile::Close` для данного объекта производится проверка на существование самого объекта. После закрытия файлов соответствующие объекты класса `CFile` уничтожаются.

После закрытия файла пользовательского буфера обмена и уничтожения соответствующего ему объекта класса `CFile` производится удаление данного файла с диска. Для этого вызывается функция `CFile::Remove`, в качестве аргумента которой передается имя уничтожаемого файла. Смысл включения в класс функции, которая может работать только после прекращения работы с объектом данного класса и не использующей хранящейся в нем информации, понятен только особо просвещенным специалистам фирмы Microsoft.

Удаление операторов из функции `CText::InitInstance` связано с тем, что данные операторы выводили на экран пустое окно документа, создавая при этом

файл пользовательского буфера обмена. Это затрудняло дальнейшую работу с приложением.

Работа с системным реестром

В 16-разрядных версиях Windows для хранения настроек приложения использовались специальные *файлы инициализации*, обычно имевшие расширение ini. Работа с ними доставляла много хлопот программисту, поскольку при переносе приложения с одной машины на другую приходилось следить за тем, чтобы все связанные с ним файлы инициализации, обычно записывавшиеся в системный каталог, были полностью перенесены на новый компьютер. В 32-разрядных версиях Windows приложение хранит всю необходимую для его запуска информацию в *системном реестре.* Теперь приложение, перенесенное на другой компьютер, вместо того, чтобы искать по всему компьютеру связанные с ним файлы инициализации и "жаловаться" на их отсутствие, при своем первом запуске использует информацию, заданную по умолчанию, а затем вносит соответствующие записи в системный реестр.

В прежних версиях Windows файлы инициализации представляли собой обычные текстовые файлы, в которые можно было внести изменения с помощью любого текстового редактора, что было сопряжено с определенным риском. В отличие от них, файл системного реестра можно редактировать только с использованием специального редактора реестра. Исполняемый файл данного редактора имеет имя RegEdit.exe и расположен в главной папке Windows. Для его запуска достаточно нажать кнопку **Start** (Пуск), расположенную на Панели задач, и в раскрывшемся меню выбрать команду **Run** (Выполнить). Появится диалоговое окно **Run** (Выполнить), изображенное на рис. 7.10.

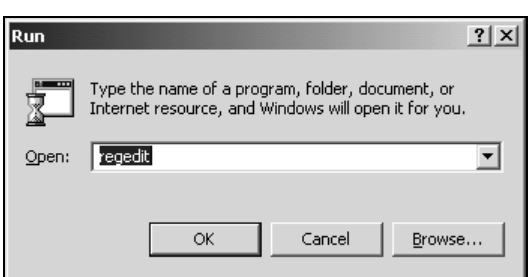


Рис. 7.10. Диалоговое окно запуска программы

Введите в текстовое поле **Open** (Открыть) имя исполняемого файла "regedit", как это показано на рис. 7.10 и нажмите кнопку **OK**. Появится окно **Registry Editor** (Редактор реестра), изображенное на рис. 7.11.

В левой панели окна приведены все предопределенные ключи реестра. Каждый ключ имеет несколько уровней иерархии, которые при желании можно после-

довательно разворачивать. Если в левой панели выделен терминальный элемент дерева, то в правой панели выводится хранящаяся по этому идентификатору информация.

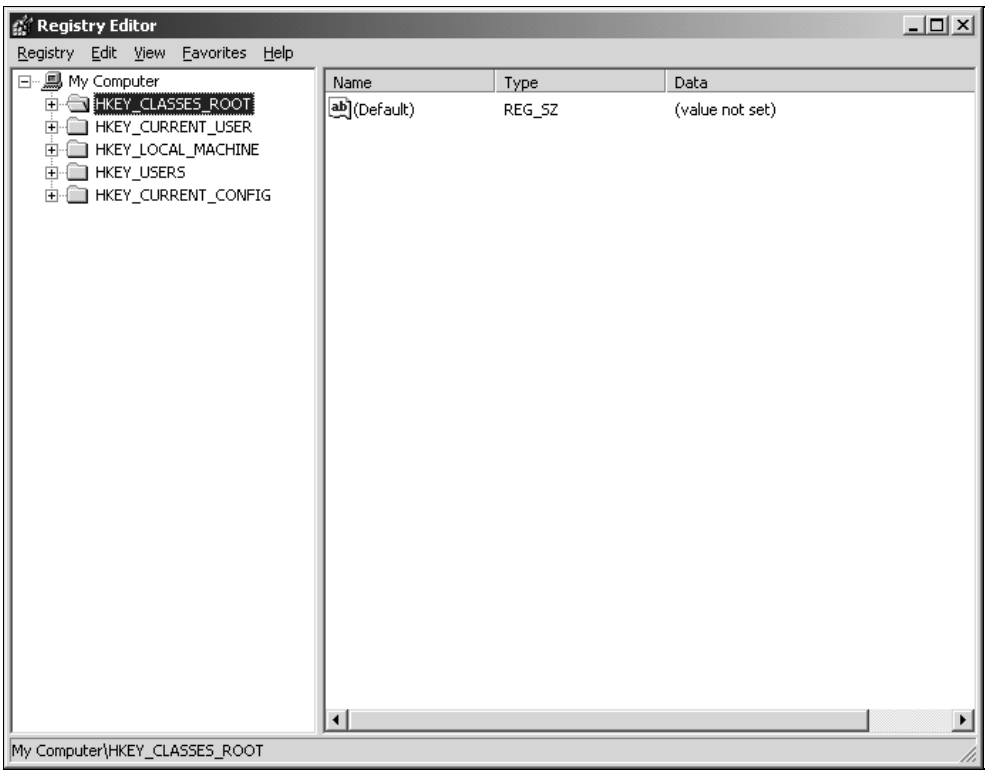


Рис. 7.11. Окно **Registry Editor**

На верхнем уровне иерархии могут располагаться шесть предопределенных ключей, пять из которых приведены на рис. 7.11. Каждому из них соответствует определенная информация:

- **HKEY_CLASS_ROOT** — содержит информацию о типах документов и их свойствах, а также информацию о классах и различных приложениях, установленных на компьютере. Здесь, помимо прочего, хранится информация о том, какое приложение обрабатывает файлы с каждым из зарегистрированных в данной системе расширений;
- **HKEY_CURRENT_USER** — содержит информацию обо всех системных установках, произведенных пользователем. Здесь также хранятся все параметры настройки создаваемых пользователем приложений;
- **HKEY_LOCAL_MACHINE** — содержит информацию о данном компьютере;

- **HKEY_USERS** — содержит информацию обо всех пользователях, зарегистрированных в системе, и о конфигурации системы по умолчанию;
- **HKEY_CURRENT_CONFIG** — содержит информацию о конфигурации аппаратных средств системы;
- **HKEY_DYN_DATA** — служит для хранения динамических данных реестра, т. е. быстро изменяющуюся информацию.

На рис. 7.12 приведен системный реестр с раскрытой папкой **CONTROL**, в которой хранится приложение.

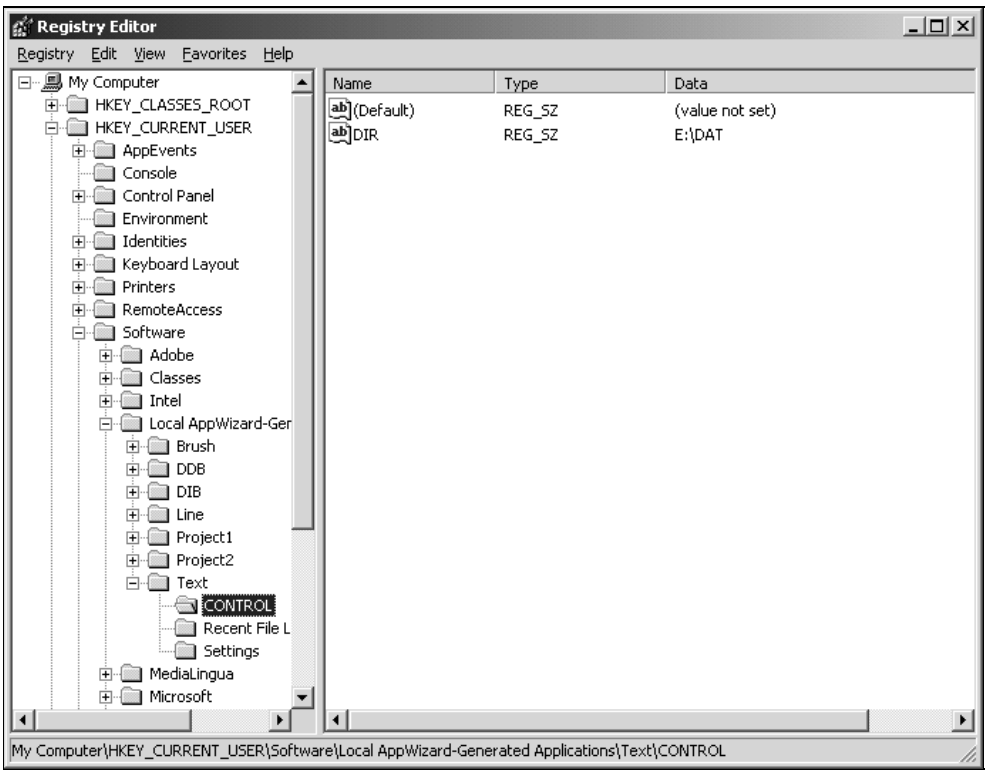


Рис. 7.12. Раскрытая папка **CONTROL**

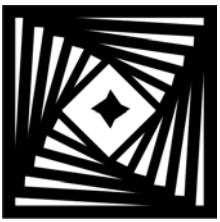
Для того чтобы приложение могло получить доступ к системному реестру в функции `InitInstance` пользовательского класса, производного от класса `CWinApp`, необходимо вызвать функцию `SetRegistryKey`, в качестве аргумента которой следует указать индивидуальный ключ приложения. Этот ключ отображается в окне **Registry Editor** (Редактор реестра) в виде папки, расположенной в папке **Software**, которая, в свою очередь, расположена в папке **HKEY_CURRENT_USER**. В этой папке содержится вся информация, записы-

ваемая в системный реестр приложением. По умолчанию мастер AppWizard использует в качестве ключа строку "Local AppWizard-Generated Applications" (локальные приложения, созданные мастером AppWizard). Если пользователь не изменил ключа, установленного по умолчанию, то для каждого из приложений, использующих данный ключ, создается отдельная папка с именем данного приложения, в которую и записываются ее установки. Однако при создании серьезных приложений рекомендуется изменить устанавливаемый по умолчанию ключ и использовать вместо него, например, имя вашей фирмы.

После установки ключа приложения следует вызвать функцию `LoadStdProfileSettings`, загружающую стандартные опции приложений, включая список последних файлов, с которыми работало данное приложение. В качестве аргумента данной функции указывается размерность списка последних файлов, с которыми работало данное приложение, расположенного в меню **File** (Файл). Существенным недостатком данного списка является то, что файлы в нем приводятся без расширений. Это очень неудобно в том случае, если приложение может работать с документами, характеризующими один и тот же объект с разных сторон. При этом естественно давать файлам, относящимся к одному и тому же объекту, одно и то же имя и различать их по расширениям. В этом случае использование списка файлов в меню **File** (Файл) будет похоже на игру в русскую рулетку: пользователь будет выбирать между несколькими одинаковыми именами файлов в списке и пытаться с первого раза угадать, какой из них имеет нужное ему расширение.

Для записи информации в системный реестр и чтения оттуда информации используются специальные функции, являющиеся членами класса `CWinApp`. Функция `WriteProfileString` позволяет записать в системный реестр строку текста, а функция `WriteProfileInt` позволяет записать туда целое число. Для извлечения данной информации из реестра используются функции `GetProfileString` и `GetProfileInt` соответственно.

Глава 8



Работа с текстовыми документами

Как уже неоднократно отмечалось в этой книге, специалисты Microsoft считают текстовые документы единственным достойным объектом для своей работы. Вся операционная система Windows разрабатывалась именно для работы с данным типом документов. Поэтому, рассматривая Visual C++, нельзя обойти молчанием эту область его применения.

Вследствие своей любви к написанию редакторов создатели Visual C++ самостоятельно разработали два типа редакторов, обеспечив их всеми необходимыми, с их точки зрения, функциями и, посчитав свои творения совершенством, практически исключили всякое вмешательство в их работу со стороны пользователя. Эти редакторы реализованы в объектах классов `CEdit` и `CRichEditView`. Первый из них представляет собой простейший редактор, используемый в текстовых полях диалоговых окон. Данный редактор не предусматривает форматирования текста и использование различных шрифтов. Все эти возможности имеются в текстовом редакторе, реализованном в классе `CRichEditView`. Возможности, предоставляемые этим редактором, наглядно продемонстрированы в текстовом редакторе WordPad, являющемся примером использования данного класса, текст которого можно найти в библиотеке MSDN.

Создание простейшего текстового редактора

Простейший текстовый редактор, использующий возможности класса `CEdit`, реализован в приложении EditApp, текст которого расположен в одноименной папке на дискете, прилагаемой к данной книге. Чтобы самостоятельно создать данное приложение:

1. Выполните все операции, указанные в главе 1 для создания многооконного приложения с именем EditApp, но в диалоговом окне **MFC Application Wizard - EditApp** (Мастер создания приложений MFC) не нажимайте кнопку **Finish** (Готово).
2. Вместо этого, раскройте в нем вкладку **Generated Classes** (Создаваемые классы), как это показано на рис. 8.1.
3. В раскрывающемся списке **Base class** (Базовый класс) выделите имя базового класса `CEditView` и только после этого нажмите кнопку **Finish** (Готово).

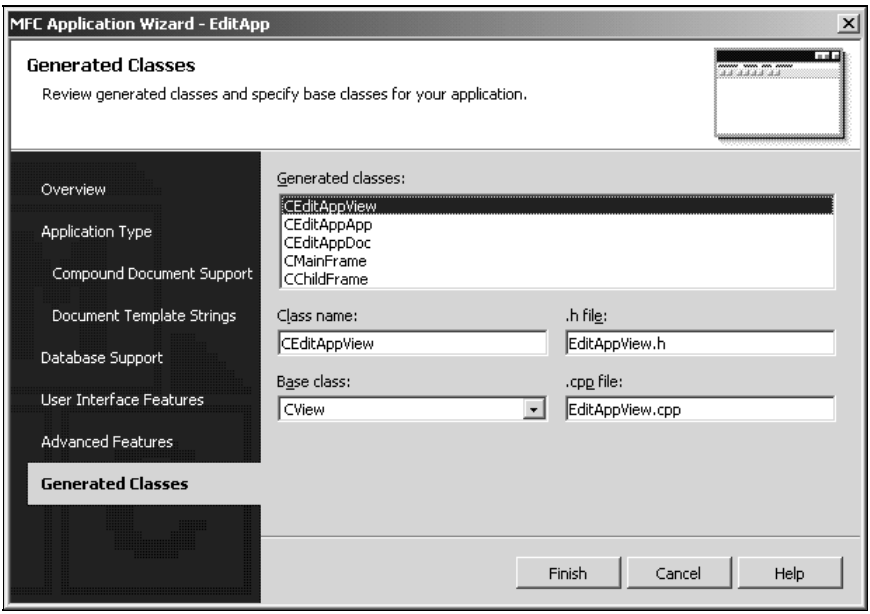


Рис. 8.1. Диалоговое окно **MFC Application Wizard - EditApp**

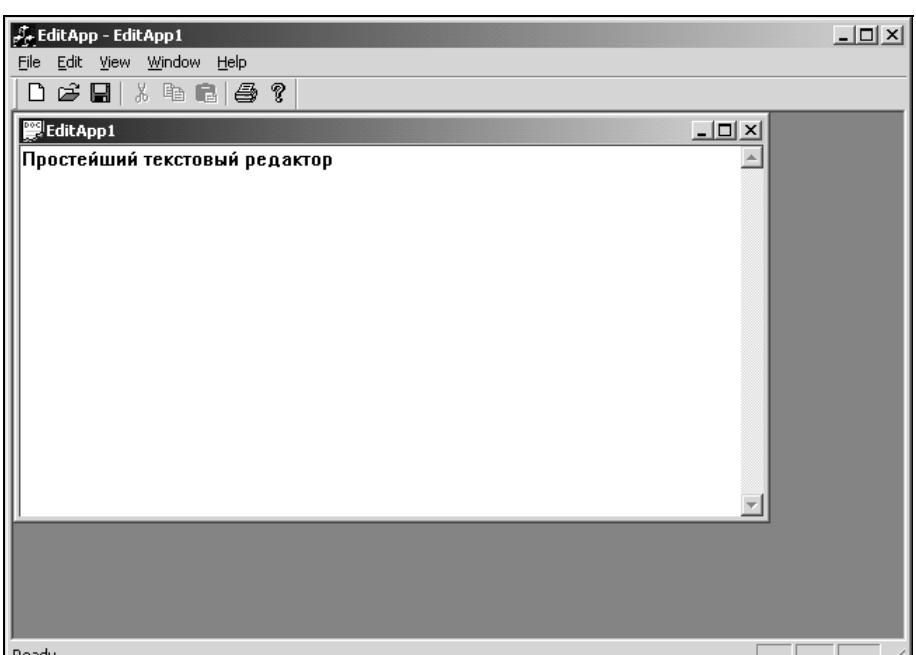


Рис. 8.2. Простейший текстовый редактор

4. Нажмите клавишу <F5> и запустите приложение на исполнение. Открывшееся в приложении окно будет окном текстового редактора, как это показано на рис. 8.2.

Данное приложение является законченным текстовым редактором, позволяющим вводить текст, вырезать или копировать его в буфер обмена и вставлять текст, содержащийся в буфере обмена, в любое место окна, указанное текстовым курсором. Для этого используются команды меню **Edit | Cut** (Правка | Вырезать) **Edit | Copy** (Правка | Копировать) и **Edit | Paste** (Правка | Вставить) и соответствующие им кнопки панели инструментов. В созданном окне текстового редактора автоматически появляются полосы прокрутки, причем, в отличие от обычного окна, они не исчезают в том случае, когда весь текст помещается в окно, а становятся недоступными. Кроме того, в нем реализованы возможности печати и предварительного просмотра печати.

Создание более сложного редактора

Чтобы более подробно ознакомиться с возможностями, предоставляемыми классом `CEdit`, создадим приложение, позволяющее находить слова в тексте и автоматически прокручивать текст в процессе его чтения. Чтобы внести в приложение EditApp соответствующие изменения:

1. Раскройте окно **Resource View** (Просмотр ресурсов) и последовательно раскройте папки **EditApp.rc** и **Menu** (Меню).
2. Дважды щелкните левой кнопкой мыши по значку `IDR_EditAppTYPE`. Откроется окно редактирования ресурса меню. Обратите внимание на то, что в ресурсе меню присутствует подчеркивание символов, используемых для вызова команд меню, которое отсутствует в самом меню.
3. Раскройте меню **Edit** (Правка) и щелкните левой кнопкой мыши на расположенном в его нижней части прямоугольнике, содержащем надпись "Type Here" (Вводи сюда).
4. Щелкните правой кнопкой мыши на выделенном прямоугольнике и выберите в появившемся контекстном меню команду **Insert Separator** (Вставить разделитель). В заготовке раскрывающегося меню появится разделитель.
5. Щелкните левой кнопкой мыши по прямоугольнику с надписью "Type Here", расположенному под разделителем, введите в него текст "&Find" и нажмите клавишу <Enter>.
6. Щелкните правой кнопкой мыши на только что созданной команде меню и выберите в раскрывшемся контекстном меню команду **Properties** (Свойства). Раскроется окно **Properties** (Свойства).
7. Введите в текстовое поле раскрывающегося списка **ID** (Идентификатор ресурса) идентификатор ресурса `ID_SEEK`, а в текстовое поле **Prompt** (Подсказка) — текст "Finds the specified text\nFind".
8. Щелкните правой кнопкой мыши по папке **Dialog** (Диалог), расположенной в папке **EditApp.rc** окна **Resource View** (Просмотр ресурсов).

9. В появившемся контекстном меню выберите команду **Insert Dialog** (Вставить диалог). В окне редактирования ресурса появится заготовка диалогового окна.
10. Раскройте папку **Dialog** (Диалог) и щелкните левой кнопкой мыши по значку **IDD_DIALOG1**.
11. Введите в текстовое поле раскрывающегося списка **ID** (Идентификатор ресурса) окна **Properties** (Свойства) идентификатор ресурса `IDD_SEEK`.
12. Щелкните левой кнопкой мыши по свободному полю заготовки диалогового окна и введите в текстовое поле **Caption** (Заголовок) окна **Properties** (Свойства) заголовок окна "Find".
13. Переместите кнопки **OK** и **Cancel** (Отмена), расположенные в заготовке диалогового окна, в его нижнюю часть.
14. В окне **Toolbox** (Инструментарий) выберите элемент управления **Edit Box** (Окно редактора) и перетащите его в заготовку диалогового окна, поместив его над кнопками.
15. Введите в текстовое поле **ID** (Идентификатор ресурса) окна **Properties** (Свойства) идентификатор ресурса `IDC_SAMPLE`.

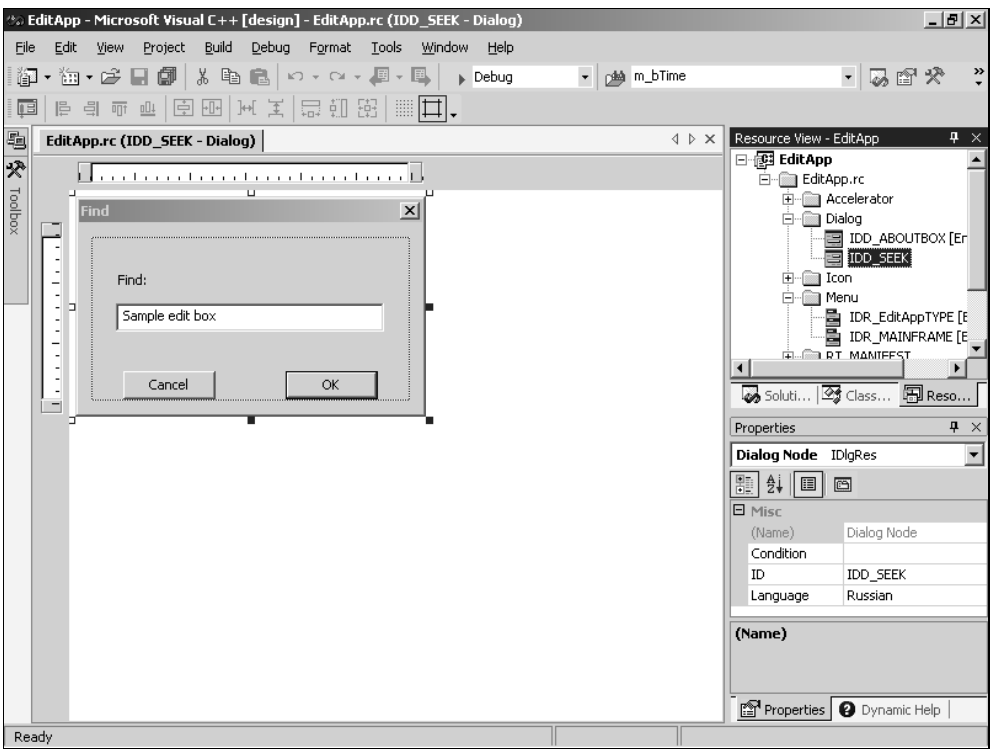


Рис. 8.3. Заготовка диалогового окна

16. В окне **Toolbox** (Инструментарий) выберите элемент управления **Static Text** (Статический текст) и перетащите его в заготовку диалогового окна, поместив его над текстовым полем.
17. Введите в текстовое поле **Caption** (Заголовок) окна **Properties** (Свойства) текст "Find:". В результате этих операций заготовка диалогового окна примет вид, изображенный на рис. 8.3.
18. Щелкните правой кнопкой мыши в окне редактирования ресурса диалогового окна и выберите в раскрывшемся контекстном меню команду **Add Class** (Добавить класс). Появится диалоговое окно **MFC Class Wizard - EditApp** (Мастер создания классов MFC), изображенное на рис. 8.4.
19. В текстовое поле **Class name** (Имя класса) введите имя нового класса `CSeekDlg` и в раскрывающемся списке **Base class** (Базовый класс) выделите имя базового класса `CDialog`.
20. Нажмите кнопку с тремя точками около текстового поля **.h file**. Появится диалоговое окно **VS Wizards Select File**, изображенное на рис. 8.5.

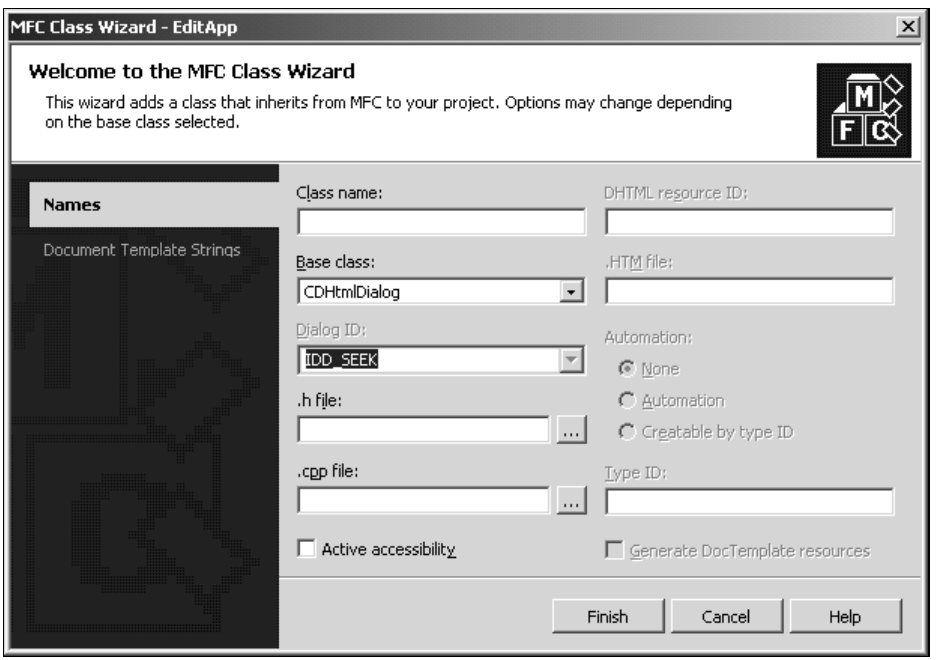


Рис. 8.4. Диалоговое окно **Add Class Wizard - EditApp**

21. В окне списка файлов выделите имя файла EditAppView.h или другим способом введите имя этого файла в текстовое поле раскрывающегося списка **File name** (Имя файла) и нажмите кнопку **Save** (Сохранить).

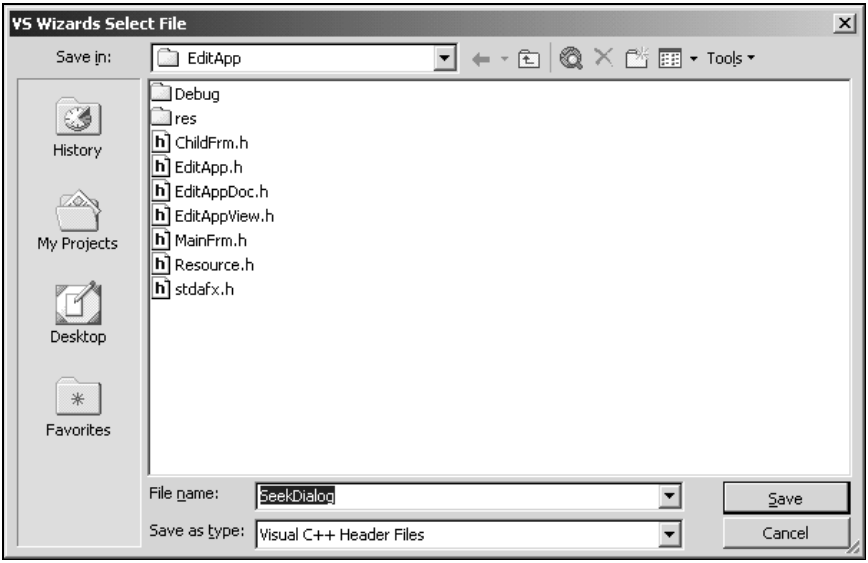


Рис. 8.5. Диалоговое окно **VS Wizards Select File**

22. Повторите п.п. 20 и 21 для текстового поля **.cpp file**, введя в него имя файла EditAppView.cpp.
23. Нажмите кнопку **Finish** (Готово). Появится окно сообщения, изображенное на рис. 8.6, предупреждающее о том, что файл EditAppView.h уже существует и спрашивающее, следует ли включать заголовок создаваемого класса в этот файл.

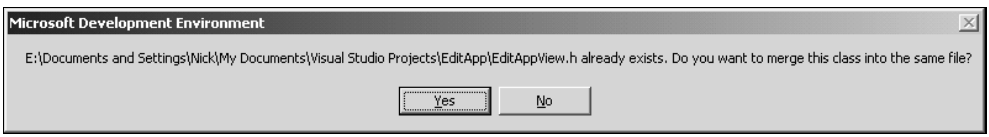


Рис. 8.6. Диалоговое окно **Microsoft Development Environment**

24. Нажмите кнопку **Yes** (Да). Появится аналогичное окно сообщения, ставящее тот же вопрос относительно файла EditAppView.cpp.
25. Нажмите кнопку **Yes** (Да). Диалоговое окно **MFC Class Wizard** (Мастер создания классов MFC) закроется.
26. Откройте окно **Class View** (Просмотр класса), а в нем раскройте папку **EditApp**.
27. Щелкните правой кнопкой мыши на папке `CSeekDlg` и выберите в раскрывшемся контекстном меню команду **Add | Add Variable** (Добавить | Доба-

вить переменную). Появится диалоговое окно **Add Member Variable Wizard** (Мастер добавления переменной в класс).

28. Установите флажок **Control variable** (Связь с элементом управления) и в ставшем после этого доступном раскрывающемся списке **Control ID** (Идентификатор элемента управления) выделите идентификатор ресурса `IDC_SAMPLE`.
29. В раскрывающемся списке **Category** (Категория) выделите строку **Value** (Переменная), введите в текстовое поле **Variable name** (Имя переменной) идентификатор переменной `m_Sample` и нажмите кнопку **Finish** (Готово). Диалоговое окно **Add Member Variable Wizard** (Мастер добавления переменной в класс) закроется.
30. Щелкните правой кнопкой мыши на папке `CEditAppView` и выберите в раскрывшемся контекстном меню команду **Properties** (Свойства). Раскроется окно **Properties** (Свойства).
31. В окне **Properties** (Свойства) нажмите кнопку **Messages** (Сообщения). В окне раскроется список сообщений, обрабатываемых данным классом.
32. В открывшемся списке выделите сообщение `WM_LBUTTONDBLCLK`, раскройте связанный с ним список и выделите в нем единственную строку. В текстовом поле раскрывающегося списка появится имя функции обработки данного сообщения, откроется окно редактирования файла EditAppView.cpp и текстовый курсор будет помещен в заготовку новой функции.
33. В окне **Properties** (Свойства) нажмите кнопку **Events** (События). Раскроется список событий, обрабатываемых данным классом.
34. Раскройте папку **ID_SEEK** и выделите в ней сообщение `COMMAND`. Добавьте функцию обработки данного сообщения аналогично тому, как это было сделано в п. 32.
35. В окне редактирования файла EditAppView.cpp перенесите созданные нами функции обработки сообщений в область функций класса `CEditAppView` и измените их в соответствии с текстом листинга 8.1.

Листинг 8.1. Функция `CEditAppView::OnSeek`

```
// Функции обработки сообщений класса CEditAppView

void CEditAppView::OnLButtonDblClk(UINT nFlags, CPoint point)
{
  // Получение ссылки на объект класса текстового поля
  CEdit&  edit = GetEditCtrl();

  // Получение позиции символа
  UINT  n = 0xffff & edit.CharFromPos(point);
  while(n < (int) GetBufferLength())
```

```
  {
    n++;
    edit.SetSel(n, n, FALSE);  // Перемещение курсора
    Sleep(200);
  }
  CEditView::OnLButtonDblClk(nFlags, point);
}

// Обработка сообщения о выборе команд Edit, Find
void CEditAppView::OnSeek(void)
{
  CSeekDlg  dlg(this);  // Создание объекта класса
                        // диалогового окна
  if(dlg.DoModal() == IDOK)  // Вывод диалогового окна на экран
  {
    FindText(dlg.m_Sample);  // Поиск по образцу
  }
}
```

36. Нажмите клавишу <F5> и запустите приложение на исполнение.
37. Введите в редактор текст с таким расчетом, чтобы в окне появилась вертикальная полоса прокрутки.
38. Перейдите в начало введенного текста и дважды щелкните левой кнопкой мыши в первой или второй строке.
39. В месте щелчка появится текстовый курсор, который начнет перемещаться по тексту. Как только он дойдет до последней строки в окне, оно автоматически прокрутится, чтобы показать следующую строку. Как только курсор дойдет до конца текста, он автоматически переместится в то место на экране, где был произведен щелчок левой кнопкой мыши. Если в этом месте на экране отсутствует текст, то курсор переместится в конец первой строки на экране.
40. Выберите команду меню **Edit | Find** (Правка | Найти). Появится диалоговое окно **Find** (Найти).
41. Введите в текстовое поле последовательность символов, которую необходимо найти в тексте, и нажмите кнопку **OK**. Если эта последовательность имеется в тексте после текущей позиции курсора, она будет найдена и выделена блоком. В противном случае курсор останется на прежнем месте.

Данное приложение демонстрирует возможности управления текстовым курсором в редакторе. Первым оператором функции `OnLButtonDown` является создание ссылки на объект класса `CEdit`, связанный с данным объектом класса приложения. Для этого используется функция `CEditView::GetEditCtrl`.

Далее, с использованием полученной ссылки на объект класса `CEdit`, вызывается функция `CEdit::CharFromPos`, позволяющая получить индекс ближайшего к данной точке символа, и индекс строки, в которой находится данный символ. Формат возвращаемой данной функцией величины указывает на то, что объем текста в данном редакторе не может превысить 65 534 символ, что явно недостаточно для любого серьезного редактора.

Поскольку интересующая нас величина находится в младшем слове, то возвращаемое значение функции `CharFromPos` маскируется и записывается в счетчик текущей позиции.

После инициализации процедуры перемещения курсора вызывается цикл его перемещения по экрану. Для того чтобы данная процедура завершилась немедленно по достижении курсором последнего символа в тексте, счетчик текущей позиции постоянно сравнивается с числом символов в тексте. Для получения этой информации вызывается функция `CEditView::GetBufferLength`.

Для реализации возможности поиска текста в редакторе в меню была введена специальная команда, а для задания искомого шаблона создано специальное диалоговое окно. Программирование всех необходимых для работы с диалоговым окном операций взяла на себя среда программирования Visual C++, поэтому единственным изменением, которое требуется внести в программу, является заполнение заготовки функции `OnSeek` соответствующим программным текстом.

Первым оператором данной функции является создание объекта класса диалогового окна, в которое будет вводиться искомый текст. После этого вызывается функция `CDialog::DoModal`, осуществляющая вывод диалогового окна на экран и обеспечивающая работу с ним. В том случае, если пользователь нажал кнопку **OK**, вызывается функция `CEditView::FindText`, осуществляющая поиск текста в буфере. Вместо нее можно было бы вызвать функцию `CEditView::OnFindNext` самостоятельно обрабатывающую ситуацию, когда искомый текст не найден, и выдающую об этом звуковой сигнал, но в этом случае пришлось бы указывать три аргумента вместо одного.

Форматирование документов

Как уже говорилось выше, форматирование документов и работа со шрифтами осуществляются в объекте класса `CRichEditView`. Базовый набор его возможностей реализован в приложении WordPad, текст которого можно найти в библиотеке MSDN. Однако это довольно-таки сложное приложение и его описание займет слишком много места. Поэтому здесь будет создано простейшее приложение, позволяющее форматировать текст.

Текст приложения, демонстрирующего возможности класса `CRichEditView` и связанных с ним классов, расположен в папке RichApp дискеты, прилагаемой к данной книге. Чтобы самостоятельно создать данное приложение:

1. Выполните все операции, указанные в *главе 1* для создания многооконного приложения с именем RichApp, но в диалоговом окне **MFC Application Wiz-**

ard - RichApp (Мастер создания приложений MFC) не нажимайте кнопку **Finish** (Готово).

2. Вместо этого раскройте в нем вкладку **Generated Classes** (Создаваемые классы).
3. В раскрывающемся списке **Base class** (Базовый класс) выделите имя базового класса `CRichEditView` и только после этого нажмите кнопку **Finish** (Готово).
4. Нажмите клавишу <F5> и запустите приложение на исполнение.

На первый взгляд оно ничем не будет отличаться от приложения TextApp. Однако в нем появятся дополнительные команды меню. Например, не закрывая данное приложение, вызовите текстовый редактор Word, выделите в нем фрагмент текста и скопируйте его в буфер обмена. После этого перейдите в приложение RichApp и выберите команду **Edit | Paste Special** (Правка | Специальная вставка). Появится диалоговое окно **Paste Special** (Специальная вставка), изображенное на рис. 8.7.

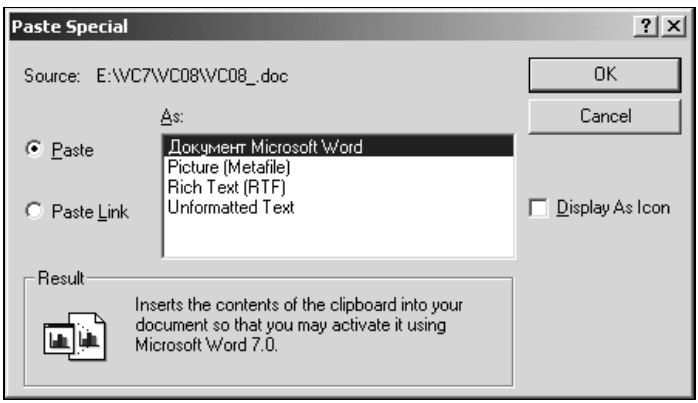


Рис. 8.7. Диалоговое окно **Paste Special**

В этом окне указан источник, из которого был взят фрагмент, расположенный в буфере обмена и предложен формат, в котором его можно вставить в документ данного приложения. Оставьте все установки без изменения и нажмите кнопку **OK**. Вставленный текст будет обведен тонкой рамкой. Дважды щелкните левой кнопкой мыши внутри этой рамки. Результат этого действия показан на рис. 8.8.

Как видно из рис. 8.8 рамка вокруг текста становится "толстой", а вместо меню и панели инструментов приложения появляется меню и панель инструментов Word. Таким образом, данное приложение безо всяких изменений позволяет редактировать документы в формате Word. Того же самого результата можно добиться, вызвав команду меню **Edit | Insert New Object** (Правка | Вставить новый объект), выделив в списке **Object Type** (Тип объекта) диалогового окна **In-**

sert Object (Вставить объект), изображенного на рис. 8.9, строку **Документ Microsoft Word** и нажав кнопку **OK**.

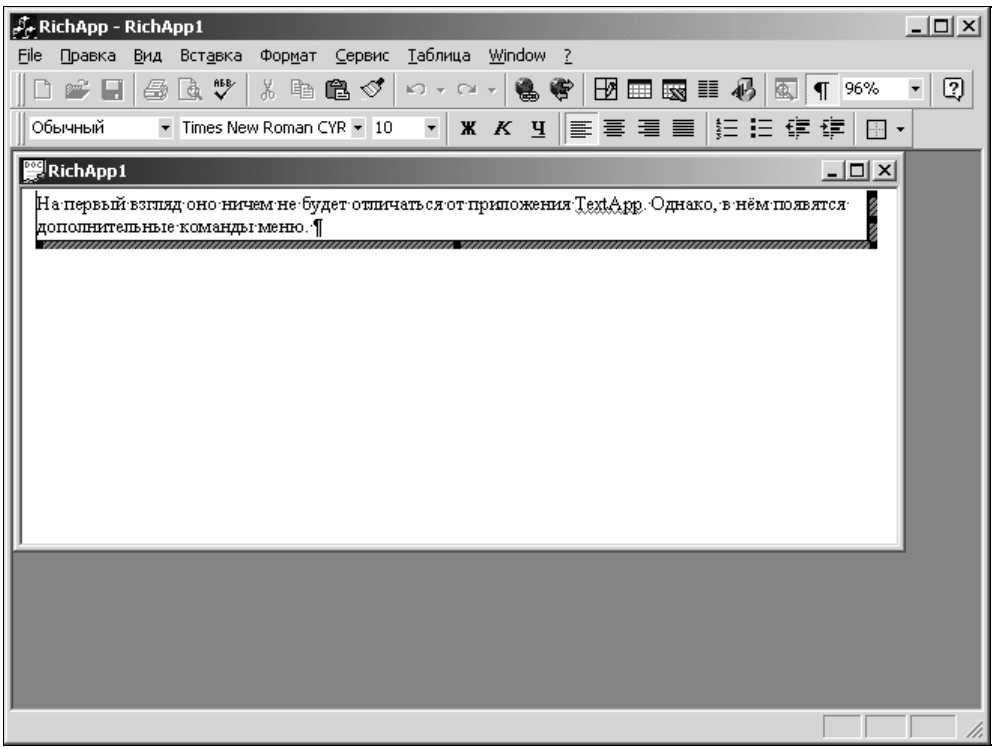


Рис. 8.8. Приложение RichApp с текстом в формате Word

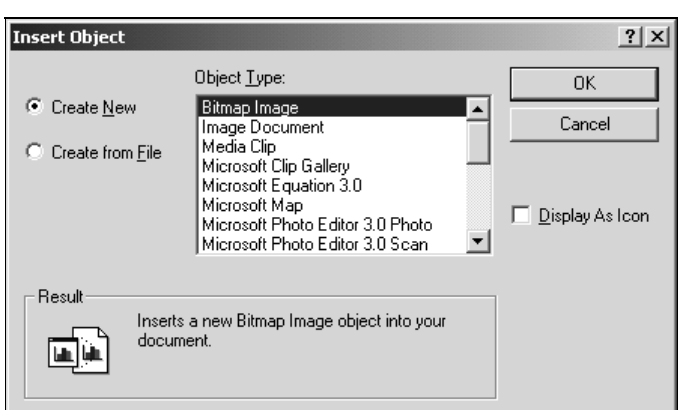


Рис. 8.9. Диалоговое окно **Insert Object**

В этом случае в окне документа сразу появляется пустая "толстая" рамка, готовая для ввода текста.

Однако цель данного раздела не в том, чтобы показать возможности контейнера OLE, а в том, чтобы продемонстрировать способы работы с классом `CRichEditView`.

Чтобы внести в приложение RichApp соответствующие изменения:

1. Раскройте окно **Resource View** (Просмотр ресурсов) и последовательно раскройте папки **EditApp.rc** и **Menu** (Меню).
2. Дважды щелкните левой кнопкой мыши по значку `IDR_RichAppTYPE`. Откроется окно редактирования соответствующего ресурса меню.
3. Щелкните левой кнопкой мыши на прямоугольнике с надписью "Type Here", расположенному в левой части строки меню, и введите в появившееся на его месте текстовое поле заголовок раскрывающегося меню "&Format".
4. Нажмите клавишу <Enter>. Текстовое поле превратится в заголовок меню.
5. Поместите указатель мыши на заголовок раскрывающегося меню **Format** (Формат), нажмите левую кнопку мыши и, не отпуская ее, перетащите этот заголовок, поместив его между пунктами меню **View** (Вид) и **Window** (Окно).
6. Раскройте меню **Format** (Формат), щелкните левой кнопкой мыши на прямоугольнике с надписью "Type Here", введите в появившееся на его месте текстовое поле заголовок "&Left" и нажмите клавишу <Enter>. Текстовое поле превратится в команду меню.
7. Щелкните правой кнопкой мыши на только что созданной команде меню и выберите в появившемся контекстном меню команду **Properties** (Свойства). Откроется диалоговое окно **Properties** (Свойства).
8. Введите в текстовое поле **Prompt** (Подсказка) окна **Properties** (Свойства) подсказку "Align left\nLeft".
9. Повторите операции, описанные в п.п. 6—8, создав команду меню "&Center" и сопоставив подсказку "Center text\nCenter".
10. Повторите операции, описанные в п.п. 6—8, создав команду меню "&Right" и сопоставив подсказку "Align right\nRight".
11. Выделите прямоугольник с надписью "Type Here", щелкните на нем правой кнопкой мыши и выберите в появившемся контекстном меню команду **Insert Separator** (Вставить разделитель). В раскрытом меню появится разделитель.
12. Повторите операции, описанные в п.п. 6—8, создав команду меню "&Italic" и сопоставив подсказку "Italic font\nItalic".
13. Повторите операции, описанные в п.п. 6—8, создав команду меню "&Underline" и сопоставив подсказку "Underlined font\nUnderline".
14. Выберите команду меню **File | Save All** (Файл | Сохранить все) или нажмите кнопку **Save All** (Сохранить все) на панели инструментов **Standard** (Стандартная). Все изменения, внесенные в ресурс меню, будут сохранены в приложении.

15. Откройте окно **Class View** (Просмотр класса), а в нем раскройте папку **RichApp**.
16. Щелкните левой кнопкой мыши на папке `CRichAppView` и нажмите кнопку **Events** (События) в окне **Properties** (Свойства). Раскроется список событий, обрабатываемых данным классом.
17. В этом списке раскройте папку `ID_FORMAT_CENTER` и выделите в ней сообщение `COMMAND`. В соответствующем текстовом поле появится значок раскрывающегося списка.
18. Выделите в этом списке единственную строку. В текстовом поле появится имя функции обработки данного сообщения и откроется окно редактирования файла RichAppView.cpp.
19. Повторите п.п. 17 и 18 для идентификаторов ресурса `ID_FORMAT_ITALIC`, `ID_FORMAT_LEFT`, `ID_FORMAT_RIGHT` и `ID_FORMAT_UNDERLINE`.
20. В окне редактирования файла RichAppView.cpp измените добавленные в класс `CRichAppView` функции обработки сообщений меню в соответствии с текстом листинга 8.2.

Листинг 8.2. Функции обработки сообщений класса `CRichAppView`

```
// Функции обработки сообщений класса CRichAppView

// Центрирование текста
void CRichAppView::OnFormatCenter(void)
{
  OnParaAlign(PFA_CENTER);
}

// Вывод текста курсивом
void CRichAppView::OnFormatItalic(void)
{
  OnCharEffect(CFM_ITALIC, CFE_ITALIC);
}

// Выравнивание текста по левому краю
void CRichAppView::OnFormatLeft(void)
{
  OnParaAlign(PFA_LEFT);
}

// Выравнивание текста по правому краю
void CRichAppView::OnFormatRight(void)
```

```
{
  OnParaAlign(PFA_RIGHT);
}

// Подчеркивание текста
void CRichAppView::OnFormatUnderline(void)
{
  OnCharEffect(CFM_UNDERLINE, CFE_UNDERLINE);
}
```

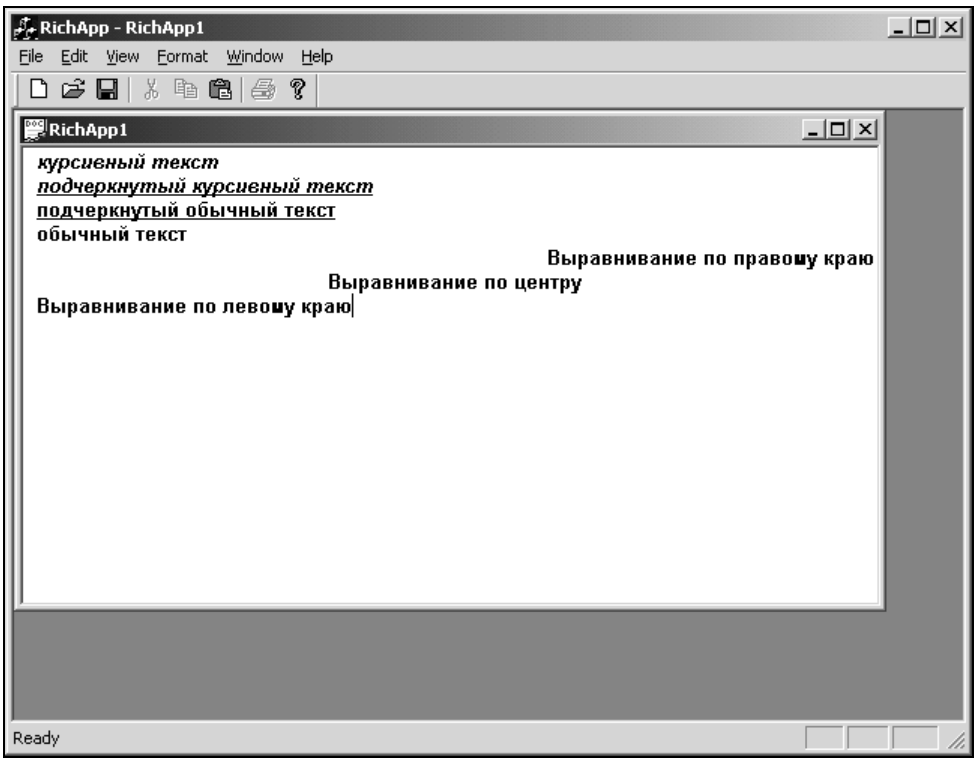


Рис. 8.10. Окно приложения RichApp

21. Нажмите клавишу <F5> и запустите приложение на исполнение.
22. Выберите команду меню **Format | Italic** (Формат | Italic) и введите текст в окно документа. Текст будет выведен курсивом.
23. Выберите команду **Format | Underline** (Формат | Underline) и продолжите ввод текста. Текст будет выведен подчеркнутым курсивом.

24. Снова выберите команду **Format | Italic** (Формат | Italic) и продолжите ввод текста. Текст будет выведен подчеркнутым обычным шрифтом.
25. Выберите команду **Format | Underline** (Формат | Underline) и продолжите ввод текста. Текст будет выведен обычным шрифтом.
26. Выберите команду **Format | Right** (Формат | По правому краю). Введенный текст будет выровнен по правому краю.
27. Выберите команду **Format | Center** (Формат | По центру). Введенный текст будет выровнен по центру.
28. Выберите команду **Format | Left** (Формат | По левому краю). Введенный текст будет выровнен по левому краю. Окно документа примет вид, изображенный на рис. 8.10.

Как видно из текстов функций обработки сообщений команд меню, процесс форматирования абзацев и шрифтов в данном приложении не представляет собой особых сложностей. Для форматирования абзацев используется функция `CRichEditView::OnParaAlign`, аргумент которой определяет требуемый режим форматирования выделенного абзаца. Для форматирования шрифтов используется функция `CRichEditView::OnCharEffect`, первый аргумент которой является маской, определяющей изменения, которые можно внести в данный шрифт, а второй ее аргумент определяет конкретные изменения, которые необходимо внести в данный шрифт.

Задание пользовательского шрифта

Одним из отличий редактора, создаваемого с использованием класса `CRichEditView`, по сравнению с редактором, создаваемым с использованием класса `CEdit`, является возможность работы с пользовательскими шрифтами. Чтобы продемонстрировать возможности работы класса `CRichEditView` по работе с пользовательскими шрифтами:

1. Выберите команду **File | Open Solution** (Файл | Открыть решение). В раскрывшемся диалоговом окне **Open Solution** (Открыть решение) раскройте папку **RichApp** и щелкните по значку **RichApp**. Откроется соответствующий проект.
2. Откройте окно **Resource View** (Просмотр ресурсов) и последовательно раскройте папки **RichtApp.rc** и **Menu** (Меню).
3. Дважды щелкните левой кнопкой мыши по значку `IDR_RichtAppTYPE`. Откроется окно редактирования ресурса меню.
4. Раскройте меню **Format** (Формат), выделите расположенный в его нижней части прямоугольник, содержащий надпись "Type Here", щелкните на нем правой кнопкой мыши и выберите в появившемся контекстном меню команду **Insert Separator** (Вставить разделитель). В заготовке раскрывающегося меню появится разделитель.

5. Щелкните левой кнопкой мыши по прямоугольнику с надписью "Type Here", расположенному под разделителем, введите в него текст "&My Font" и нажмите клавишу <Enter>.
6. Щелкните правой кнопкой мыши на только что созданной команде меню и выберите в раскрывшемся контекстном меню команду **Properties** (Свойства).
7. Введите в текстовое поле **Prompt** (Подсказка) подсказку "Set user font\nMy Font".
8. Выберите команду меню **File | Save All** (Файл | Сохранить все) или нажмите кнопку **Save All** (Сохранить все) на панели инструментов **Standard** (Стандартная). Все изменения, внесенные в ресурс меню, будут сохранены в приложении.
9. Откройте окно **Class View** (Просмотр класса), а в нем раскройте папку **RichApp**.
10. Щелкните левой кнопкой мыши на папке `CRichAppView` и нажмите кнопку **Events** (События) в окне **Properties** (Свойства). Раскроется список событий, обрабатываемых данным классом.
11. В этом списке раскройте папку `ID_FORMAT_MYFONT` и выделите в ней сообщение `COMMAND`. В соответствующем текстовом поле появится значок раскрывающегося списка.
12. Выделите в этом списке единственную строку. В текстовом поле появится имя функции обработки данного сообщения и откроется окно редактирования файла RichAppView.cpp.
13. В окне редактирования файла RichAppView.cpp измените функцию `OnFormatMyfont` в соответствии с текстом листинга 8.3.

Листинг 8.3. Функция `CRichAppView::OnFormatMyfont`

```
// Задание пользовательского формата

void CRichAppView::OnFormatMyfont(void)
{
  CHARFORMAT2&    cf = GetCharFormatSelection();

  cf.dwMask     | = CFM_COLOR;
  cf.yHeight    = 600;
  cf.crTextColor  = RGB(0, 0, 255);
  cf.bCharSet    = ANSI_CHARSET;
  cf.bPitchAndFamily  = VARIABLE_PITCH  |  FF_ROMAN;
  strcpy(cf.szFaceName, "MS Sans Serif");
```

```
  if(cf.dwEffects & CFE_AUTOCOLOR)
    cf.dwEffects = 0;

  SetCharFormat(cf);
}
```

14. Нажмите клавишу <F5> и запустите приложение на исполнение.
15. Введите какой-нибудь текст в окно документа, после чего выберите команду **Format | My Font** (Формат | Мой шрифт).
16. Продолжите ввод текста в окно документа. При этом текст будет иметь другое начертание, другой размер, будет иметь синий цвет и выводиться на черном фоне, как это показано на рис. 8.11.

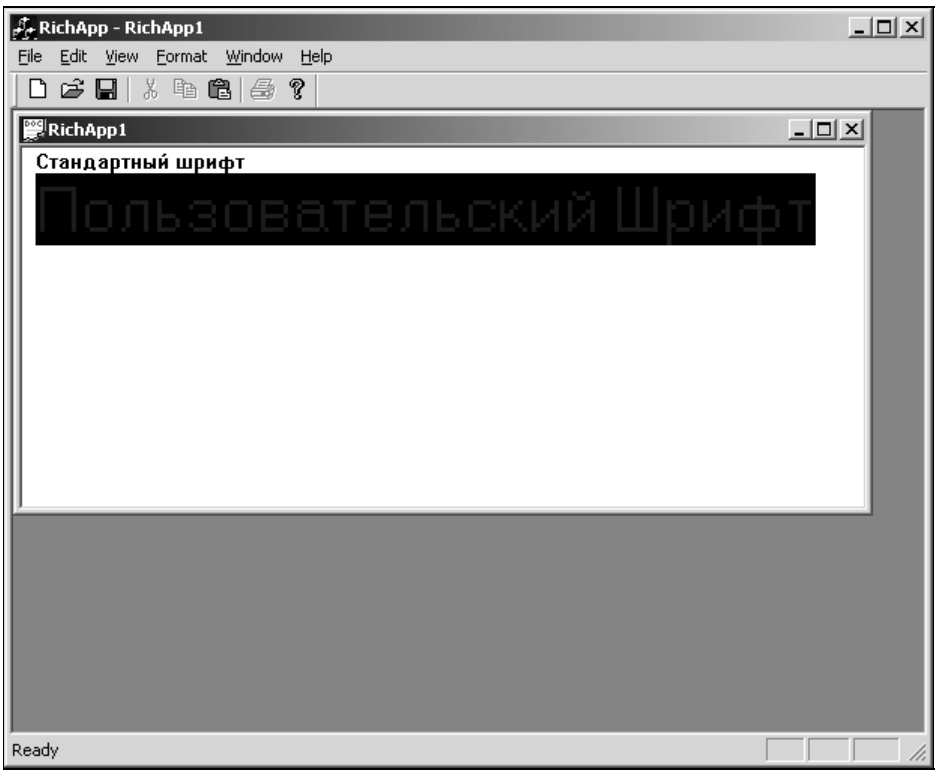


Рис. 8.11. Окно приложения RichApp с пользовательским шрифтом

Для задания параметров пользовательского шрифта использовался объект структуры `CHARFORMAT2`, содержащей различные параметры форматирования шрифтов. Все операции форматирования шрифта, произведенные нами в разделе,

посвященном форматированию текста, можно было бы осуществить с использованием объекта структуры `CHARFORMAT2`, тем более, что аргументы функции `CRichEditView::OnCharEffect` являются, по сути, членами данной структуры.

В Visual C++ 6.0 наряду со структурой `CHARFORMAT2` можно было использовать структуру `CHARFORMAT`, представляющую собой упрощенную (а точнее говоря — исходную) версию данной структуры. Однако в Visual C++ 7.0 такая возможность отсутствует.

Хотя работа со структурой `CHARFORMAT2` позволяет осуществлять большее количество операций форматирования шрифта, чем это позволяет функция `OnCharEffect`, работать с функцией намного удобнее.

В первом операторе функции `OnFont` получается ссылка на объект класса `CHARFORMAT2`, содержащий параметры текущего шрифта. Для этого используется функция `CRichEditView::GetCharFormatSelection`, позволяющая получить текущие атрибуты форматирования выделенного участка текста. После этого в полученный объект структуры `CHARFORMAT2` вносятся изменения. Прежде всего, в переменную `dwMask` данной структуры добавляется флаг `CFM_COLOR`, позволяющий изменять цвет текста (вообще-то этот флаг уже установлен по умолчанию, но осторожность не помешает). В переменной `yHeight` устанавливается высота шрифта в пиках. В переменную `crTextColor` записывается значение нового цвета шрифта. Значение переменной `bCharSet` определяет используемый набор символов. Переменная `bPitchAndFamily` определяет, будет ли шрифт иметь фиксированную или переменную ширину литер, а также семейство, к которому принадлежит данный шрифт. В текстовой переменной `szFaceName` хранится имя шрифта. После задания параметров шрифта производится проверка того, установлен ли в переменной `dwEffects` флаг `CFE_AUTOCOLOR`, препятствующий использованию переменной `crTextColor` для задания цвета текста и, если этот флаг установлен, то он сбрасывается вместе со всеми остальными флагами данной переменной. В завершение функции `OnFont` вызывается функция `CRichEditView::SetCharFormat`, устанавливающая атрибуты форматирования нового текста в объекте класса `CRichEditView`. В качестве аргумента данной функции используется модифицированный объект структуры `CHARFORMAT2`.

Глава 9

Панели инструментов и строка состояния

Панели инструментов представляют собой неотъемлемую часть любой полноценной программы Windows. Основное их назначение заключается в обеспечении быстрого доступа к наиболее часто используемым командам меню. В принципе, можно создать панель инструментов, кнопки которой не будут дублироваться в меню, но это существенно усложнит работу с приложением без использования мыши.

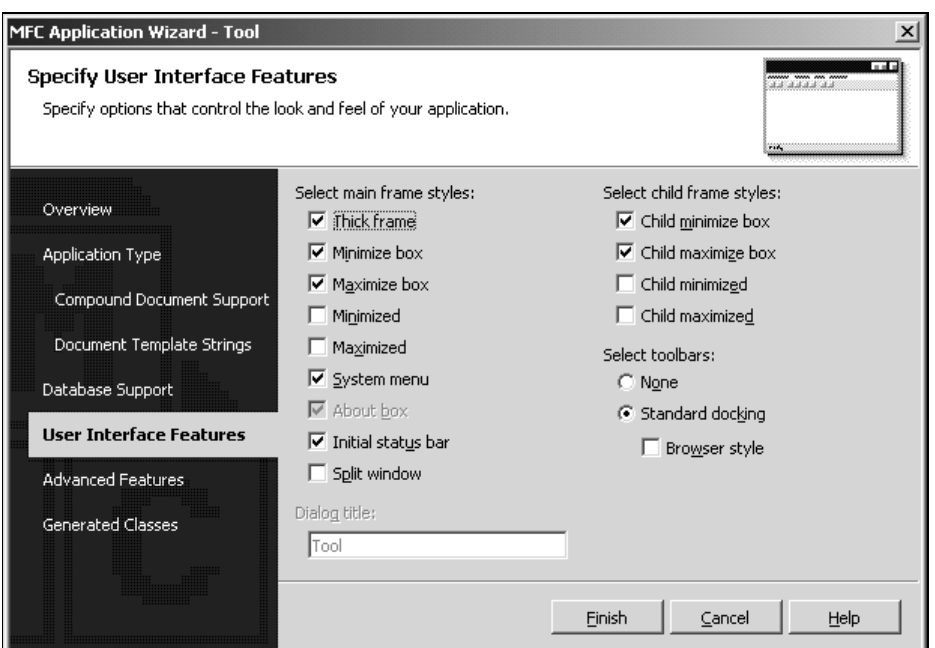


Рис. 9.1. Диалоговое окно **MFC Application Wizard - Tool**

Строка состояния располагается в нижней части главного окна программы и служит для отображения расширенной справки по командам меню и кнопкам панели инструментов. Кроме того, строка состояния содержит специальные поля для вывода дополнительной информации. По умолчанию в них отображается

состояние расположенных на клавиатуре индикаторов Caps Lock (поле **Cap**), Num Lock (поле **Num**) и Scroll Lock (поле **SCRL**). Если соответствующий светодиод на клавиатуре не светится, то в данном поле ничего не выводится. В противном случае в поле выводится его идентификатор, указанный в скобках.

Наличие в приложении панели инструментов и строки состояния определяется во вкладке **User Interface Features** (Свойства интерфейса пользователя) диалогового окна **MFC Application Wizard - Tool** (Мастер создания приложения MFC), изображенного на рис. 9.1. Для того чтобы создаваемое приложение имело стандартную панель инструментов, нужно оставить переключатель **Select toolbars** (Выбор панели инструментов) в положении **Standard docking** (Перемещаемая панель инструментов). Чтобы создаваемое приложение имело строку состояния, не нужно сбрасывать установленный по умолчанию флажок **Initial status bar** (Отображение строки состояния по умолчанию).

Работа с панелью инструментов

По умолчанию приложение создает стандартную панель инструментов, состоящую из четырех групп кнопок. Эта панель инструментов изображена на рис. 9.2.

Рис. 9.2. Стандартная панель инструментов приложения

В первую группу кнопок, осуществляющих операции по созданию и сохранению документа, входят кнопки (**New**, открытие нового документа), (**Open**, открытие существующего документа) и (**Save**, сохранение документа). Во вторую группу, осуществляющую операции по редактированию документа, входят кнопки (**Cut**, вырезать выделенный блок в буфер обмена), (**Copy**, копировать выделенный блок в буфер обмена) и (**Paste**, вставить блок из буфера обмена). В третью группу входит кнопка (**Print**, печать текущего документа), а в четвертую — (**About**, вызов диалогового окна, содержащего сведения о приложении).

Система программирования Visual C++ позволяет добавлять и уничтожать кнопки в стандартной панели инструментов, но в большинстве случаев для новых кнопок целесообразнее использовать новые панели инструментов, а из стандартной панели инструментов только удалять ненужные кнопки.

Удаление кнопок из панели инструментов

Предположим, что в создаваемом приложении нет необходимости в использовании буфера обмена. В этом случае необходимо или постоянно делать недоступными соответствующие кнопки панели инструментов, или же просто удалить их из данной панели инструментов. Второй путь, конечно же, предпочтительнее, поскольку, как говорил К. С. Станиславский, если в первом акте на стене висит ружье, то в третьем акте оно должно выстрелить. Постоянное присутствие в панели инструментов "не стреляющих" кнопок вызывает у пользователя законный вопрос: "В каком же режиме они стреляют?"

Текст демонстрационного приложения, позволяющего показать принципы работы с панелью инструментов, расположен в папке Tool дискеты, прилагаемой к данной книге. Чтобы самостоятельно создать данное приложение:

1. Создайте приложение с именем Tool по методике, описанной в *главе 1*. Оставьте все установки по умолчанию без изменения.
2. Откройте окно **Resource View** (Просмотр ресурсов), раскройте папку **Tool.rc**, а в ней — папку **Toolbar**.

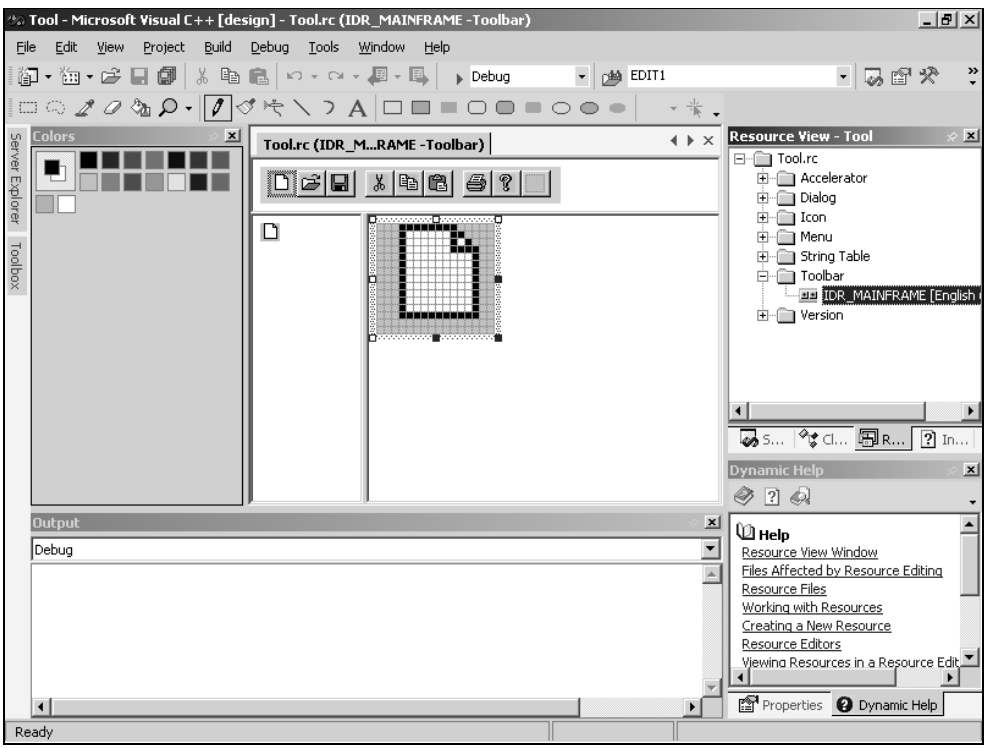


Рис. 9.3. Редактор ресурса панели инструментов

3. Дважды щелкните по идентификатору ресурса панели инструментов `IDR_MAINFRAME`. В окне редактора ресурсов появится панель инструментов, в которой будет выделена первая кнопка, а ее увеличенное изображение будет находиться в специальной области редактирования. Экран при этом примет вид, изображенный на рис. 9.3.
4. Поместите указатель мыши на изображение кнопки **Cut** (Вырезать) на панели инструментов, изображенной в редакторе ресурса. Нажмите левую кнопку мыши и, не отпуская ее, перетащите данную кнопку за пределы панели инструментов. Отпустите кнопку — перетаскиваемое изображение исчезнет, а панель инструментов не будет содержать кнопки **Cut** (Вырезать).
5. Повторите эту же операцию для кнопок **Copy** (Копировать) и **Paste** (Вставка). Панель инструментов не будет содержать кнопок редактирования документа.
6. Нажмите клавишу <F5> и запустите приложение на исполнение. Главное окно приложения примет вид, изображенный на рис. 9.4.

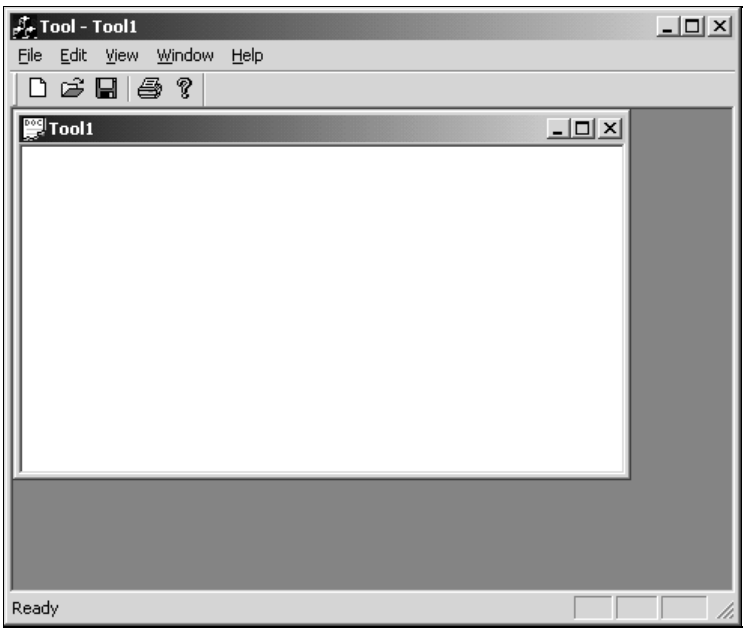


Рис. 9.4. Главное окно приложения Tool

Если с удаляемыми кнопками были связаны какие-либо сообщения, обрабатывавшиеся приложением, то перед удалением кнопок из панели инструментов необходимо предварительно уничтожить функции обработки связанных с ними сообщений.

Добавление кнопок в панель инструментов

Поскольку процедура добавления кнопки в стандартную панель инструментов ничем не отличается от процедуры добавления кнопки в любую другую панель инструментов, а создание пользовательской панели инструментов является далеко не тривиальной задачей, то в созданное в предыдущем разделе приложение будет добавлена еще одна панель инструментов, содержащая всего одну кнопку.

Чтобы внести соответствующие изменения в приложение Tool:

1. Выберите команду **File | Open Solution** (Файл | Открыть решение) и в открывшемся диалоговом окне **Open Solution** (Открыть решение) раскройте папку **Tool**, выделите значок **Tool** и нажмите кнопку **Open** (Открыть).
2. Откройте окно **Resource View** (Просмотр ресурсов), а в нем — папку **Tool.rc**.
3. Щелкните правой кнопкой мыши на папке **Toolbar** и в появившемся контекстном меню выберите команду **Insert Toolbar** (Вставить панель инструментов). В окне редактирования ресурса появится новая панель инструментов и экран примет вид, изображенный на рис. 9.5.

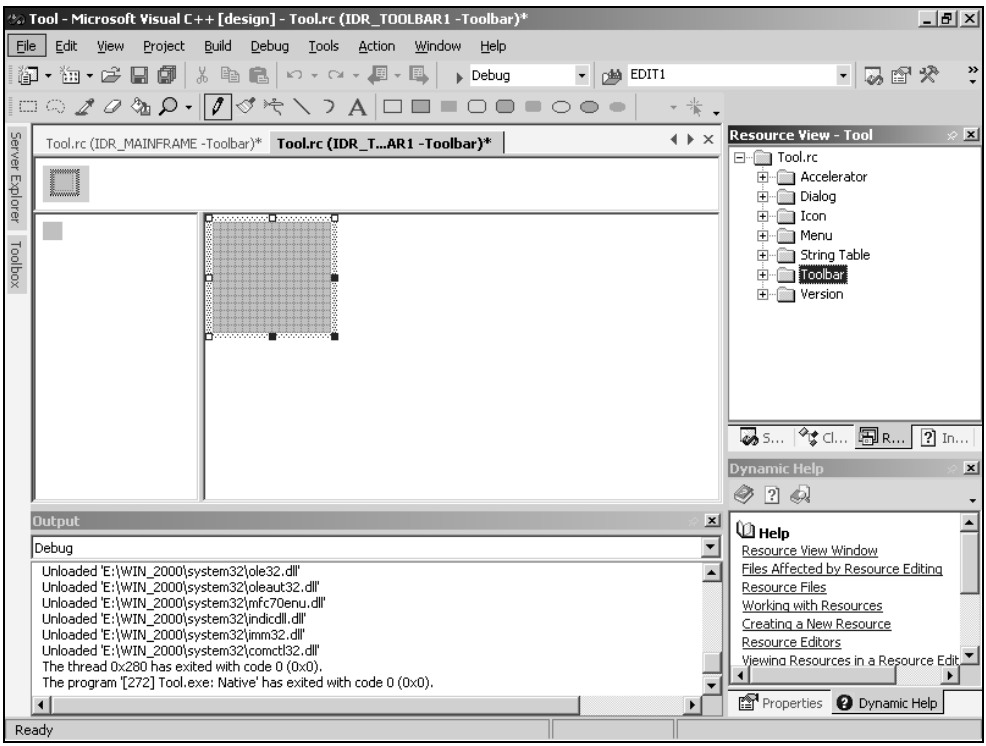


Рис. 9.5. Окно редактирования пользовательской панели инструментов

4. Раскройте папку **Toolbar** (Панель инструментов), щелкните правой кнопкой мыши по идентификатору `IDR_TOOLBAR1` и выберите в раскрывшемся контекстном меню команду **Properties** (Свойства). Откроется окно **Properties** (Свойства).
5. В текстовом поле раскрывающегося списка **ID** (Идентификатор ресурса) окна **Properties** (Свойства) замените идентификатор ресурса `IDR_TOOLBAR1` идентификатором ресурса `IDR_TOOLBAR`.
6. Нажмите кнопку **Rectangle** (Прямоугольник) панели инструментов **Image Editor** (Редактор рисунков) и нарисуйте в заготовке кнопки диалогового окна черный прямоугольник.
7. Щелкните правой кнопкой мыши в окне редактирования ресурса и выберите в появившемся контекстном меню команду **Show Color Window** (Вывести окно палитры). В окне редактирования ресурса появится окно **Colors** (Палитра), показанное на рис. 9.3 и 9.6.
8. Щелкните левой кнопкой мыши на красном квадрате в окне **Colors** (Палитра).
9. Нажмите кнопку **Fill** (Заполнение) на панели инструментов **Image Editor** (Редактор рисунков), после чего щелкните левой кнопкой мыши внутри квадрата, нарисованного нами в заготовке кнопки панели инструментов. Квадрат будет заполнен красным цветом.
10. Введите в текстовое поле раскрывающегося списка **ID** (Идентификатор ресурса) окна **Properties** (Свойства) идентификатор ресурса `ID_SQUARE` и щелкните левой кнопкой мыши на заготовке второй кнопки панели инструментов (иначе текстовое поле **Prompt** (Подсказка) останется недоступным).
11. Щелкните на заготовке созданной кнопки панели инструментов, введите в текстовое поле **Prompt** (Подсказка) окна **Properties** (Свойства) подсказку "Draw red square\nSquare". Окно редактирования ресурса примет вид, изображенный на рис. 9.6.
12. Раскройте папку **Menu** (Меню) в окне **Resource View** (Просмотр ресурсов) и дважды щелкните по значку `IDR_MAINFRAME`. Появится окно редактирования ресурса основного меню приложения.
13. В окне редактирования ресурса раскройте меню **View** (Вид).
14. Щелкните левой кнопкой мыши на прямоугольнике с надписью "Type Here", расположенном в нижней части раскрывшегося меню. Прямоугольник превратится в текстовое поле.
15. Введите в это текстовое поле команду "&My Toolbar" и выберите любую другую команду меню. На месте текстового поля появится введенная команда меню.
16. Поместите указатель мыши на команду меню **My Toolbar** (Моя панель инструментов), нажмите левую кнопку мыши и, не отпуская ее, перетащите ее под команду **Toolbar** (Панель инструментов).

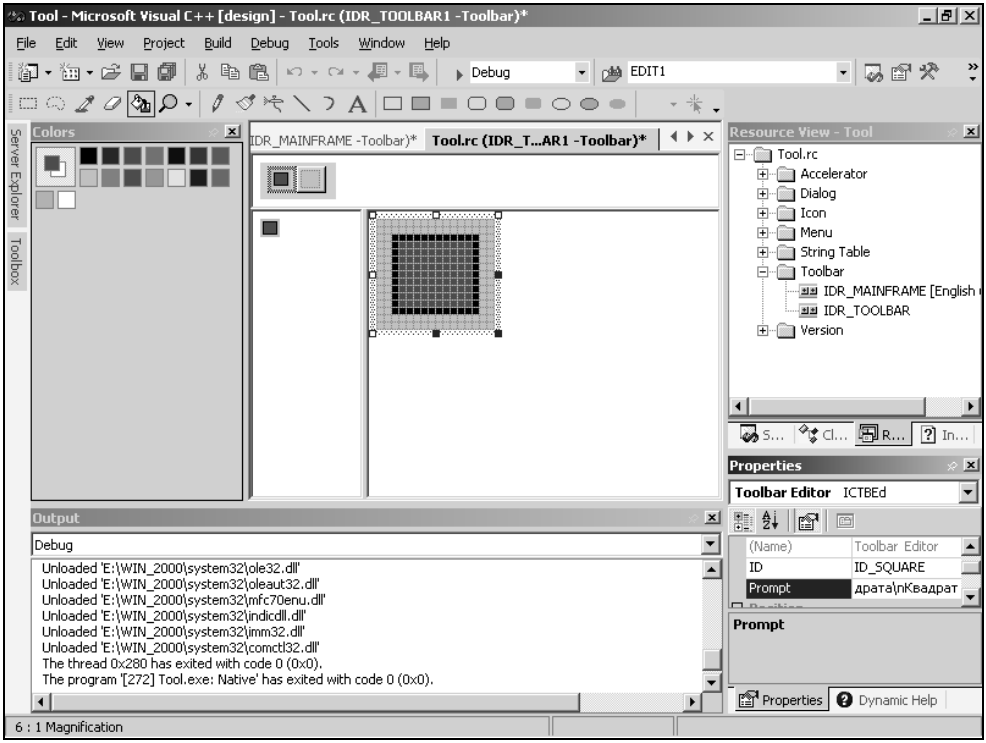


Рис. 9.6. Окно редактирования пользовательской панели инструментов с окном **Colors**

17. В текстовое поле раскрывающегося списка **ID** (Идентификатор ресурса) окна **Properties** (Свойства) введите идентификатор ресурса `ID_VIEW_MYTOOLBAR` и выберите команду меню **File | Save All** (Файл | Сохранить все) или нажмите кнопку **Save All** (Сохранить все) на панели инструментов **Standard** (Стандартная). Все изменения, внесенные в ресурс меню, будут сохранены в приложении.
18. В текстовое поле **Prompt** (Подсказка) окна **Properties** (Свойства) введите подсказку "Show or hide the toolbar\nToggle ToolBar".
19. Дважды щелкните по значку `IDR_ToolTYPE` в окне **Resource View** (Просмотр ресурсов). Появится окно редактирования ресурса меню работы с документом приложения.
20. Повторите операции, описанные в п.п. 13—19, для данного ресурса меню.
21. Щелкните левой кнопкой мыши на прямоугольнике с надписью "Type Here", расположенном в правой части строки меню. Прямоугольник превратится в текстовое поле.

22. Введите в это текстовое поле заголовок "&Draw" и выберите любую другую команду меню. На месте текстового поля появится введенный заголовок раскрывающегося меню.
23. Поместите этот заголовок между пунктами меню **View** (Вид) и **Window** (Окно).
24. Щелкните левой кнопкой мыши на прямоугольнике с надписью "Type Here", расположенном в новом раскрывающемся меню. Прямоугольник превратится в текстовое поле.
25. Введите в это текстовое поле команду "&Square" и выберите любую другую команду меню. На месте текстового поля появится введенная команда меню.
26. В текстовое поле раскрывающегося списка **ID** (Идентификатор ресурса) окна **Properties** (Свойства) введите идентификатор ресурса `ID_SQUARE` (или выделите его в этом раскрывающемся списке) и выберите команду меню **File | Save All** (Файл | Сохранить все) или нажмите кнопку **Save All** (Сохранить все) на панели инструментов **Standard** (Стандартная). Все изменения, внесенные в ресурс меню, будут сохранены в приложении.
27. В текстовое поле **Prompt** (Подсказка) окна **Properties** (Свойства) введите подсказку "Draw red square\nSquare".
28. Откройте окно **Class View** (Просмотр класса) и выделите в нем папку `CToolView`.
29. В окне **Properties** (Свойства) нажмите кнопку **Events** (События). Раскроется список событий, обрабатываемых данным классом.
30. В этом списке раскройте папку `ID_SQUARE` и выделите в ней сообщение `COMMAND`. В соответствующем текстовом поле появится значок раскрывающегося списка.
31. Раскройте этот список и добавьте функцию обработки данного сообщения.
32. Повторите п.п. 30 и 31 для добавления в приложение функции обработки сообщения `UPDATE_COMMAND_UI` для того же идентификатора ресурса.
33. В окне **Class View** (Просмотр класса) выделите папку `CMainFrame`.
34. Повторите п.п. 29 и 32 для идентификатора ресурса `ID_VIEW_MYTOOLBAR`.
35. В окне **Properties** (Свойства) нажмите кнопку **Messages** (Собщения). Раскроется список сообщений, обрабатываемых данным классом.
36. В этом списке выделите сообщение `WM_CLOSE` и добавьте в приложение функцию обработки данного сообщения.
37. В окне **Class View** (Просмотр класса) щелкните правой кнопкой мыши на папке `CMainFrame` и выберите в появившемся контекстном меню команду **Add | Add variable** (Добавить | Добавить переменную). Появится диалоговое окно **Add Member Variable Wizard** (Мастер добавления переменной в класс).
38. В текстовое поле раскрывающегося списка **Variable type** (Тип переменной) введите тип переменной `CToolBar`, в текстовое поле **Variable name** (Имя перемен-

ной) введите идентификатор переменной `m_MyToolBar` и нажмите кнопку **Finish** (Готово). В класс будет добавлена новая переменная, в окне редактирования файлов появятся панели файла заголовка и файла реализации данного класса и откроется панель редактирования файла реализации класса `CMainFrame`.

39. Откройте окно редактирования файла MainFrm.h.
40. В карте сообщений класса `CMainFrame` после строки

    ```
    void OnUpdateViewMytoolbar(CCmdUI *pCmdUI);
    ```

    вставьте строку

    ```
    BOOL OnMyToolbarCheck(UINT nID);
    ```

41. Откройте окно редактирования файла MainFrm.cpp и в карте сообщений перед строкой

    ```
    ON_UPDATE_COMMAND_UI(ID_VIEW_MYTOOLBAR, OnUpdateViewMytoolbar)
    ```

    вставьте строку

    ```
    ON_COMMAND_EX(ID_PROCESS_TOOLBAR, OnMyToolbarCheck)
    ```

42. Измените функцию `OnCreate` в соответствии с текстом листинга 9.1.

Листинг 9.1. Функция `CMainFrame::OnCreate`

```
// Процедура создания главного окна приложения

int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
  // Вызов метода базового класса
  if (CMDIFrameWnd::OnCreate(lpCreateStruct) == -1)
    return -1;

  // Создание стандартной панели инструментов
  if (!m_wndToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD  |
       WS_VISIBLE  |  CBRS_TOP  |  CBRS_GRIPPER  |  CBRS_TOOLTIPS  |
       CBRS_FLYBY  |  CBRS_SIZE_DYNAMIC)  | |
       !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
  {
    TRACE0("Failed to create toolbar\n");
    return -1;  // Ошибка при создании
  }

  // Создание пользовательской панели инструментов
  if (!m_MyToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD  | WS_VISIBLE  |
       CBRS_TOP  |  CBRS_GRIPPER  |  CBRS_TOOLTIPS  |  CBRS_FLYBY  |
```

```
      CBRS_SIZE_DYNAMIC)  |  | !m_MyToolBar.LoadToolBar(IDR_TOOLBAR))
  {
    TRACE0("Failed to create toolbar\n");
    return -1;  // Ошибка при создании
  }

  // Создание строки состояния
  if (!m_wndStatusBar.Create(this)  |  |
    !m_wndStatusBar.SetIndicators(indicators,
      sizeof(indicators)/sizeof(UINT)))
  {
    TRACE0("Failed to create status bar\n");
    return -1;  // Ошибка при создании
  }

  // Установка нового идентификатора окна
  SetWindowLong(m_MyToolBar.m_hWnd, GWL_ID, ID_VIEW_MYTOOLBAR);

  // Обеспечение возможности фиксации панелей инструментов
  m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
  m_MyToolBar.EnableDocking(CBRS_ALIGN_ANY);
  EnableDocking(CBRS_ALIGN_ANY);
  DockControlBar(&m_wndToolBar);
  DockControlBar(&m_MyToolBar);

  // Загрузка предыдущего состояния панелей инструментов
  // из системного реестра
  LoadBarState("Local AppWizard-Generated Applications");
  return 0;
}
```

43. Измените функцию `OnClose` в соответствии с текстом листинга 9.2.

Листинг 9.2. Функция `CMainFrame::OnClose`

```
// Функции обработки сообщений класса CMainFrame

// Закрытие главного окна приложения
void CMainFrame::OnClose(void)
{
```

```
  // Сохранение состояния панелей инструментов в системном реестре
  SaveBarState("Local AppWizard-Generated Applications");
  CMDIFrameWnd::OnClose();
}
```

44. Измените функцию `OnUpdateMyToolbar` в соответствии с текстом листинга 9.3.

Листинг 9.3. Функция `CMainFrame::OnUpdateViewMytoolbar`

```
// Изменение состояния команды меню

void CMainFrame::OnUpdateViewMytoolbar(CCmdUI *pCmdUI)
{
  if (&m_MyToolBar != NULL)  // Панель инструментов существует
  {
    // Установка состояния команды меню в зависимости
    // от видимости панели инструментов
    pCmdUI->SetCheck((m_MyToolBar.GetStyle() & WS_VISIBLE) != 0);
    return;  // Завершение обработки команда
  }

  // Продолжение обработки команды
  pCmdUI->ContinueRouting();
}
```

45. В конец данного файла добавьте текст листинга 9.4.

Листинг 9.4. Функция `CMainFrame::OnMyToolbarCheck`

```
// Обработка сообщения об отмене команды меню

BOOL CMainFrame::OnMyToolbarCheck(UINT nID)
{
  if (&m_MyToolBar != NULL)  // Панель инструментов существует
  {
    ShowControlBar(&m_MyToolBar,
       (m_MyToolBar.GetStyle() & WS_VISIBLE) == 0, FALSE);
    return TRUE;
  }
  return FALSE;
}
```

46. В окне **Class View** (Просмотр класса) щелкните правой кнопкой мыши по папке `CToolView` и выберите в появившемся контекстном меню команду **Add | Add Variable** (Добавить | Добавить переменную). Появится диалоговое окно **Add Member Variable Wizard** (Мастер добавления переменной в класс).
47. Введите в текстовое поле раскрывающегося списка **Variable type** (Тип переменной) тип переменной `bool`, в текстовое поле **Variable name** (Имя переменной) введите идентификатор переменной `bDrawn` и нажмите кнопку **Finish** (Готово). В класс будет добавлена новая переменная и откроется окно редактирования файла ToolAppView.cpp.
48. Измените функцию `OnSquare` в соответствии с текстом листинга 9.5.

Листинг 9.5. Функция `CToolView::OnSquare`

```
// Вывод красного квадрата

void CToolView::OnSquare(void)
{
  CClientDC  dc(this);
  CBrush    redBrush(RGB(255,0,0));
  CPen    pen(PS_SOLID, 0, RGB(0,0,255));
  CPen*    oldPen = dc.SelectObject(&pen);
  CBrush*  oldBrush = dc.SelectObject(&redBrush);

  dc.Rectangle(20, 20, 120, 120);
  dc.SelectObject(oldBrush);
  dc.SelectObject(oldPen);

  bDrawn  = true;
}
```

49. Измените функцию `OnUpdateSquare` в соответствии с текстом листинга 9.6.

Листинг 9.6. Функция `CToolView::OnUpdateSquare`

```
// Определение состояния команды меню и кнопки панели инструментов

void CToolView::OnUpdateSquare(CCmdUI *pCmdUI)
{
  pCmdUI-> Enable(!bDrawn);
}
```

50. Измените функцию `OnLButtonDown` в соответствии с текстом листинга 9.7.

Листинг 9.7. Функция `CToolView::OnLButtonDown`

```
// Функции обработки сообщений класса CToolView

// Очистка экрана
void CToolView::OnLButtonDown(UINT nFlags, CPoint point)
{
  bDrawn = false;
  Invalidate();
  CView::OnLButtonDown(nFlags, point);
}
```

51. Нажмите клавишу <F5> и запустите приложение на исполнение. Появится главное окно приложения, изображенное на рис. 9.7.

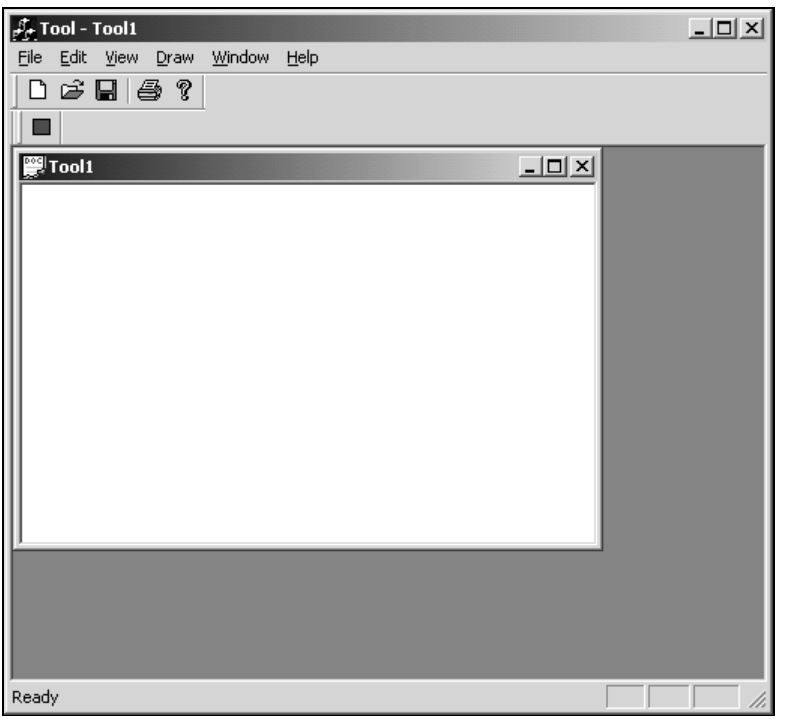


Рис. 9.7. Главное окно приложения Tool с пользовательской панелью инструментов.

52. Нажмите кнопку **Square** (Квадрат) на пользовательской панели инструментов. В окне документа появится красный квадрат, а кнопка **Square** (Квадрат) на панели инструментов станет недоступной.
53. Щелкните левой кнопкой мыши в окне документа. Красный квадрат исчезнет, а кнопка **Square** (Квадрат) на пользовательской панели инструментов станет доступной.
54. Выберите команду меню **Draw | Square** (Рисование | Квадрат). В окне документа появится красный квадрат, а кнопка **Square** (Квадрат) на панели инструментов станет недоступной.
55. Раскройте меню **Draw** (Рисование) и убедитесь, что команда **Square** (Квадрат) в нем недоступна.
56. Снова щелкните левой кнопкой мыши в окне документа и закройте окно документа. Кнопка **Square** (Квадрат) на панели инструментов станет недоступной, а меню **Draw** (Рисование) исчезнет, как это показано на рис. 9.8.

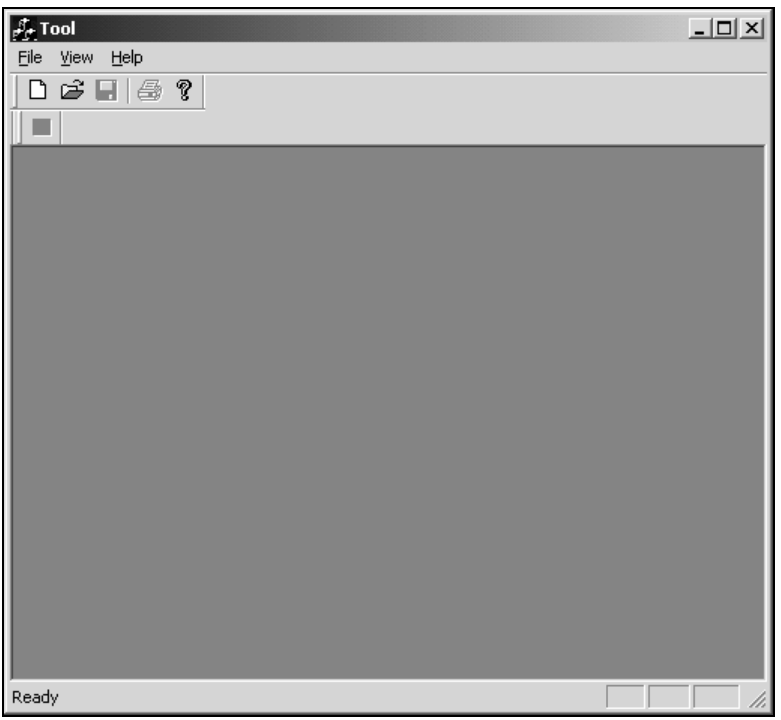


Рис. 9.8. Главное окно приложения Tool с закрытым окном документа

57. Откройте меню **View** (Вид) и снимите флажок с команды **My Toolbar** (Моя панель инструментов). Новая панель инструментов исчезнет из главного окна приложения, как это показано на рис. 9.9.

Рис. 9.9. Главное окно приложения Tool без пользовательской панели инструментов

58. Снова откройте меню **View** (Вид) и установите флажок у команды **My Toolbar** (Моя панель инструментов). Новая панель инструментов появится в главном окне приложения.
59. Перетащите новую панель инструментов в один ряд со стандартной панелью, как это показано на рис. 9.10.
60. Закройте приложение и снова запустите его на исполнение. Новая панель инструментов сохранит свое последнее положение.

Как следует из описанной выше процедуры, процесс создания пользовательской панели инструментов не полностью автоматизирован и пользователю приходится вносить исправления туда, где должна хозяйничать среда программирования. Это говорит о том, что разработчики Visual C++ не предполагали, что пользователям потребуется использовать несколько панелей инструментов.

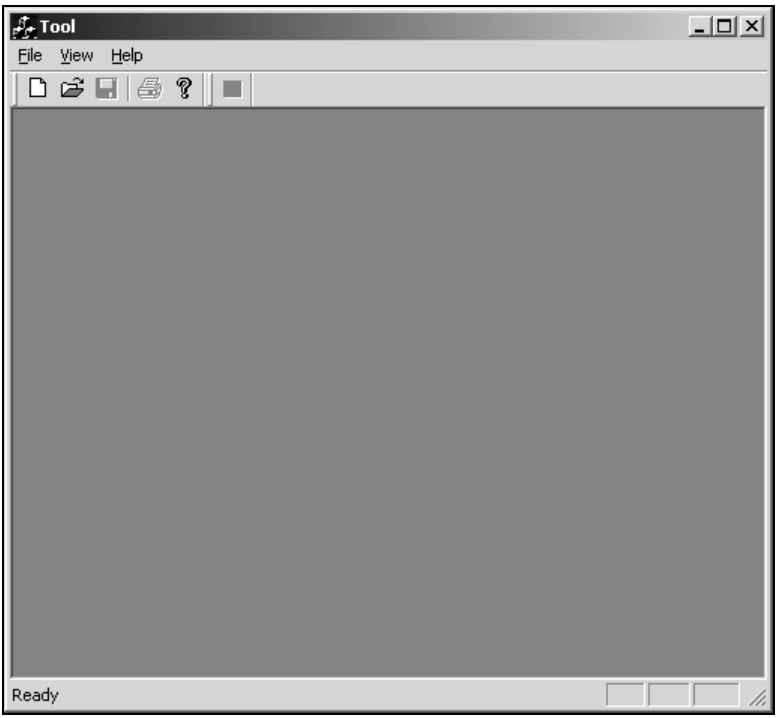


Рис. 9.10. Перемещенная пользовательская панель инструментов

Примечание

В отличие от Visual C++ 6.0 в Visual C++ 7.0 отсутствует возможность связи панели инструментов с классом, в котором будут обрабатываться ее сообщения. Эта возможность имеется только для команд меню (для которых связывание производится автоматически). Поэтому в Visual C++ 7.0, в отличие от Visual C++ 6.0, отсутствует возможность создания на панели инструментов кнопок, не имеющих соответствия среди команд меню.

Объект класса `CToolBar` включается пользователем в описание класса `CMainFrame` вручную, что не представляет особой сложности, поскольку в данном классе уже определен один объект класса `CToolBar` — стандартная панель инструментов. Если пользователь хочет иметь возможность убирать созданную им панель инструментов с экрана, он должен самостоятельно вставить в карту сообщений соответствующие макросы и объявления функций. Формат этих макросов и функций аналогичен форматам макросов и функций, использованных в данном приложении.

Панели инструментов создаются в функции `CMainFrame::OnCreate`, т. е. при создании главного окна приложения. Создание пользовательской панели инструментов в приложении во многом аналогично созданию уже существующей

стандартной панели инструментов, поэтому большинство операторов, используемых для ее создания, повторяют уже содержащиеся в данной функции операторы, используемые для создания стандартной панели инструментов.

Первым оператором данной функции является вызов метода базового класса. Следующим оператором является вызов функции `CToolBar::CreateEx` для объекта класса стандартной панели инструментов. Данная функция создает панель инструментов Windows (дочернее окно) и связывает ее с объектом класса `CToolBar`. После этого в том же условном операторе вызывается функция `CToolBar::LoadToolBar`, загружающая панель инструментов Windows, определяемую ее аргументом. В случае возникновения ошибки при выполнении одной из этих функций выдается сообщение об ошибке с использованием макроса `TRACE0`, не использующего строковых ресурсов, вопреки установленному разработчиками Windows правилу, согласно которому все выдаваемые системой сообщения должны храниться в строковых ресурсах для обеспечения возможности полной локализации программного обеспечения.

После выдачи сообщения об ошибке производится выход из функции с присвоением выходной величине значения −1, свидетельствующего об ошибке при создании главного окна приложения.

Следующий условный оператор производит те же действия для создания объекта класса пользовательской панели инструментов. Меняется только идентификатор объекта класса. В следующем условном операторе создается объект класса строки состояния. Поскольку мы еще не рассматривали данный класс, то пока оставим этот оператор без рассмотрения. Его описание будет приведено при рассмотрении класса строки состояния.

После этого следует вызов глобальной функции `SetWindowLong`, устанавливающей идентификатор окна пользовательской панели инструментов. Дело в том, что по непонятным причинам функции, используемые для создания панели инструментов, присваивают всем создаваемым ими панелям инструментов один и тот же идентификатор 0x0000e800. Это не позволяет приложению правильно работать с системным реестром, поскольку при записи и чтении информации из него две панели инструментов становятся неразличимыми и их порядок появления на экране определяется не тем, как их расположил пользователь, а тем, в каком порядке они зарегистрированы в приложении. Вызов данной функции устанавливает для пользовательской панели инструментов другой идентификатор, соответствующий идентификатору ее ресурса, что снимает проблемы, связанные с идентификацией панелей инструментов.

Функция `CControlBar::EnableDocking` определяет возможность фиксации панели управления на одной из сторон окна и определяет стороны, на которых панель управления может фиксироваться. В данном случае любая из панелей инструментов может фиксироваться на любой из сторон рабочей области главного окна приложения. Далее вызывается функция `CFrameWnd::EnableDocking`, разрешающая фиксацию панелей управления на любой стороне своей рабочей области. Аргумент данной функции должен совпадать с аргументом функции `CControlBar::EnableDocking`. После этого для каждой панели инструментов

вызываются функции DockControlBar, осуществляющие фиксацию указанной в качестве аргумента панели инструментов. Последним оператором функции OnCreate является вызов функции CFrameWnd::LoadBarState, позволяющей восстановить установки для каждой панели управления, принадлежащей главному окну приложения. В качестве аргумента данной функции используется ключ, заданный функцией CWinApp::SetRegistryKey, вызываемой в функции CToolAppApp::InitInstance.

Для того чтобы при вызове функции CFrameWnd::LoadBarState панели инструментов принимали свое последнее состояние, в класс CMainFrame введена функция OnClose, являющаяся обработчиком сообщения WM_CLOSE, посылаемого окну или приложению для того, чтобы известить его о необходимости завершить свою работу. В данную функцию добавлен вызов функции CFrameWnd::SaveBarState, позволяющей сохранить информацию об установках каждой панели управления, принадлежащей данному окну.

Функция OnUpdateMyToolbar является функцией обработки сообщения UPDATE_COMMAND_UI, посылаемого приложением при открытии соответствующего раскрывающегося меню для определения состояния его команд. Данная функция, прежде всего, проверяет наличие в приложении объекта класса, управление которым осуществляется данной командой меню. Если такой объект класса присутствует, вызывается функция CCmdUI::SetCheck, устанавливающая состояние команды меню или кнопки панели инструментов. В качестве аргумента данной функции используется логическое выражение, проверяющее текущее состояние панели инструментов. Для этого вызывается функция CWnd::GetStyle, возвращающая стиль окна, и в полученной величине проверяется значение бита WS_VISIBLE. Таким образом, если панель инструментов отображается на экране, то у данной команды меню устанавливается флажок. Если в данном приложении отсутствует объект класса, управление которым осуществляется данной командой меню, то функция OnUpdateMyToolbar вызывает функцию CCmdUI::ContinueRouting, сообщающую процедуре обработки сообщений, что обработка данного сообщения не завершена в данной функции и требуется найти обработчик сообщения, который завершит процедуру его обработки.

Прототип функции OnMyToolbarCheck не создается средой программирования, поэтому пользователю необходимо написать ее целиком. Данная функция является функцией обработки сообщения об установке и снятии флажка у команды меню, идентификатор ресурса которой передается ей в качестве аргумента, и нигде в данной функции не используется. В данной функции так же, как и в функции OnUpdateMyToolbar, производится проверка наличия в приложении объекта класса, управление которым осуществляется данной командой меню. Если такой объект класса присутствует, то вызывается функция CFrameWnd::ShowControlBar, позволяющая выводить панель управления на экран и убирать ее с экрана. Первым ее аргументом является указатель на объект класса панели управления, которую необходимо вывести на экран или убрать с экрана. Второй аргумент определяет действие, которое необходимо произвести с данной панелью управления, а третий аргумент определяет задержку при выводе панели управления на экран. В случае благоприятного исхода функция

`OnMyToolbarCheck` возвращает значение `TRUE`, в противном случае возвращается значение `FALSE`.

Функция `CToolAppView::OnSquare` является функцией обработки сообщения, поступающего от кнопки панели инструментов и от команды меню. Кнопка панели инструментов и команда меню дублируют друг друга, поскольку они имеют один и тот же идентификатор ресурса, поэтому все изменения с данными элементами управления происходят синхронно. Сама функция `OnSquare` является демонстрационной и не представляет собой ничего особенного. В ней создаются объекты классов, необходимые для рисования прямоугольника, выбираются в контекст устройства, рисуется прямоугольник, а после этого контекст устройства возвращается в исходное состояние. После этого переменной `bDrawn`, которой при инициализации объекта данного класса было присвоено значение `FALSE`, присваивается значение `TRUE`.

Функция `OnUpdateSquare` определяет состояние соответствующей кнопки панели инструментов и команды меню. Вызываемая в ней функция `CCmdUI::Enable` в зависимости от значения переменной `bDrawn` делает доступными или недоступными данные элементы управления. Логика работы данной функции состоит в том, что при фиксированных координатах и размерах в окне можно нарисовать только один квадрат. Поэтому после его рисования кнопка панели инструментов и команда меню, осуществляющие данную операцию, становятся недоступными.

Для возобновления доступа к данным элементам управления в функции `OnLButtonDown`, являющейся функцией обработки сообщения о нажатии на левую кнопку мыши, переменной `bDrawn` присваивается значение `FALSE` и вызывается функция `Invalidate`, производящая очистку рабочей области окна документа, поскольку в функцию `OnDraw` не были внесены никакие изменения.

Работа со строкой состояния

Строка состояния в приложениях, созданных мастером MFC Application Wizard, является настолько стандартным элементом, что для нее даже не создается никакого ресурса, позволяющего использовать для работы с ней возможности среды программирования.

Строка состояния разрабатывалась специалистами фирмы Microsoft для простейшего текстового редактора, поскольку только в нем имеет смысл отображать состояние функциональных клавиш, определяющих режим вывода текста на экран. В подавляющем большинстве других применений информация, выводимая в панелях строки состояния, не несет никакой полезной нагрузки. Это обстоятельство при отсутствии в основной среде разработки приложений, поставляемой фирмой Microsoft для своей операционной системы, средств для удобного редактирования строки состояния, еще раз подтверждает, что Microsoft считает текстовые редакторы единственно возможной формой приложения, работающего в среде Windows.

Однако широкое проникновение этой "операционной системы для домохозяек", как изначально позиционировала ее на рынке фирма Microsoft, в несвойственные ей области применения привело к необходимости редактирования строки состояния практически в любом серьезном приложении, создаваемом с использованием Visual C++.

В большинстве случаев это редактирование сводится к уничтожению стандартных панелей строки состояния и созданию на их месте новых панелей. Поскольку функции обработки сообщений от стандартных панелей строки состояния находятся в библиотеке MFC и не могут быть отредактированы пользователем, то вся операция уничтожения стандартных панелей сводится к удалению их идентификаторов в массиве `indicators`.

Существует два способа обновления информации в строке состояния. Для начала рассмотрим способ, рекомендуемый Microsoft.

Чтобы изменить информацию, выводимую в строке состояния приложения Tool:

1. Выберите команду **File | Open Solution** (Файл | Открыть решение) и в открывшемся диалоговом окне **Open Solution** (Открыть решение) раскройте папку **Tool**, выделите значок **Tool** и нажмите кнопку **Open** (Открыть).

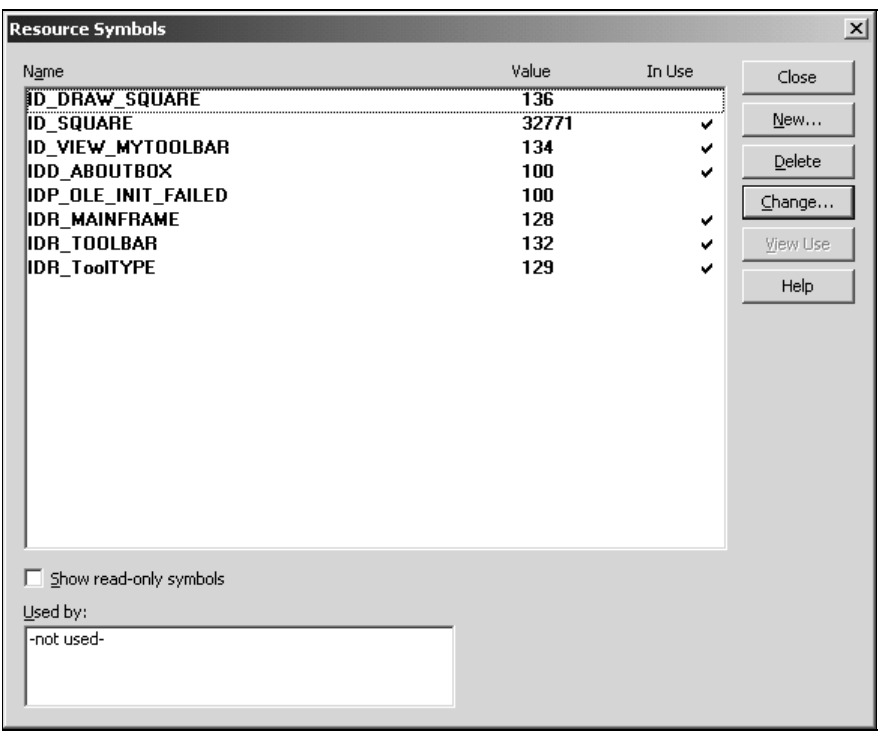


Рис. 9.11. Диалоговое окно **Resource Symbols**

2. Откройте окно **Resource View** (Просмотр ресурсов), раскройте в нем папку **Tool**, щелкните правой кнопкой мыши на папке **Tool.rc** и выберите в появившемся контекстном меню команду **Resource Symbols** (Символы ресурсов). Появится диалоговое окно **Resource Symbols** (Символы ресурсов), изображенное на рис. 9.11.
3. Нажмите кнопку **New** (Создать), появится диалоговое окно **New Symbol** (Новый символ), изображенное на рис. 9.12.

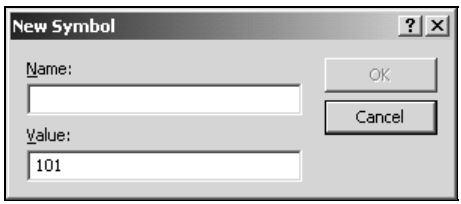


Рис. 9.12. Диалоговое окно **New Symbol**

4. В текстовое поле **Name** (Имя) введите идентификатор ресурса `ID_INDICATOR_X` и нажмите кнопку **OK**. В списке появится новый идентификатор ресурса.
5. Повторите эту операцию для идентификатора ресурса `ID_INDICATOR_Y` и нажмите кнопку **Close** (Закрыть) в диалоговом окне **Resource Symbols** (Символы ресурсов).
6. Раскройте папку **Tool.rc**, а в ней папку **String Table** (Список строковых ресурсов). Дважды щелкните левой кнопкой мыши по значку **String Table (group) [English (U.S.)]**. Откроется окно редактирования строковых ресурсов.
7. Щелкните правой кнопкой мыши по любому идентификатору ресурса в данном окне.
8. В появившемся контекстном меню выберите команду **New String** (Новая строка). В последней строке списка строковых ресурсов появится идентификатор нового строкового ресурса.
9. Щелкните левой кнопкой мыши на идентификаторе нового ресурса. На его месте появится текстовое поле раскрывающегося списка, содержащее данный идентификатор ресурса.
10. Выделите в этом списке идентификатор ресурса `ID_INDICATOR_X`, как это показано на рис. 9.13.
11. Щелкните левой кнопкой мыши на пересечении строки нового строкового ресурса и столбца **Caption** (Заголовок) и введите в появившееся текстовое поле текст "X".
12. Повторите п.п. 7—11 для создания строкового ресурса с идентификатором `ID_INDICATOR_Y` и текстом "Y".

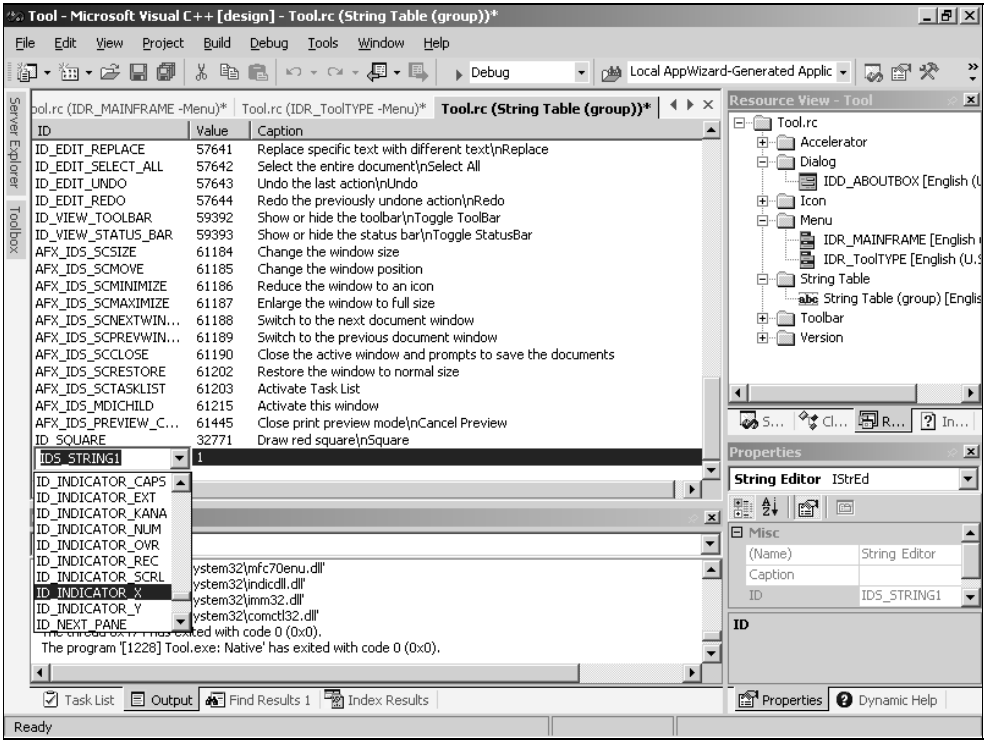


Рис. 9.13. Выбор идентификатора строкового ресурса

13. Откройте окно редактирования файла MainFrm.h и переместите строку `CStatusBar m_wndStatusBar;` на две строки вверх, т. е. из раздела `protected` в раздел `public`.
14. Откройте окно редактирования файла MainFrm.cpp и измените содержимое массива `indicators` в соответствии с приведенным ниже текстом:

```
static UINT indicators[] =
{
  ID_SEPARATOR,     // Индикатор строки состояния
  ID_INDICATOR_X,
  ID_INDICATOR_Y,
};
```

15. В окне **Class View** (Просмотр класса) выделите папку `CToolView` и нажмите кнопку **Messages** (Сообщения). Раскроется список сообщений, обрабатываемых данным классом.
16. Выделите в этом списке сообщение `WM_MOUSEMOVE` и добавьте в приложение его обработчик.

17. В окне **Class View** (Просмотр класса) щелкните правой кнопкой мыши на папке **CToolView** и выберите в раскрывшемся контекстном меню команду **Add | Add Variable** (Добавить | Добавить переменную). Появится диалоговое окно **Add Member Variable Wizard** (Мастер добавления переменной в класс).
18. Введите в текстовое поле раскрывающегося списка **Variable type** (Тип переменной) тип переменной `CPoint`, а в текстовое поле **Variable name** (Имя переменной) — идентификатор переменной `m_Point` и нажмите кнопку **Finish** (Готово).
19. Откройте окно редактирования файла реализации ToolView.h и измените текст карты сообщений класса `CToolView` в соответствии с приведенным ниже текстом.

```
// Функции обработки сообщений
protected:
  DECLARE_MESSAGE_MAP()
public:
  afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
  void OnSquare(void);
  void OnUpdateSquare(CCmdUI *pCmdUI);
  afx_msg void OnMouseMove(UINT nFlags, CPoint point);
  void OnUpdateXPanel(CCmdUI* pCmdUI);
  void OnUpdateYPanel(CCmdUI* pCmdUI);
```

20. Откройте окно редактирования файла реализации ToolView.cpp и измените текст карты сообщений класса `CToolView` в соответствии с приведенным ниже текстом.

```
BEGIN_MESSAGE_MAP(CToolView, CView)
  // 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)
  ON_WM_LBUTTONDOWN()
  ON_WM_MOUSEMOVE()
  ON_COMMAND(ID_SQUARE, OnSquare)
  ON_UPDATE_COMMAND_UI(ID_SQUARE, OnUpdateSquare)
  ON_UPDATE_COMMAND_UI(ID_INDICATOR_X, OnUpdateXPanel)
  ON_UPDATE_COMMAND_UI(ID_INDICATOR_Y, OnUpdateYPanel)
END_MESSAGE_MAP()
```

21. Измените функцию `OnMouseMove` в соответствии с текстом листинга 9.8.

Листинг 9.8. Функция `CToolView::OnMouseMove`

```
// Обработка сообщений о перемещении мыши

void CToolView::OnMouseMove(UINT nFlags, CPoint point)
{
  // Сохранение координат точки
  m_Point  = point;
  CView::OnMouseMove(nFlags, point);
}
```

22. После описания функции `OnMouseMove` поместите текст листинга 9.9.

Листинг 9.9. Функции для работы со строкой состояния

```
// Вывод горизонтальной координаты

void CToolView::OnUpdateXPanel(CCmdUI* pCmdUI)
{
  CString text;
  text.Format("%d", m_Point.x);
  pCmdUI->SetText(text);
}

// Вывод вертикальной координаты
void CToolView::OnUpdateYPanel(CCmdUI* pCmdUI)
{
  CString text;
  text.Format("%d", m_Point.y);
  pCmdUI->SetText(text);
}
```

23. Нажмите клавишу <F5> и запустите приложение на исполнение.
24. Поместите указатель мыши в рабочую область окна документа. В строке состояния появятся координаты указателя мыши, как это показано на рис. 9.14.

Функция `CToolAppView::OnMouseMove` является функцией обработки сообщения о перемещении мыши в рабочей области окна. В ее аргументе `point` содержатся текущие координаты указателя мыши. Поскольку эта информация доступна только в функциях обработки сообщений мыши, то эти координаты сохраняются в переменной `m_Point` для использования в других функциях данного класса.

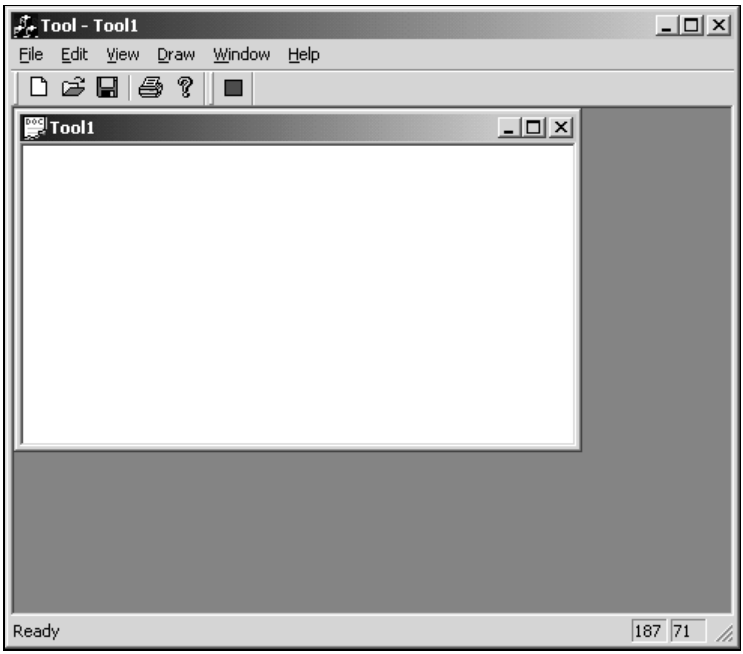


Рис. 9.14. Строка состояния, отображающая координаты курсора мыши

Вывод информации в панели строки состояния осуществляется с использованием макроса карты сообщений `ON_UPDATE_COMMAND_UI`. Поскольку строка состояния не связывается средой программирования с каким-либо классом, то все действия с картой сообщений производятся программистом. Он же пишет и соответствующую функцию обработки сообщения.

Операции, производимые с картой сообщений, достаточно просты и не нуждаются в дальнейших комментариях. Поэтому перейдем сразу к функциям обработки сообщений. Вызовы функций `CToolView::OnUpdateXPanel` и `CToolView::OnUpdateYPanel` включены в цикл обработки сообщений приложения и поэтому обеспечивают оперативное обновление информации в панелях строки состояния. При своем вызове они с использованием функции `CString::Format` преобразуют значение соответствующей координаты указателя мыши, хранящейся в переменой `m_Point`, в текстовый вид, а затем, с использованием функции `CCmdUI::SetText`, выводят этот текст в панели строки состояния.

Примечание

Обратите внимание на то, что в строковых ресурсах `ID_INDICATOR_X` и `ID_INDICATOR_Y` соответствующим символам предшествуют и следуют по два пробела. Это связано с тем, что размер этих строк используется для определения размера соответствующей панели строки состояния. Как видно из рис. 9.14, размер этих панелей достаточен для вывода трех символов.

Предлагаемый Microsoft метод вывода информации в панели строки состояния имеет один существенный недостаток: вывод информации в каждую панель осуществляется в своей функции обработки сообщения. Во многих случаях информация, выводимая в эти панели, взаимосвязана и ее предпочтительно выводить из одной функции.

Чтобы осуществить вывод информации в панели строки состояния из одной функции:

1. Выберите команду **File | Open Solution** (Файл | Открыть решение) и в открывшемся диалоговом окне **Open Solution** (Открыть решение) раскройте папку **Tool**, выделите значок **Tool** и нажмите кнопку **Open** (Открыть).
2. Откройте окно редактирования файла MainFrm.cpp и измените функцию `CMainFrame::OnCreate` в соответствии с текстом, приведенным в листинге 9.10.

Листинг 9.10. Функция `CMainFrame::OnCreate`

```
// Процедура создания главного окна приложения

int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
  // Вызов метода базового класса
  if (CMDIFrameWnd::OnCreate(lpCreateStruct) == -1)
    return -1;

  // Создание стандартной панели инструментов
  if (!m_wndToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD | WS_VISIBLE  |
       CBRS_TOP |  CBRS_GRIPPER  |  CBRS_TOOLTIPS  |  CBRS_FLYBY  |
       CBRS_SIZE_DYNAMIC)  |  | !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
  {
    TRACE0("Failed to create toolbar\n");
    return -1;  // Ошибка при создании
  }

  // Создание пользовательской панели инструментов
  if (!m_MyToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD  | WS_VISIBLE  |
       CBRS_TOP  |  CBRS_GRIPPER  |  CBRS_TOOLTIPS  |  CBRS_FLYBY  |
       CBRS_SIZE_DYNAMIC)  |  | !m_MyToolBar.LoadToolBar(IDR_TOOLBAR))
  {
    TRACE0("Failed to create toolbar\n");
    return -1;  // Ошибка при создании
  }
```

```
  // Создание строки состояния
  if (!m_wndStatusBar.Create(this)  |  |
    !m_wndStatusBar.SetIndicators(indicators,
      sizeof(indicators)/sizeof(UINT)))
  {
    TRACE0("Failed to create status bar\n");
    return -1;  // Ошибка при создании
  }

  // Установка нового идентификатора окна
  SetWindowLong(m_MyToolBar.m_hWnd, GWL_ID, ID_VIEW_MYTOOLBAR);

  // Обеспечение возможности фиксации панелей инструментов
  m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
  m_MyToolBar.EnableDocking(CBRS_ALIGN_ANY);
  EnableDocking(CBRS_ALIGN_ANY);
  DockControlBar(&m_wndToolBar);
  DockControlBar(&m_MyToolBar);

  // Загрузка предыдущего состояния панелей инструментов
  // из системного реестра
  LoadBarState("Local AppWizard-Generated Applications");

  // Инициализация панелей строки состояния
  m_wndStatusBar.SetPaneInfo(m_wndStatusBar.CommandToIndex(ID_INDICATOR_X),
  ID_INDICATOR_X, SBPS_NORMAL, 64);
m_wndStatusBar.SetPaneInfo(m_wndStatusBar.CommandToIndex(ID_INDICATOR_Y),
  ID_INDICATOR_Y, SBPS_NORMAL, 64);

  return 0;
}
```

3. Откройте окно редактирования файла ToolAppView.cpp и после строки `#include "ToolView.h"` вставьте строку

```
#include "MainFrm.h"
```

4. Измените функции обработки сообщений в соответствии с текстом листинга 9.11.

Листинг 9.11. Функции обработки сообщений класса `CToolView`

```
// Обработка сообщений о перемещении мыши

void CToolView::OnMouseMove(UINT nFlags, CPoint point)
{
  // Получение указателя на главное окно приложения
  CMainFrame*  lpFrame  = (CMainFrame*) GetParentOwner();

  // Вывод текста в панели строки состояния
  char  szTemp[16];
  sprintf(szTemp,"%d", point.x);
  lpFrame-> m_wndStatusBar.SetPaneText(lpFrame->
    m_wndStatusBar.CommandToIndex(ID_INDICATOR_X), szTemp);

  sprintf(szTemp,"%d", point.y);

  lpFrame-> m_wndStatusBar.SetPaneText(lpFrame->
    m_wndStatusBar.CommandToIndex(ID_INDICATOR_Y), szTemp);

  // Сохранение координат точки
  m_Point  = point;
  CView::OnMouseMove(nFlags, point);
}

// Вывод горизонтальной координаты
void CToolView::OnUpdateXPanel(CCmdUI* pCmdUI)
{
  CString text;
  text.Format("%d", m_Point.x);
//  pCmdUI->SetText(text);
}

// Вывод вертикальной координаты
void CToolView::OnUpdateYPanel(CCmdUI* pCmdUI)
{
  CString text;
  text.Format("%d", m_Point.y);
//  pCmdUI->SetText(text);
}
```

5. Нажмите клавишу <F5> и запустите приложение на исполнение. Появится главное окно приложения, в котором будет открыто окно документа.
6. Если указатель мыши не расположен в окне документа, в панелях строки состояния будут отображаться символы **X** и **Y**, как это показано на рис. 9.15.

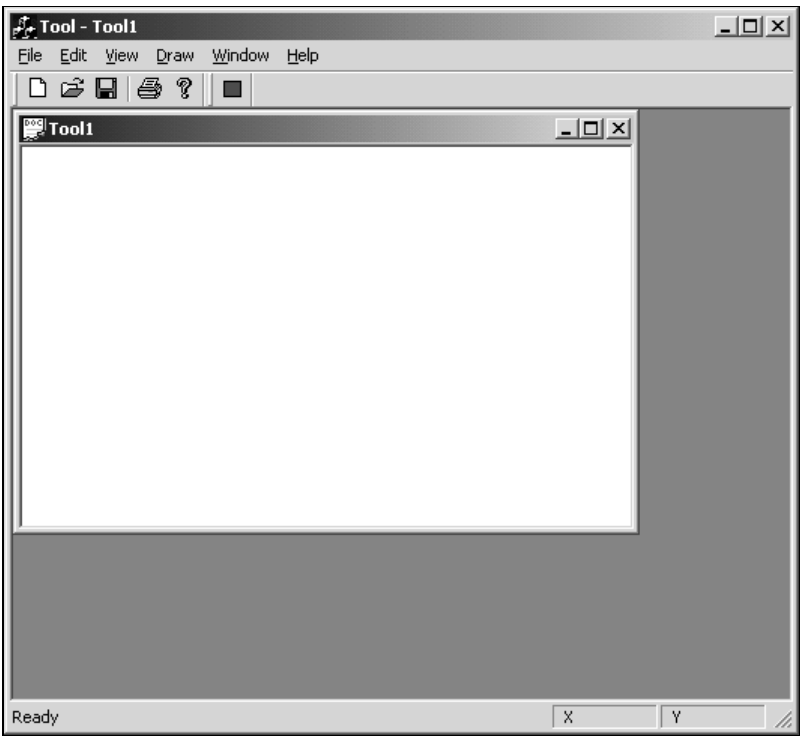


Рис. 9.15. Строка состояния открытого приложения

7. Переместите курсор мыши в окно документа. В панелях строки состояния будут постоянно отображаться текущие координаты указателя мыши, как это показано на рис. 9.16.
8. При перемещении курсора мыши за пределы окна документа в панелях строки состояния будут отображаться координаты точки выхода курсора мыши из окна документа.

Поскольку на предыдущем этапе в функцию `CMainFrame::OnCreate` изменения не вносились и ее текст приведен только сейчас, то описание функций, вызываемых для вывода строки состояния, приведено только здесь. Функция `CStatusBar::Create` создает строку состояния (дочернее окно) и связывает его с объектом класса `CStatusBar`. Функция `CStatusBar::SetIndicators` сопоставляет каждому идентификатору индикатора значение, указанное для него в массиве, на

который указывает ее первый аргумент, загружает строковый ресурс, определенный для каждого идентификатора, и выводит его в соответствующей панели. Для определения размерности массива `indicators` находится его размер в байтах, который затем делится на размер элемента массива. В случае возникновения ошибки при выполнении данных функций выдается сообщение об ошибке и функция `OnCreate` прекращает свою работу, возвращая значение −1.

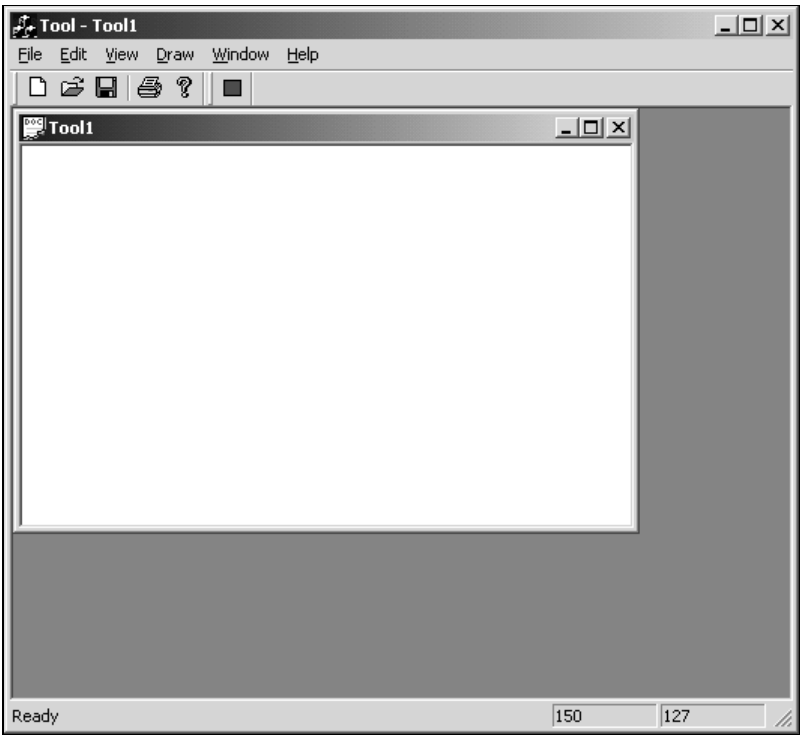


Рис. 9.16. Строка состояния, отображающая координаты курсора мыши

Функция `CStatusBar::SetPaneInfo` устанавливает для панели индикатора новые идентификатор, стиль и ширину. В данном случае она используется для установки новой ширины для каждой из панелей. Применение фиксированной ширины панели в данном случае вызвано тем, что информация в ней будет быстро изменяться. В случае постоянной настройки ширины панели на ширину выводимого в ней текста появится неприятное для глаза мигание панели даже в том случае, когда ширина выводимого в ней текста не будет меняться. Если не предполагается часто менять выводимую в панелях индикаторов информацию, а сама выводимая информация существенно различается по ширине текста, для задания ширины панели можно использовать функцию `CDC::GetTextExtent`,

возвращающую объект класса `CSize`, содержащий текстовые строки. Переменная `cx` данного объекта содержит требуемую ширину строки.

Основные изменения внесены в функцию `CToolAppView::OnMouseMove`. В первом ее операторе с помощью вызова функции `CWnd::GetParentOwner`, получается указатель на ближайшее в иерархии родительское окно, которое само не является дочерним окном. В данном случае это главное окно приложения. После этого горизонтальная координата позиции курсора преобразуется в текстовую форму и используется в качестве аргумента функции `CStatusBar::SetPaneText`, обновляющей текст в панели индикаторов. В качестве первого аргумента данной функции необходимо указать индекс панели, в которую нужно вывести текст. Для получения этой информации используется функция `CStatusBar::CommandToIndex`, позволяющая получить индекс индикатора по заданному идентификатору. После этого подобная операция производится с вертикальной координатой позиции курсора. И в завершение функции вызывается соответствующий метод базового класса.

Для того чтобы сохранить в приложении возможность быстро вернуться к рекомендуемому Microsoft методу вывода информации в панели строки состояния, в данном приложении были сохранены все функции его предыдущей версии. Однако, чтобы в одну и ту же панель не выводился один и тот же текст различными способами, в функциях `CToolView::OnUpdateXPanel` и `CToolView::OnUpdateYPanel` заремаркированы вызовы функции `CCmdUI::SetText`, осуществляющей вывод этой информации.

Глава 10

Печать документов и организация прокрутки в окне

Печать документа в операционной системе Windows до появления библиотеки MFC представляла собой достаточно сложную задачу. Это было связано с необходимостью обеспечения аппаратной независимости создаваемых программ при достаточно скудных программных средствах.

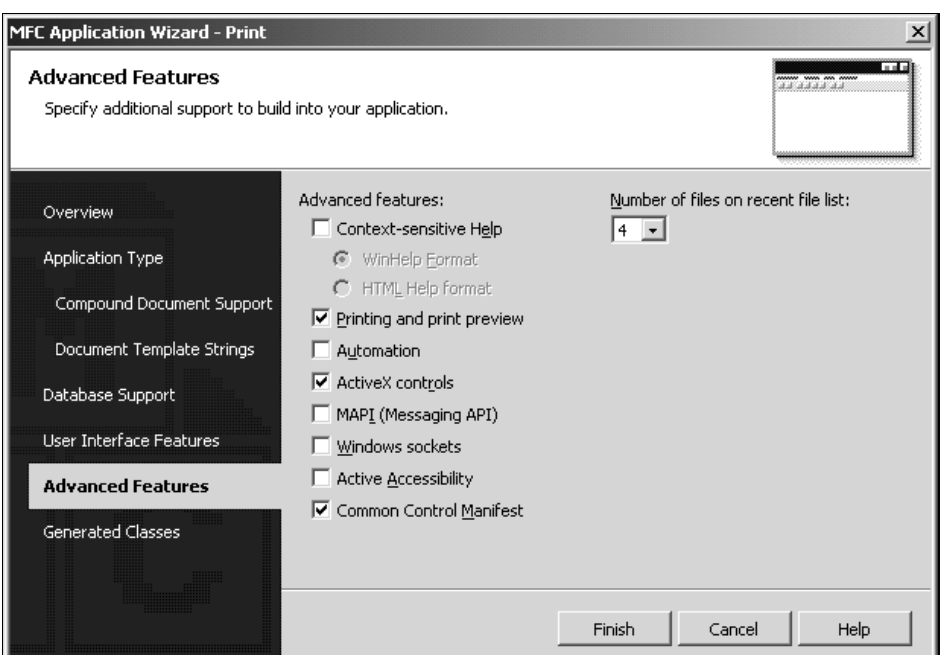


Рис. 10.1. Диалоговое окно **MFC Application Wizard - Print**

Теперь, для того, чтобы обеспечить возможность печати документа и его предварительного просмотра достаточно только не сбрасывать флажок **Printing and print preview** (Печать и предварительный просмотр) во вкладке **Advanced Features** (Дополнительные возможности) мастера MFC Application Wizard, приведенной

на рис. 10.1. Однако не все так просто. Дело в том, что по умолчанию печать документа производится "точка в точку", т. е. элементу изображения на экране соответствует точка в распечатке. Поскольку большинство принтеров имеют разрешение намного выше, чем разрешение большинства мониторов, то такая распечатка приводит к выводу на печать слишком мелких изображений.

Кроме того, многие документы состоят из нескольких страниц, и, поскольку не существует единого принципа разбиения любого документа на страницы, то эта задача полностью ложится на плечи пользователя.

Другим вопросом, тесно связанным с разбиением документа на страницы, является организация его прокрутки в окне, поскольку разбиение на страницы можно условно представить себе, как последовательную прокрутку документа в окне.

Организация прокрутки в окне

Прежде чем приступить к печати документа, его нужно предварительно создать. Поскольку данная глава в основном посвящена распечатке многостраничных документов, то этот документ не будет полностью помещаться в окно на экране, что приводит к необходимости организации *прокрутки изображения в окне* для просмотра всего документа.

Текст демонстрационного приложения, реализующего печать документа и его прокрутку в окне, расположен в папке Print дискеты, прилагаемой к данной книге. Чтобы самостоятельно создать данное приложение:

1. Выполните все операции, указанные в *главе 1* для создания многооконного приложения с именем Print, но в диалоговом окне **MFC Application Wizard - Print** (Мастер создания приложений MFC) не нажимайте кнопку **Finish** (Готово).
2. Вместо этого, раскройте в нем вкладку **Generated Classes** (Создаваемые классы), как это показано на рис. 10.2.
3. В раскрывающемся списке **Base class** (Базовый класс) выделите имя базового класса `CScrollView` и только после этого нажмите кнопку **Finish** (Готово).
4. Раскройте окно **Class View** (Просмотр класса). Раскройте папку **Print** (Печать) и щелкните правой кнопкой мыши по имени класса `CPrintDoc`.
5. В раскрывшемся контекстном меню выберите команду **Add | Add Variable** (Добавить | Добавить переменную). Появится диалоговое окно **Add Member Variable Wizard - Print** (Мастер добавления переменной в класс), изображенное на рис. 10.3.
6. В раскрывающемся списке **Variable type** (Тип переменной) выделите тип переменной `int`, в текстовое поле **Variable name** (Имя переменной) введите идентификатор переменной `m_nRects` и нажмите кнопку **Finish** (Готово). В класс будет добавлена новая переменная и откроется окно редактирования файла PrintAppDoc.cpp.
7. Измените функцию `Serialize` в соответствии с текстом листинга 10.1.

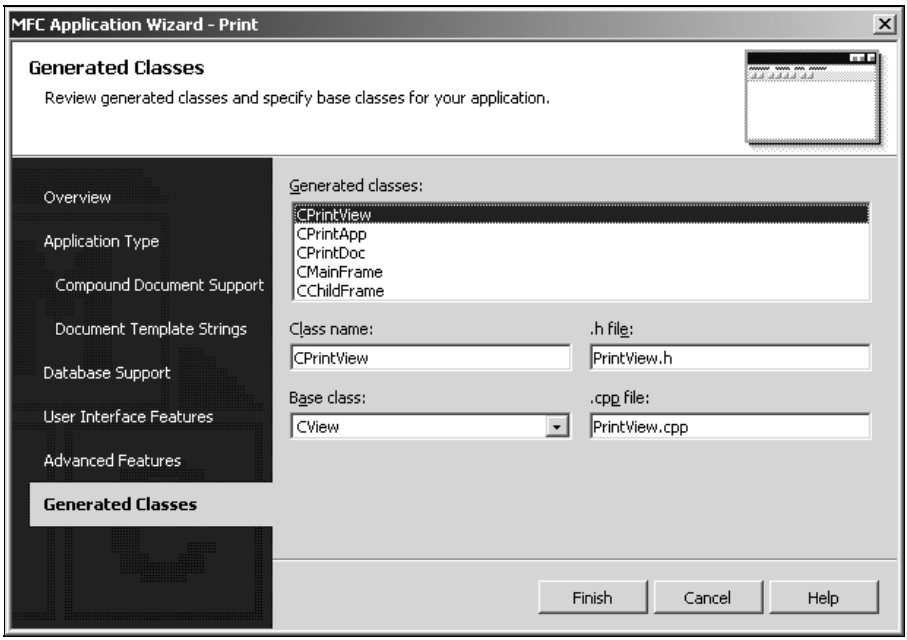


Рис. 10.2. Диалоговое окно **MFC AppWizard - Print**

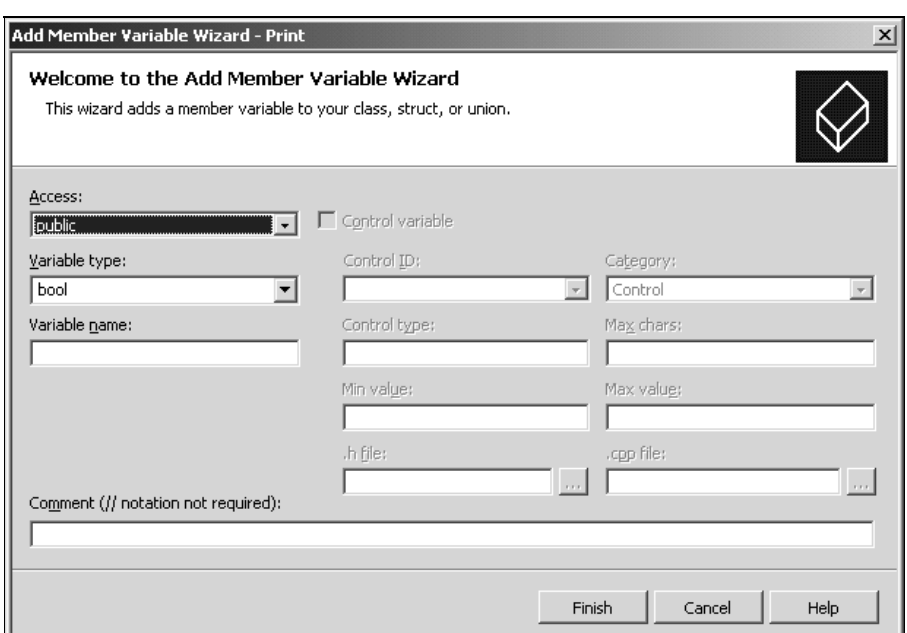


Рис. 10.3. Диалоговое окно **Add Member Variable Wizard - Print**

Листинг 10.1. Функция `CPrintDoc::Serialize`

```
// Работа с файловыми потоками в классе CPrintDoc

void CPrintDoc::Serialize(CArchive& ar)
{
  if (ar.IsStoring())
  {
    // Сохранение информации
    ar << m_nRects;
  }
  else
  {
    // Загрузка информации
    ar >> m_nRects;
  }
}
```

8. В окне **Class View** (Просмотр класса) щелкните правой кнопкой мыши на папке `CPrintView` и выберите в появившемся контекстном меню команду **Properties** (Свойства). Откроется окно **Properties** (Свойства).
9. В окне **Properties** (Свойства) нажмите кнопку **Messages** (Сообщения). Раскроется список сообщений, обрабатываемых данным классом.
10. В раскрывшемся списке выделите идентификатор сообщения `WM_LBUTTONDBLCLK` и добавьте в приложение функцию обработки данного сообщения. При этом откроется окно редактирования файла PrintView.cpp, а текстовый курсор будет располагаться в заготовке данной функции.
11. Измените функцию `OnLButtonDblClk` в соответствии с текстом листинга 10.2.

Листинг 10.2. Функция `CPrintView::OnLButtonDblClk`

```
// Функции обработки сообщений класса CPrintView

// Увеличение числа квадратов
void CPrintView::OnLButtonDblClk(UINT nFlags, CPoint point)
{
  CPrintDoc*  lpDoc = GetDocument();
  ASSERT_VALID(lpDoc);
  lpDoc-> m_nRects++;
  Invalidate();
```

```
  CScrollView::OnLButtonDblClk(nFlags, point);
}
```

12. Измените функцию `OnDraw` в соответствии с текстом листинга 10.3.

Листинг 10.3. Функция `CPrintView::OnDraw`

```
// Вывод графической информации в классе CPrintView

void CPrintView::OnDraw(CDC* pDC)
{
  CPrintDoc* pDoc = GetDocument();
  ASSERT_VALID(pDoc);

  // Вывод квадратов
  CPen      pen(PS_SOLID, 0, RGB(0,0,0));
  CPen*      oldPen = pDC-> SelectObject(&pen);

  for(int i=0; i < pDoc-> m_nRects; i++)
    pDC-> Rectangle(50, 50+500*i, 250, 250+500*i);
  pDC-> SelectObject(oldPen);

  // Настройка параметров прокрутки
  CSize  docSize(300, 300 + 500*pDoc-> m_nRects);
  CRect  rect;
  GetClientRect(&rect);
  CSize  pageSize(rect.right, rect.bottom);
  CSize  lineSize(0, 500);
  SetScrollSizes(MM_TEXT, docSize, pageSize, lineSize);
}
```

13. Нажмите клавишу <F5> и запустите приложение на исполнение.
14. Дважды щелкните левой кнопкой мыши в окне документа. В нем появится квадрат, а вдоль правой его границы расположится полоса прокрутки, как это показано на рис. 10.4.
15. Повторите эту операцию. Размер бегунка полосы прокрутки уменьшится.
16. Щелкните левой кнопкой мыши по стрелке вниз в полосе прокрутки. В рабочей области окна ничего не изменится, но бегунок полосы прокрутки переместится вниз, как это показано на рис. 10.5. Сохранение изображения в окне связано с тем, что размер строки выбран равным шагу расстановки квадратов в окне, что приводит к возникновению "стробоскопического эффекта".

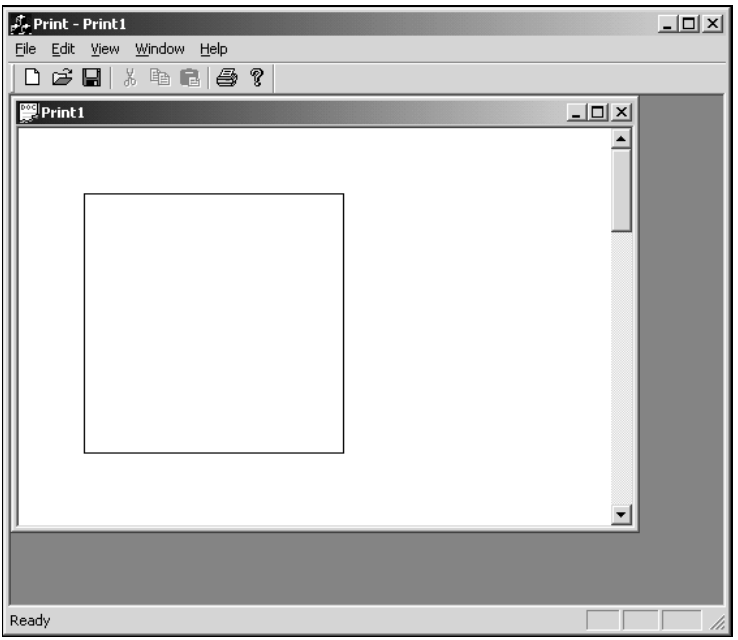


Рис. 10.4. Полоса прокрутки в окне документа

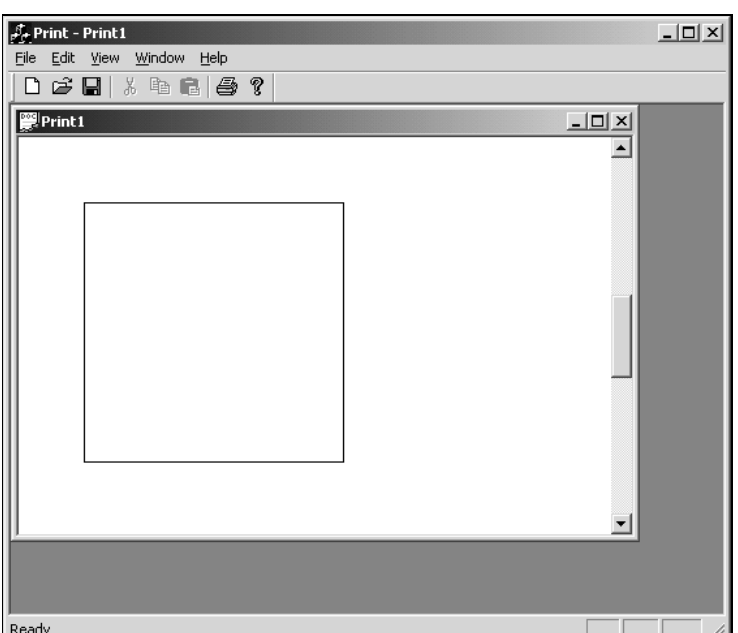


Рис. 10.5. Прокрутка документа вниз

17. Щелкните левой кнопкой мыши в полосе прокрутки выше бегунка. Изображение в окне прокрутится на один экран вверх и примет вид, изображенный на рис. 10.6. В данном случае не наблюдается "стробоскопического эффекта", поскольку размер экрана не связан с шагом расстановки квадратов в окне.

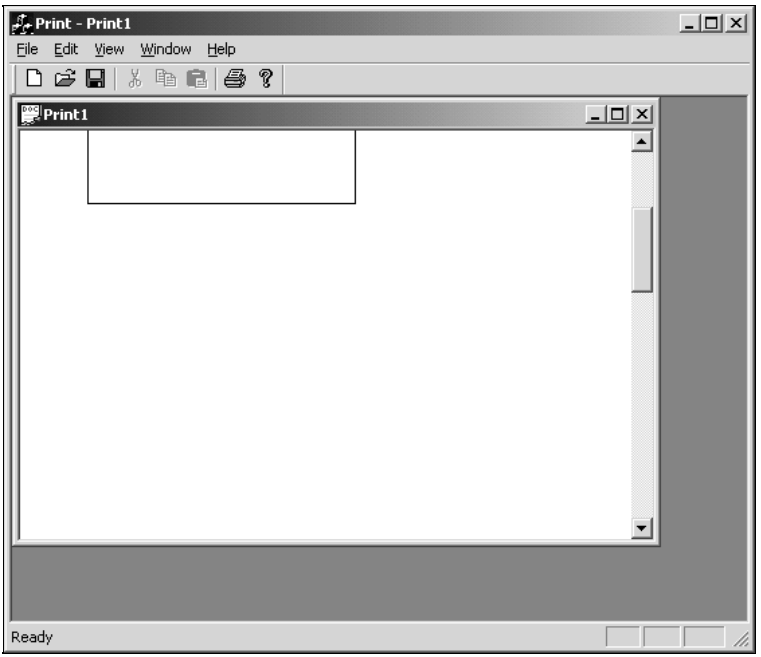


Рис. 10.6. Прокрутка документа вверх

18. Нажмите кнопку **Save** (Сохранить) на панели инструментов. Появится диалоговое окно **Save As** (Сохранить как).
19. Оставьте предлагаемое имя файла без изменений и нажмите кнопку **Save** (Сохранить).
20. Закройте окно документа и нажмите кнопку **Open** (Открыть) в панели инструментов. Появится диалоговое окно **Open** (Открыть).
21. Выделите в списке файлов имя файла сохраненного документа и нажмите кнопку **Open** (Открыть). Откроется окно документа, имеющее вертикальную полосу прокрутки, в котором выведены квадраты, а бегунок полосы прокрутки стоит в крайнем верхнем положении.

Как видно из приведенного выше примера, практически всю работу по созданию в окне полосы прокрутки приложение берет на себя. Единственное, что требуется от пользователя, это указать параметры прокрутки в окне.

Изменения, внесенные в функцию `CPrintAppDoc::Serialize`, позволяют сохранять в архиве число квадратов, выведенных в окне, и не затрагивают вопросов прокрутки изображения.

Функция `CPrintAppView::OnLButtonDblClk` используется для увеличения числа квадратов в окне и тоже не затрагивает вопросов прокрутки изображения. Для выполнения этой операции прежде всего с помощью вызова функции `CView::GetDocument` получается указатель на объект класса документа, связанный с данным объектом класса представления, и проверяется на допустимость. В случае отрицательного исхода проверки в отладочной версии программы выполнение данной функции будет прекращено. С использованием полученного указателя обеспечивается доступ к переменной `m_nRects`, хранящей число квадратов в окне. Значение этой переменной увеличивается на единицу и вызывается функция `CWnd::Invalidate`, приводящая к перерисовке рабочей области окна.

Перерисовка рабочей области окна выполняется функцией `OnDraw`. В этой функции так же, как и в описанной выше функции `OnLButtonDblClk`, получается и проверяется указатель на объект класса документа, связанный с данным объектом класса представления. После этого создается объект класса `CPen` и выбирается в текущий контекст устройства, причем указатель на старый объект класса запоминается в переменной `oldPen`. Затем производится рисование квадратов с использованием функции `CDC::Rectangle`. После завершения операции рисования в контекст устройства выбирается старый объект класса `CPen`.

После этого производится вычисление параметров прокрутки изображения в окне. Эти параметры записываются в объекты класса `CSize`.

В переменной `docSize` хранится размер всего отображаемого документа. Он определяется границами выводимых квадратов и окружающей их рамкой, шириной в 50 логических единиц.

В переменной `pageSize` хранится размер страницы. Под страницей понимается шаг перемещения изображения по горизонтали или по вертикали при щелчке левой кнопкой мыши в полосе прокрутки за пределами бегунка. В данном случае он равен размеру рабочей области окна, получаемой с помощью вызова функции `CWnd::GetClientRect`.

В переменной `lineSize` хранится размер строки. Под строкой понимается шаг перемещения изображения по горизонтали или по вертикали при щелчке левой кнопкой мыши по стрелкам полосы прокрутки.

Эти переменные передаются в качестве аргументов в функцию `CScrollView::SetScrollSizes`. Кроме указанных аргументов в данной функции задается режим отображения. В качестве режима отображения в данном случае выбран `MM_TEXT`, при котором логическая единица соответствует элементу изображения, а ось Y направлена сверху вниз.

Режимы отображения

Практически все функции GDI требуют определения координат или размеров каких-либо графических объектов. Данные координаты задаются в так называемых логических единицах. Windows преобразует эти логические координаты и размеры в физические координаты (элементы изображения) устройства, используемого для вывода информации. При этом преобразовании используются первые пять атрибутов контекста устройства, относящиеся к группе атрибутов режимов отображения: начальные координаты, направление осей координат и масштаб координат для каждой оси.

Характеристики различных режимов отображения приведены в табл. 10.1.

Таблица 10.1. *Характеристики режимов отображения*

Режим отображения	Логические единицы	Направление по оси X	Направление по оси Y
`MM_TEXT`	1 элемент изображения	Возрастает слева направо	Возрастает сверху вниз
`MM_HIMETRIC`	0.01 мм	Возрастает слева направо	Возрастает снизу вверх
`MM_TWIPS`	1/1440 дюйма	Возрастает слева направо	Возрастает снизу вверх
`MM_HIENGLISH`	0.001 дюйма	Возрастает слева направо	Возрастает снизу вверх
`MM_LOMETRIC`	0.1 мм	Возрастает слева направо	Возрастает снизу вверх
`MM_LOENGLISH`	0.01 дюйма	Возрастает слева направо	Возрастает снизу вверх
`MM_ISOTROPIC`	Определяется пользователем (X=Y)	Определяется пользователем	Определяется пользователем
`MM_ANISOTROPIC`	Определяется пользователем (X!=Y)	Определяется пользователем	Определяется пользователем

Windows использует три системы координат. Во всех системах координат устройств отображения координаты измеряются в элементах изображения, ось X направлена слева направо, а ось Y — сверху вниз.

При использовании всего экрана дисплея работа ведется в экранных координатах. При этом начало координат находится в левом верхнем углу экрана. Экранные координаты используются в сообщении `WM_MOVE` и в функциях `CreateWindow`, `MoveWindow`, `GetCursorPos`, `SetCursorPos`, `GetWindowRect` и `WindowFromPoint`. Если для получения контекста устройства всего экрана дисплея использовалась глобальная функция `CreateDC`, то при использовании функций GDI логические координаты преобразуются в экранные координаты.

В оконной системе координат начало координат находится в левом верхнем углу рамки окна. Данная система координат описывает все окно, включая меню, полосы прокрутки и заголовок. Для использования оконной системы координат следует получать контекст устройства с применением функции `GetWindowDC`. При этом функции GDI, использующие данный контекст, преобразуют логические координаты в оконные.

Пользовательская система координат связана с рабочей областью окна. Начало координат в данной системе координат расположено в левом верхнем углу рабочей области окна. При получении контекста устройства с помощью вызова функций GDI `GetDC` и `BeginPaint`, использующие данный контекст функции, преобразуют логические координаты в координаты рабочей области окна. Для преобразования экранных координат в пользовательские применяется функция `CWnd::ScreenToClient`.

Поскольку пользователь в основном выводит информацию в рабочую область окна, то ему практически во всех случаях приходится работать в пользовательской системе координат. При выводе информации на печать эта система координат претерпевает определенные изменения, рассмотренные в следующем разделе.

Распечатка и предварительный просмотр

Как уже говорилось выше, возможности печати документов автоматически закладываются в приложение при его создании. Однако существуют несколько задач, связанных с печатью документа, которые должны быть решены самим пользователем.

Работа с окном предварительного просмотра печати

Одним из полезных свойств, предоставляемых библиотекой MFC при работе с распечатываемым документом, является *предварительный просмотр печати.* Эта возможность реализуется в виде специального окна, позволяющего оценить внешний вид каждой страницы распечатываемого документа. Чтобы вывести на экран окно предварительного просмотра в приложении Print:

1. Откройте приложение PrintApp.
2. Нажмите клавишу <F5> и запустите его на исполнение.
3. Дважды щелкните левой кнопкой мыши в рабочей области окна документа для вывода квадрата.
4. Выберите команду меню **File | Print Preview** (Печать | Предварительный просмотр). Появится окно предварительного просмотра, изображенное на рис. 10.7.

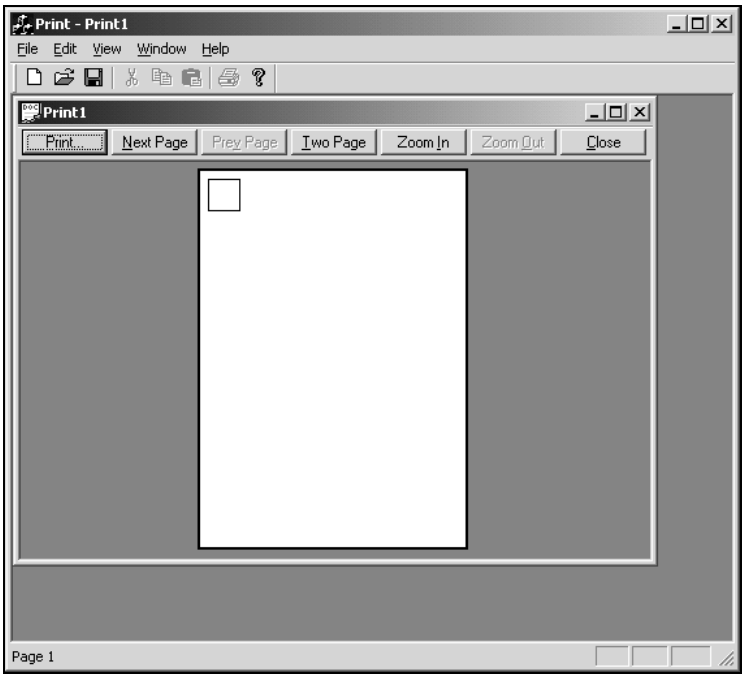


Рис. 10.7. Исходное окно предварительного просмотра

5. Переместите курсор мыши в область печати. Он примет вид увеличительного стекла.
6. Нажмите левую кнопку мыши. Изображение листа увеличится, и в нем появятся горизонтальная и вертикальная полосы прокрутки, как это показано на рис. 10.8.
7. Повторите эту операцию еще раз. Изображение листа еще больше увеличится, как это показано на рис. 10.9.
8. Нажмите кнопку **Zoom Out** (Сжатие). Размеры листа уменьшатся и окно примет вид, изображенный на рис. 10.8.
9. Еще раз нажмите кнопку **Zoom Out** (Сжатие), приводя тем самым изображение к его исходному масштабу, и нажмите кнопку **Two Page** (Две страницы).
10. На экране появятся две страницы, как это показано на рис. 10.10.
11. Нажмите кнопку **Close** (Закрыть) и закройте окно предварительного просмотра.
12. Закройте приложение и откройте окно редактирования файла PrintAppView.cpp.
13. Измените функцию `OnDraw` в соответствии с текстом листинга 10.4.

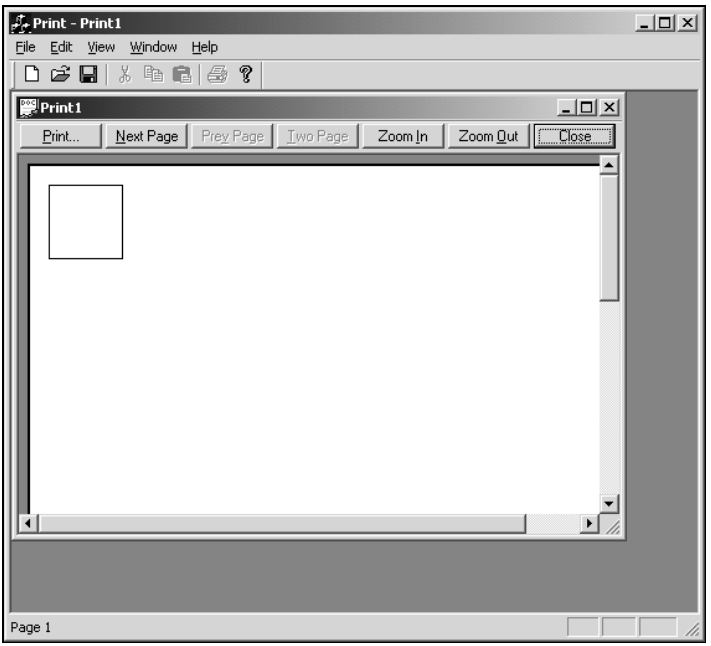


Рис. 10.8. Увеличенный масштаб окна предварительного просмотра

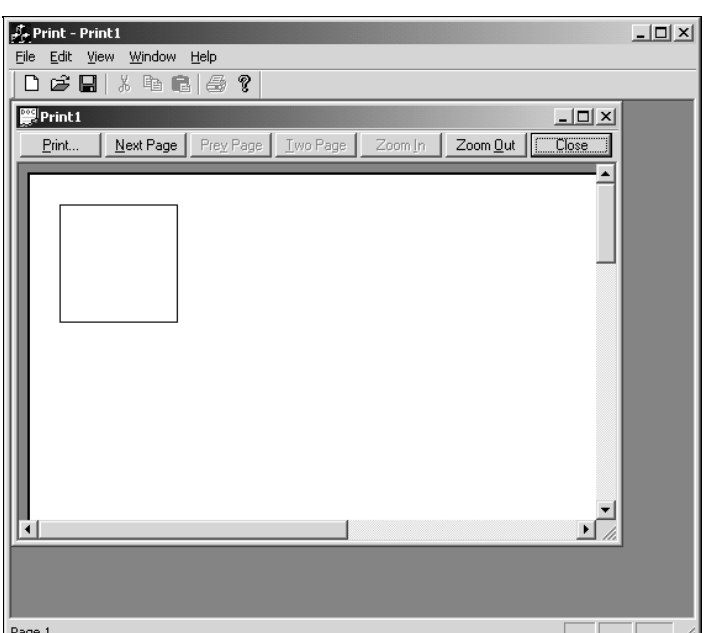


Рис. 10.9. Максимальный масштаб окна предварительного просмотра

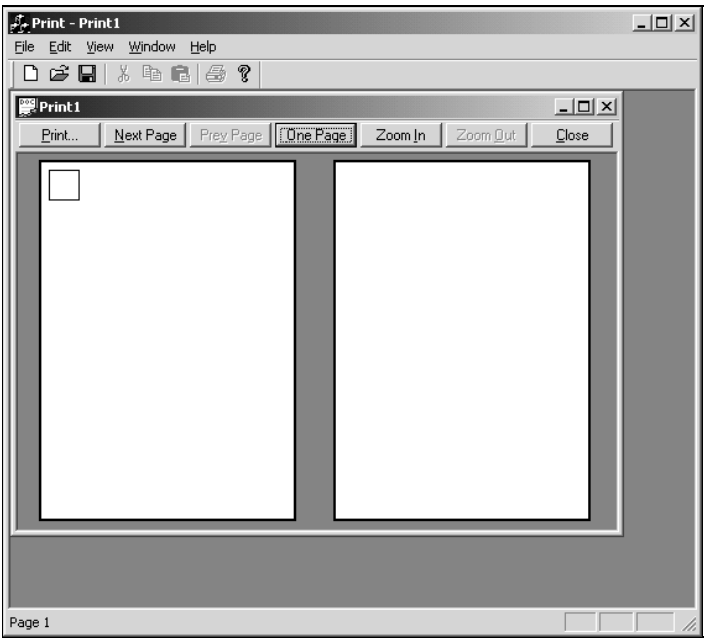


Рис. 10.10. Окно предварительного просмотра в режиме одновременного просмотра двух страниц

Листинг 10.4. Функция `CPrintView::OnDraw`

```
// Вывод графической информации в классе CPrintView

void CPrintView::OnDraw(CDC* pDC)
{
  CPrintDoc* pDoc = GetDocument();
  ASSERT_VALID(pDoc);

  // Вывод квадратов
  CPen      pen(PS_SOLID, 0, RGB(0,0,0));
  CPen*      oldPen = pDC-> SelectObject(&pen);
  for(int i=0; i < pDoc-> m_nRects; i++)
    pDC-> Rectangle(50, — 50-500*i, 250, -250-500*i);
  pDC-> SelectObject(oldPen);

  // Настройка параметров прокрутки
  CSize  docSize(300, 100 + 500*pDoc-> m_nRects);
```

```
  CRect  rect;
  GetClientRect(&rect);
  CSize  pageSize(rect.right, rect.bottom);
  CSize  lineSize(0, 500);
  SetScrollSizes(MM_LOENGLISH, docSize, pageSize, lineSize);
}
```

14. Повторите операции, описанные в п.п. 2—4. Появится окно предварительного просмотра, показанное на рис. 10.11.

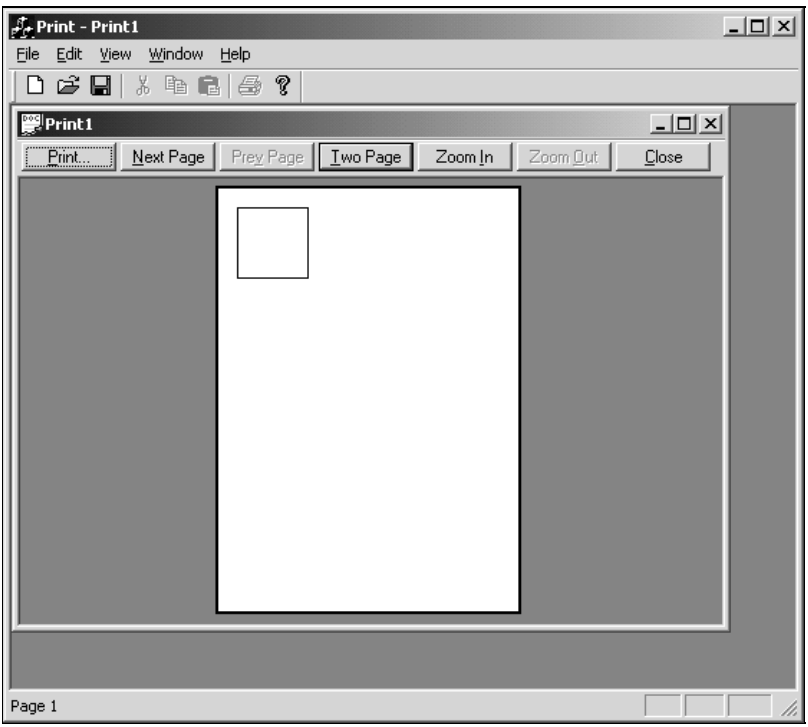


Рис. 10.11. Окно предварительного просмотра печати

15. Закройте окно предварительного просмотра печати и еще дважды щелкните левой кнопкой мыши в рабочей области окна документа.
16. Откройте окно предварительного просмотра печати и нажмите кнопку **Two Page** (Две страницы). Окно предварительного просмотра примет вид, изображенный на рис. 10.12.
17. Закройте окно предварительного просмотра и приложение.

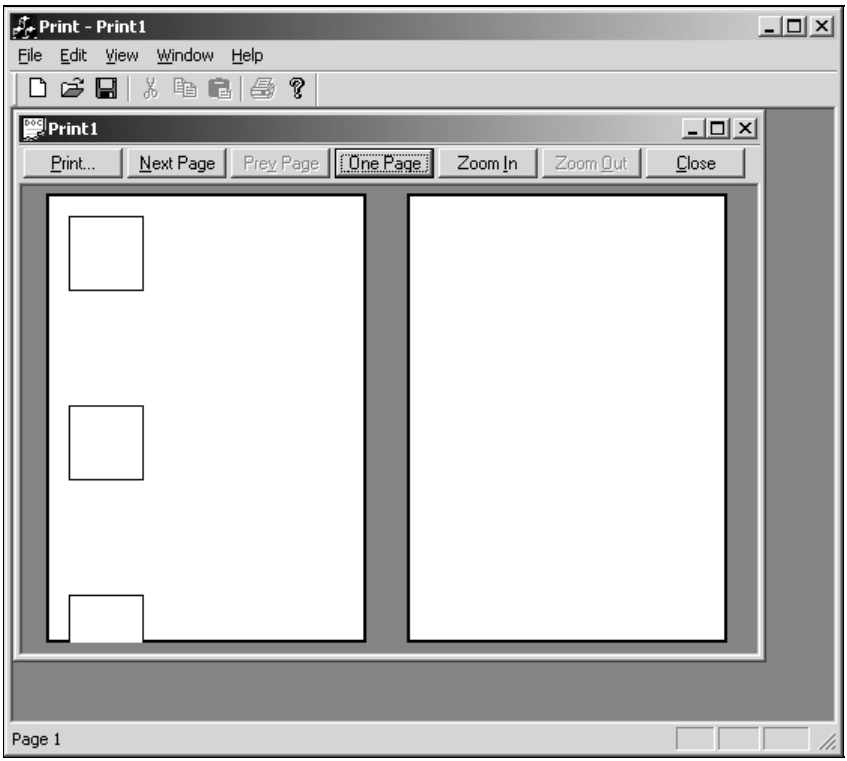


Рис. 10.12. Окно предварительного просмотра печати с двумя страницами

Как видно из сравнения рис. 10.7 и 10.11, режим отображения `MM_LOENGLISH` позволяет существенно увеличить размеры печатаемого изображения по сравнению с режимом `MM_TEXT`. Кроме того, существенным преимуществом режима `MM_LOENGLISH` является его независимость от разрешения принтера, на котором будет распечатываться данное изображение.

Однако гораздо больший интерес представляет рис. 10.12. Он показывает, что приложение Print в его настоящем виде не позволяет распечатывать многостраничные документы, т. е. переносить остаток изображения, не распечатанного на текущей странице, на следующую страницу. Исправлению данного недостатка будет посвящен следующий раздел.

Распечатка многостраничного документа

Распечатка многостраничного документа в чем-то аналогична операции прокрутки изображения в окне, где каждая страница документа представляет собой окно, в которое выводится соответствующая часть документа. Поскольку разбиение документа на страницы при его печати должно производиться автоматически и при этом должны соблюдаться некоторые специфические правила, ко-

торые достаточно сложно формализовать для всех случаев жизни, данная операция практически полностью ложится на плечи программиста.

Чтобы обеспечить в приложении Print возможность распечатки многостраничного документа:

1. Откройте приложение Print.
2. Откройте окно редактирования файла PrintView.cpp.
3. Измените функцию `OnBeginPrinting` в соответствии с текстом листинга 10.5.

Листинг 10.5. Функция `CPrintView::OnBeginPrinting`

```
// Инициализация процесса печати

void CPrintView::OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo)
{
  // Получение указателя на объект класса
  // документа и проверка его
  CPrintDoc*  lpDoc = GetDocument();
  ASSERT_VALID(lpDoc);

  // Определение числа страниц в документе
  pInfo-> SetMaxPage((pDC-> GetDeviceCaps(LOGPIXELSY)*(1 + 5*lpDoc->
    m_nRects))/pDC-> GetDeviceCaps(VERTRES) + 1);
}
```

4. Нажмите клавишу <F5> и запустите приложение на исполнение.
5. Выведите в окно документа три квадрата, откройте окно предварительного просмотра печати и нажмите кнопку **Two Page** (Две страницы). Окно предварительного просмотра примет вид, изображенный на рис. 10.13.
6. Закройте окно предварительного просмотра печати и закройте приложение.
7. В окне **Class View** (Просмотр класса) выделите папку `CPrintView` и нажмите кнопку **Overrides** (Перегружаемые виртуальные функции базовых классов) в окне **Properties** (Свойства). Раскроется список виртуальных функций базовых классов класса `CPrintView`.
8. Выделите в этом списке имя функции `OnPrepareDC`. В соответствующем текстовом поле появится значок раскрывающегося списка.
9. Раскройте этот список и выделите его единственную строчку. В класс `CPrintView` будет добавлена функция обработки сообщения `OnPrepareDC`, откроется окно редактирования файла PrintView.cpp, а текстовый курсор будет расположен в заготовке данной функции.
10. Измените функцию `OnPrepareDC` в соответствии с текстом листинга 10.6.

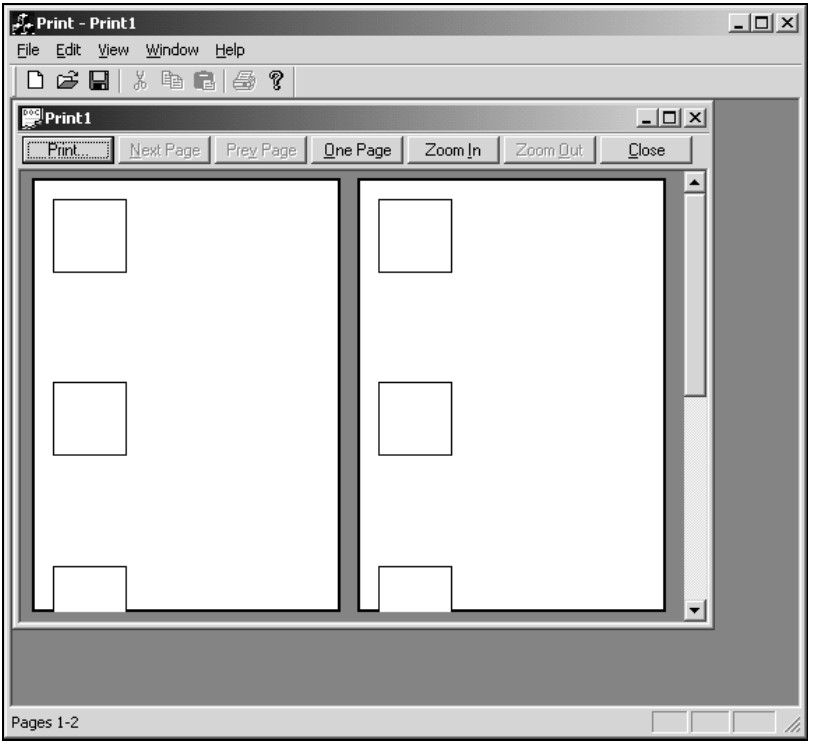


Рис. 10.13. Окно предварительного просмотра печати

Листинг 10.6. Функция `CPrintView::OnPrepareDC`

```
// Подготовка контекста устройства печати

void CPrintView::OnPrepareDC(CDC* pDC, CPrintInfo* pInfo)
{
  if(pDC-> IsPrinting())
  {
    // Производится печать документа

    ASSERT(m_nMapMode > 0);       // Проверка режима  отображения
    pDC->SetMapMode(m_nMapMode); // Задание режима отображения

    // Установка координат нулевой точки отображения
    pDC-> SetViewportOrg(0, -pDC-> GetDeviceCaps(VERTRES)*
      (pInfo->m_nCurPage — 1));
```

```
    // Вызов метода базового класса
    CView::OnPrepareDC(pDC, pInfo);
  }
  else
    //  Вывод на экран
    CScrollView::OnPrepareDC(pDC, pInfo);
}
```

11. Повторите операции, описанные в п.п. 4 и 5. Окно предварительного просмотра примет вид, изображенный на рис. 10.14.

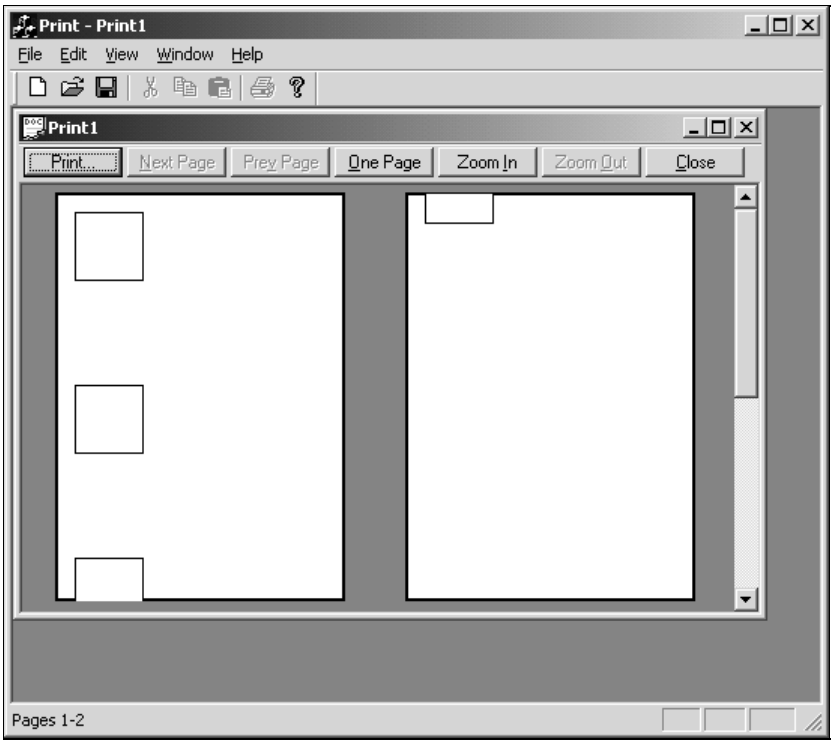


Рис. 10.14. Окно предварительного просмотра печати с учетом листинга 10.6

12. Закройте окно предварительного просмотра печати и закройте приложение.
13. В окне **Class View** (Просмотр класса) выделите папку `CPrintView` и нажмите кнопку **Overrides** (Перегружаемые виртуальные функции базовых классов) в окне **Properties** (Свойства). Раскроется список виртуальных функций базовых классов класса `CPrintView`.

14. Выделите в этом списке имя функции `OnPrepareDC`. В соответствующем текстовом поле появится значок раскрывающегося списка.
15. Раскройте этот список и выделите в нем первую строку (начинающуюся с <**Delete**>). Функция обработки сообщения `OnPrepareDC` будет удалена из класса `CPrintView`.

Примечание

В Visual C++ 7.0 при удалении функции из класса ее объявление в классе заголовка и ее реализация ремаркируются. В Visual C++ 6.0 при удалении функции ее объявление удалялось из файла заголовка и карты сообщений, а задача уничтожения ее реализации возлагалась на программиста.

16. В окне **Properties** (Свойства) выделите имя функции `OnPrint` и добавьте в класс `CPrintView` соответствующую функцию обработки сообщения.
17. В окне редактирования файла PrintView.cpp измените функцию `OnPrint` в соответствии с текстом листинга 10.7.

Листинг 10.7. Функция `CPrintView::OnPrint`

```
// Печать документа

void CPrintView::OnPrint(CDC* pDC, CPrintInfo* pInfo)
{
  LOGFONT    logFont;
  CFont      font;
  CString    Out;

  // Инициализация объекта структуры LOGFONT
  logFont.lfHeight    = 24;
  logFont.lfWidth    = 0;
  logFont.lfEscapement    = 0;
  logFont.lfOrientation  = 0;
  logFont.lfWeight    = FW_NORMAL;
  logFont.lfItalic    = 0;
  logFont.lfUnderline    = 0;
  logFont.lfStrikeOut    = 0;
  logFont.lfCharSet    = ANSI_CHARSET;
  logFont.lfOutPrecision  = OUT_DEFAULT_PRECIS;
  logFont.lfClipPrecision  = CLIP_DEFAULT_PRECIS;
  logFont.lfQuality    = PROOF_QUALITY;
  logFont.lfPitchAndFamily  = VARIABLE_PITCH  |  FF_ROMAN;
  strcpy(logFont.lfFaceName, "MS Sans Serif");
```

```
  // Создание шрифта
  font.CreateFontIndirect(&logFont);

  // Выбор шрифта в контекст устройства
  CFont*  oldFont = pDC-> SelectObject(&font);

  // Формирование текста колонтитула
  sprintf(Out.GetBuffer(16),"Страница № %d", pInfo-> m_nCurPage);
  Out.ReleaseBuffer();

  // Установка координат нулевой точки отображения
  pDC-> SetViewportOrg(0, ((pInfo-> m_rectDraw.top + 50)*pDC->
     GetDeviceCaps(LOGPIXELSY))/100 — pDC->
     GetDeviceCaps(VERTRES)*(pInfo->m_nCurPage — 1));

  // Вызов метода базового класса
  CScrollView::OnPrint(pDC, pInfo);

  // Очистка места под колонтитул
  CBrush  wtBrush(RGB(255, 255, 255));
  CRect  rect = pInfo-> m_rectDraw;
  rect.bottom  = pInfo-> m_rectDraw.bottom*(pInfo->m_nCurPage — 1);
  rect.top  = rect.bottom + 60;
  pDC-> FillRect(rect, &wtBrush);

  // Вывод колонтитула
  pDC-> TextOut(0, rect.bottom + 36, Out);
  pDC-> SelectObject(oldFont);
}
```

18. Измените функцию `OnBeginPrinting` в соответствии с текстом листинга 10.8.

Листинг 10.8. Функция `CPrintView::OnBeginPrinting`

```
// Инициализация процесса печати

void CPrintView::OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo)
{
  // Получение указателя на объект класса
  // документа и проверка его
```

```
  CPrintDoc*  lpDoc = GetDocument();
  ASSERT_VALID(lpDoc);

  // Определение числа страниц в документе
  pInfo-> SetMaxPage((pDC-> GetDeviceCaps(LOGPIXELSY)*(1 + 5*lpDoc->
    m_nRects))/(pDC-> GetDeviceCaps(VERTRES) — 50) + 1);
}
```

19. Повторите операции, описанные в п.п. 4 и 5. Окно предварительного просмотра примет вид, изображенный на рис. 10.15.

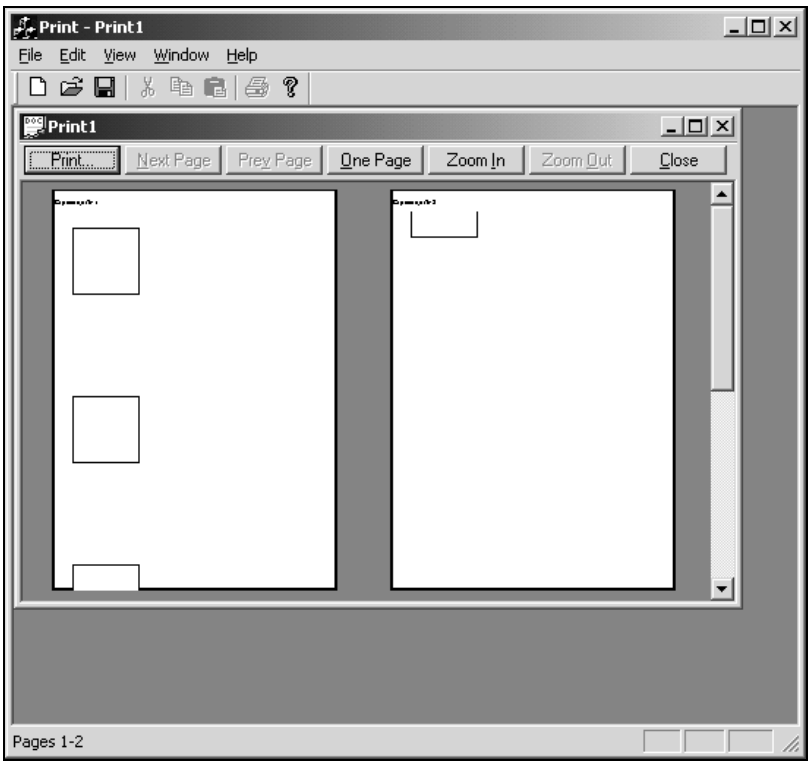


Рис. 10.15. Окно предварительного просмотра печати после редактирования

20. Включите принтер и нажмите кнопку **Print** (Печать) на панели инструментов окна предварительного просмотра документа. Появится диалоговое окно **Print** (Печать), изображенное на рис. 10.16.
21. Нажмите кнопку **OK**. Документ будет распечатан.
22. Закройте окно предварительного просмотра печати и закройте приложение.

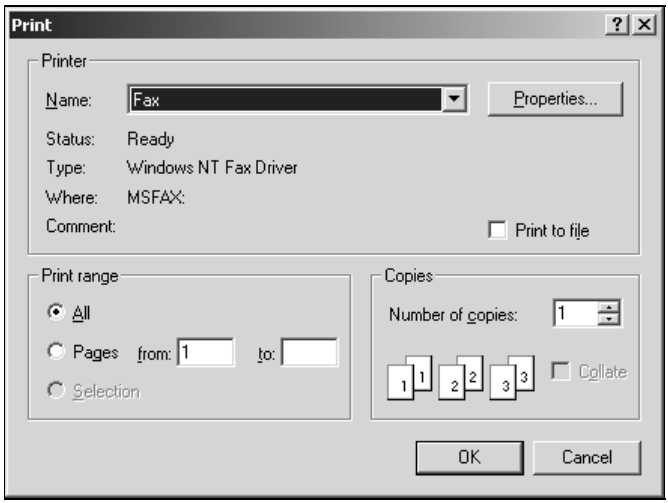


Рис. 10.16. Диалоговое окно **Print**

Внесение в документ изменений, связанных с печатью многостраничного документа, было разбито на три этапа. На первом этапе были внесены изменения в функцию `OnBeginPrinting`, заготовку которой создал мастер MFC Application Wizard при генерации приложения.

Следует отметить, что при создании заготовки данной функции ее аргументы были заремаркированы и заголовок функции выглядел следующим образом:

```
void CPrintView::OnBeginPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
```

Это было сделано потому, что в заготовке функции ее аргументы нигде не используются, что неизбежно приведет к появлению соответствующих сообщений при трансляции приложения, содержащего данную функцию. Указание в списке аргументов функции только типа аргумента без его идентификатора означает, что данный аргумент в функции не используется, что в данном случае соответствует действительности. При включении в тело функции операторов, использующих данные аргументы, комментарии с них снимаются.

Данная функция вызывается приложением при инициализации процесса печати или предварительного просмотра печати после вызова функции `OnPreparePrinting`. В данном приложении функция `OnBeginPrinting` используется для определения количества страниц в документе. Для этого вызывается функция `CPrintInfo::SetMaxPage`, позволяющая задать номер последней страницы в документе. Для вычисления числа страниц в документе используется функция `CDC::GetDeviceCaps`, позволяющая получить разнообразные сведения об устройствах отображения информации. Задание в качестве аргумента данной функции значения `LOGPIXELSY` позволяет получить число элементов изображения в логическом дюйме по вертикали дисплея.

Поскольку нами задан режим отображения MM_LOENGLISH, при котором один элемент изображения имеет размер 0,01 дюйма, то для определения вертикального размера выводимого изображения в элементах изображения необходимо умножить полученную величину на размер изображения в логических единицах, и разделить его на 100. Операция деления была произведена заранее. При задании в качестве аргумента значения VERTRES функция CDC::GetDeviceCaps возвращает размер дисплея по вертикали в элементах изображения. Поскольку в данном случае в качестве дисплея выступает печатаемая страница, то операция деления позволяет определить количество страниц, необходимое для печати всего документа. Поскольку результатом деления целых чисел является целая часть результата деления, к частному следует прибавить единицу, соответствующую неполной последней странице.

После этого приложение может правильно определить количество страниц в документе, но, как видно из рис. 10.13, оно на каждой странице начинает печать документа с начала. Для того чтобы на каждой странице печаталась своя информация, необходимо произвести настройку начала отсчета для каждой страницы. Эта операция производится в функции OnPrepareDC.

Поскольку данная функция используется как для вывода информации на экран, так и для печати документа, то, прежде всего, выясняется цель ее вызова. Для этого вызывается функция CDC::IsPrinting, позволяющая определить, является ли данный объект класса CDC объектом класса контекста устройства принтера. В случае отрицательного ответа вызывается метод базового класса. Этот метод сам вызывает функцию CDC::IsPrinting и проверяет возвращаемое ею значение.

При печати документа функция CScrollView::OnPrepareDC по непонятной причине устанавливает начало отсчета для каждой страницы в начало документа, хотя, казалось бы, документ, использующий полосы прокрутки, скорее всего, будет многостраничным, и, по крайней мере, неразумно сводить на нет все усилия пользователя по его полной распечатке. Однако, поскольку это так, при распечатке данного документа в функции CPrintAppView::OnPrepareDC производятся основные установки, выполняемые функцией CScrollView::OnPrepareDC, определяется начало отсчета для данной страницы документа и вызывается функция CView::OnPrepareDC.

Функция CDC::SetMapMode устанавливает режим отображения, задаваемый его аргументом. В качестве аргумента данной функции используется переменная CScrollView::m_nMapMode, в которой хранится значение текущего режима отображения.

Функция CDC::SetViewportOrg устанавливает начало отсчета рабочей области контекста устройства. Начало отсчета рабочей области определяет точку, заданную в системе координат устройства, в которую GDI помещает начало координат окна, под которым понимается точка, заданная в логической системе координат, определяемая функцией SetWindowOrg.

В качестве аргументов данной функции выступают координаты начала отсчета рабочей области, выраженные в координатах устройства. Горизонтальная координата начала отсчета рабочей области равна нулю, поскольку нет необходимо-

сти разбиения документа по ширине. Вертикальная координата начала отсчета каждой страницы совпадает с координатами нижней границы рабочей области предыдущей страницы. Для определения этой величины используется описанная выше функция `CDC::GetDeviceCaps` с аргументом `VERTRES`, возвращающая размер дисплея по вертикали в элементах изображения, и переменная `CPrintInfo::m_nCurPage`, содержащая имя текущей страницы. После этого вызывается функция `CView::OnPrepareDC`, являющаяся методом базового класса. Как видно из рис. 10.14, после внесения изменений в функцию `OnPrepareDC` пользовательского класса представления приложение корректно производит разбиение на страницы.

При выводе документа на экран его имя и текущее положение рабочей области в документе обычно отображаются в заголовке окна и в строке состояния. При печати документа бывает полезно вывести на печать номер текущей страницы и имя документа. Эту операцию проделывают даже те программы, которые полностью соответствуют требованиям WYSIWYG (что вы видите, то вы и получите).

Обычно для вывода подобной информации используются *колонтитулы*. Поскольку данная операция производится только при выводе документа на печать, то ее целесообразно производить в функции `OnPrint`. Для этого необходимо соответствующим образом изменить начало отсчета рабочей области и, возможно, масштаб по координатам окна. Вывод колонтитулов может привести также к увеличению числа страниц в документе.

Все произведенные изменения необходимо передать в контекст устройства перед вызовом функции `OnDraw`, чтобы исключить перекрытие областей вывода колонтитулов и самого документа. Может возникнуть необходимость внесения изменений и в саму функцию `OnDraw`.

Как утверждается в библиотеке MSDN, одним из путей учета областей, занимаемых колонтитулами, является использование переменной `m_rectDraw`, являющейся членом класса `CPrintInfo`. При инициализации процесса печати каждой страницы в данную переменную записывается область печати страницы, заданная в логических координатах. После печати колонтитулов предлагается соответствующим образом изменить размеры прямоугольника, хранящегося в переменной `m_rectDraw`. После этого функция `OnPrint` может обращаться к данной переменной для определения области, остающейся для печати документа.

Печать колонтитулов или какой-либо другой информации не может производиться в функции `OnPrepareDC`, поскольку она вызывается до вызова функции `CDC::StartPage`. До вызова этой функции контекст устройства принтера используется для всей страницы. Печать может осуществляться только в функции `OnPrint`.

Для печати верхнего колонтитула в функции `OnPrint` создается структура логического шрифта и на ее основе с помощью функции `CFont::CreateFontIndirect` инициализируется объект класса `CFont`. Процедура вывода текста подробно описана нами при рассмотрении приложения TextApp в *главе 6*. Поэтому здесь мы не будем на ней подробно останавливаться. После записи в объект класса `CString` содержимого верхнего колонтитула производится установка начала от-

счета для данной страницы. Для этого используется уже описанная нами функция `CDC::SetViewportOrg`. Начало координат страницы, выраженное в логических координатах, хранится в переменной `top` объекта класса `CRect`, хранящейся в переменной `CPrintInfo::m_rectDraw`. Поскольку для вывода колонтитула необходимо пространство, имеющее размер по вертикали, равный 50 логическим единицам, то эта величина прибавляется к текущему значению начала отсчета (имеющему отрицательное значение). После этого вызывается метод базового класса `CScrollView::OnPrint`, осуществляющий вывод текущей страницы документа на экран. Поскольку в данном случае производятся достаточно сложные преобразования области отсечки, лучше доверить эту операцию методу базового класса *(см. главу 6)*.

После этого необходимо удалить с экрана ту часть документа, которая перекрывается с областью колонтитула. Для этого в нее выводится белый прямоугольник, левая и правая границы которого совпадают с границами прямоугольника, хранящегося в переменной `CPrintInfo::m_rectDraw`, нижняя граница совпадает с началом отсчета рабочей области документа, а верхняя — находится за верхней границей листа. Использование в качестве верхней границы области колонтитулов старого значения начала отсчета рабочей области приведет к тому, что часть документа будет отображаться над этой областью. Это связано с тем, что область печати устанавливается меньше физического размера страницы для компенсации влияния различных допусков при настройке принтера и изготовлении бумаги. После очистки содержимого области колонтитула в него выводится текст самого колонтитула.

Уничтожение функции `OnPrepareDC` связано с тем, что устанавливаемое в ней начало отсчета страницы нигде не используется, поскольку эту задачу решает функция `OnPrint`.

Использование функций библиотеки MFC при печати

При печати многостраничного документа функции библиотеки MFC осуществляют постоянное взаимодействие между приложением и объектом пользовательского класса представления. Сначала приложение выводит диалоговое окно **Print** (Печать), создает контекст устройства для принтера и вызывает функцию `CDC::StartDoc`. Затем для каждой страницы документа приложение вызывает функцию `CDC::StartPage` и сообщает объекту класса представления о необходимости напечатать страницу, после этого вызывается функция `CDC::EndPage`. Если необходимо изменить настройки принтера перед началом печати данной страницы, то объект класса представления должен вызвать функцию `CDC::Escape` и послать принтеру соответствующие управляющие последовательности. После завершения процесса печати документа приложение вызывает функцию `CDC::EndDoc`.

Класс `CView` содержит различные функции, вызываемые приложением в процессе печати. Перегрузка этих функций в пользовательском классе представле-

ния позволяет настроить параметры процесса печати в соответствии с требованиями пользователя. В табл. 10.2 перечислены эти функции и приведены причины, по которым они могут быть перегружены.

Таблица 10.2. *Функции класса* `CView`*, используемые при печати*

Имя функции	Причина перегрузки
`OnPreparePrinting`	Установить нестандартные значения величин в текстовых полях диалогового окна **Print** (Печать)
`OnBeginPrinting`	Создать шрифты и другие ресурсы GDI, используемые при печати
`OnPrepareDC`	Настроить атрибуты контекста устройства для данной страницы или произвести разбиение на страницы в процессе печати
`OnPrint`	Печатать данную страницу
`OnEndPrinting`	Освободить ресурсы GDI, используемые при печати

Настройка параметров печати может производиться и в других функциях, но перечисленные выше функции являются непременными участниками процесса печати.

Процесс печати происходит следующим образом:

- вызывается функция `OnPreparePrinting` пользовательского класса представления, в которой устанавливается число страниц документа, если оно известно заранее, вызывается диалоговое окно **Print** (Печать), в котором, в случае необходимости, задаются нестандартные значения для содержимого его текстовых полей, и создается объект класса контекста устройства для принтера;
- вызывается функция `OnBeginPrinting` пользовательского класса представления, в которой определяется число страниц документа исходя из информации, содержащейся в контексте устройства, если эта величина не была определена заранее, и создаются объекты классов ресурсов GDI, используемых при печати;
- вызывается функция `CDC::StartDoc`, информирующая драйвер принтера о начале нового сеанса печати и о том, что все последующие вызовы функций `StartPage` и `EndPage` будут относиться к этому сеансу печати, пока не будет вызвана функция `EndDoc`;
- вызывается функция `OnPrepareDC` пользовательского класса представления, в которой устанавливается начало отсчета рабочей области и другие атрибуты контекста устройства. Если в данном приложении не определено количество страниц в документе, данная функция проверяет, не достигнут ли конец документа;
- вызывается функция `CDC::StartPage` для подготовки принтера к приему данных;

- вызывается функция `OnPrint` пользовательского класса представления, в которой производится печать колонтитулов и текущей страницы документа. Если вывод изображения на экран осуществляется функцией `OnDraw`, то она вызывается из функции `OnPrint`;
- вызывается функция `CDC::EndPage`, информирующая устройство о том, что приложение закончило передачу текущей страницы. После этого приложение вызывает функцию `OnPrepareDC` для новой страницы;
- после печати последней страницы вызывается функция `CDC::EndDoc`, завершающая текущий сеанс печати, инициированный вызовом функции `StartDoc`;
- вызывается функция `OnEndPrinting` пользовательского класса представления, в которой уничтожаются все объекты классов ресурсов GDI, созданных в функции `OnBeginPrinting`.

Приложение хранит большую часть информации, касающейся процесса печати в объекте класса `CPrintInfo`. Некоторые из членов данного класса связаны с процессом разбиения документа на страницы. Эти члены класса `CPrintInfo` перечислены в табл. 10.3.

Таблица 10.3. *Члены класса `CPrintInfo`, связанные с разбиением на страницы*

Член класса	Назначение
`GetMinPage/SetMinPage`	Позволяют получить и установить номер первой страницы документа
`GetMaxPage/SetMaxPage`	Позволяют получить и установить номер последней страницы документа
`GetFromPage`	Позволяют получить номер первой печатаемой страницы документа
`GetToPage`	Позволяют получить номер последней печатаемой страницы документа
`m_nCurPage`	Содержит номер текущей печатаемой страницы документа

Номера страниц начинаются с 1, т. е. первая страница имеет номер 1, а не 0.

При инициализации процесса печати приложение вызывает функцию `OnPreparePrinting` пользовательского класса представления, передавая ей указатель на объект класса `CPrintInfo`. Мастер MFC Application Wizard создает при генерации приложения заготовку функции `OnPreparePrinting`, в которой вызывается функция `CView::DoPreparePrinting`. Функция `DoPreparePrinting` выводит диалоговое окно **Print** (Печать) и создает контекст устройства принтера.

На этот момент приложение не знает, сколько страниц содержится в документе. По умолчанию номер первой страницы документа устанавливается в 1, а номер последней — в 0xFFFF. Если число страниц в документе известно заранее, то в перегруженной функции `OnPreparePrinting` следует вызвать функцию `CPrintInfo::SetMaxPage` прежде, чем вызывать функцию `DoPreparePrinting`.

Это позволит вывести в полях диалогового окна **Print** (Печать) истинные размеры документа.

Функция `DoPreparePrinting` выводит диалоговое окно **Print** (Печать). После его закрытия объект класса `CPrintInfo` содержит установки данного диалогового окна. Если пользователь хочет распечатать некоторый диапазон страниц, то ему необходимо указать границы этого диапазона в соответствующих текстовых полях диалогового окна **Print** (Печать). Приложение получает информацию о данных установках пользователя путем вызова функций `CPrintInfo::GetFromPage` и `CPrintInfo::GetToPage`. Если пользователь определил границы диапазона печати, то приложение вызывает функции `GetMinPage` и `GetMaxPage` и использует возвращенные ими значения для печати всего документа.

Для каждой печатаемой страницы документа приложение вызывает две функции пользовательского класса представления: `OnPrepareDC` и `OnPrint`. Каждая из этих функций имеет два аргумента: указатель на объект класса `CDC` и указатель на объект класса `CPrintInfo`. Каждый раз, когда приложение вызывает функции `OnPrepareDC` и `OnPrint`, оно передает им различные значения в переменной `CPrintInfo::m_nCurPage`, содержащей номер текущей печатаемой страницы. Таким образом, приложение сообщает объекту класса представления, какая страница должна быть напечатана.

Функция `OnPrepareDC` используется также для вывода документа на экран. Это позволяет вносить изменения в контекст устройства прежде, чем будет запущена процедура вывода изображения на экран. Функция `OnPrepareDC` играет сходную роль и в процессе печати, но в данном случае имеются два существенных различия: во-первых, объект класса `CDC` представляет собой контекст устройства принтера, а не контекст устройства дисплея, и во-вторых, в качестве второго аргумента передается указатель на объект класса `CPrintInfo`. (Этот аргумент имеет значение `NULL` при использовании функции `OnPrepareDC` для вывода на экран.) Перегрузка функции `OnPrepareDC` позволяет вносить изменения в контекст устройства в зависимости от того, какая страница печатается в данном случае. Например, в данной функции может быть произведена установка начала отсчета рабочей области и настроена область отсечки для вывода соответствующего фрагмента документа на печать.

Функция `OnPrint` используется для непосредственного вывода страницы на печать. Приложение вызывает функцию `OnPrint` и передает ей в качестве аргументов указатели на объекты классов `CDC` и `CPrintInfo`. Функция `OnPrint` передает контекст устройства функции `OnDraw`. Перегрузка функции `OnPrint` позволяет выводить информацию только в процессе печати, исключая ее вывод при отображении на экран. Например, в данной функции можно печатать колонтитулы. После вывода специфической для печати информации вызывается функция `OnDraw` для вывода информации, направляемой как на экран, так и на принтер.

Тот факт, что функция `OnDraw` осуществляет вывод информации как на экран, так и на принтер, гарантирует, что данное приложение соответствует принципам WYSIWYG (что вы видите, то вы и получите). Если перед вашим приложением не стоит данная задача, то вместо функции `OnDraw` можно вызвать любую пользовательскую функцию, используемую только для вывода на печать. Microsoft

советует использовать для этого саму функцию `OnPrint`, т. е. производить все операции по выводу информации на печать в этой функции, не вызывая в ней функций базовых классов.

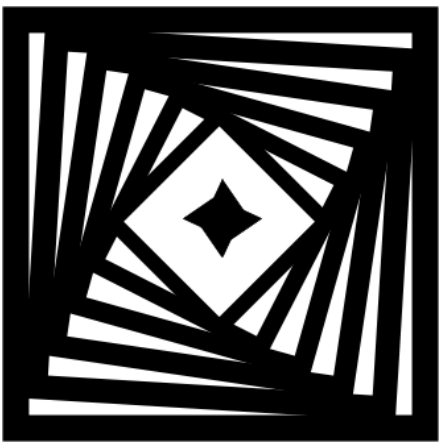
При нумерации страниц иногда следует помнить о различии между страницами принтера и страницами документа. С точки зрения принтера страница представляет собой лист бумаги. Однако один лист бумаги необязательно должен соответствовать одной странице документа. Например, при печати информационного бюллетеня, в котором листы должны складываться, один лист бумаги может содержать одновременно первые и последние страницы документа. А при печати электронных таблиц документ вообще не содержит страниц. Вместо этого один лист бумаги может содержать строки с 1 по 20 и столбцы с 6 по 10.

Все номера страниц в структуре `CPrintInfo` относятся к страницам принтера. Приложение вызывает функции `OnPrepareDC` и `OnPrint` один раз для каждого листа бумаги, заложенного в принтер. При перегрузке функции `OnPreparePrinting` для задания числа станиц в документе указываются страницы принтера. Если имеется взаимнооднозначное соответствие (то есть одна страница принтера равняется одной странице документа), то все вопросы снимаются просто. Если же взаимно однозначное соответствие отсутствует, необходимо установить между ними другой тип соответствия. Например, в случае печати электронной таблицы при перегрузке функции `OnPreparePrinting` необходимо вычислить, сколько листов бумаги потребуется для распечатки всей электронной таблицы, а затем использовать эту величину при вызове функции `CPrintInfo::SetMaxPage`. Точно так же при перегрузке функции `OnPrepareDC`, необходимо определить по текущему значению переменной `m_nCurPage` диапазон строк и столбцов, которые необходимо распечатать на данном листе.

В некоторых случаях пользовательский класс представления не может заранее определить размер документа. Например, эта ситуация возникает в том случае, если данное приложение не соответствует требованиям WYSIWYG и по размеру документа на экране нельзя определить его размер при печати.

Это вызывает проблемы при перегрузке функции `OnPreparePrinting` пользовательского класса просмотра, поскольку в нем не может быть вызвана функция `CPrintInfo::SetMaxPage` для задания размера документа. Если пользователь не определяет число страниц в документе, приложение должно само определить, когда ему прекратить процесс печати. Единственным способом для этого является распечатка документа до его конца и автоматическое определение момента достижения конца документа.

Приложение возлагает эту задачу на функцию `OnPrepareDC` пользовательского класса представления. После каждого вызова этой функции приложение проверяет значение переменной `CPrintInfo::m_bContinuePrinting`. Если оно равно `TRUE`, то процесс печати следует продолжить. Если оно равно `FALSE`, то процесс печати следует прекратить. По умолчанию функция `OnPrepareDC` присваивает переменной `m_bContinuePrinting` значение `FALSE`, если номер текущей страницы превышает 1. Это означает, что, если длина документа не была определена, приложение считает, что документ состоит из одной страницы.



ЧАСТЬ III

Особенности программирования в среде Visual C++

Глава 11. Исключения, шаблоны и новые возможности Visual C++

Глава 12. Многозадачность на основе потоков Windows

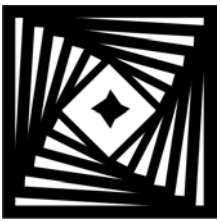
Глава 13. Справка в приложении

Глава 14. Отладка приложения

Глава 15. Создание пользовательской библиотеки

Глава 16. Создание простейшего приложения Internet

Глава 11



Исключения, шаблоны и новые возможности Visual C++

Специалисты Microsoft постоянно совершенствуют язык Visual C++. Новыми средствами, которые сравнительно недавно были добавлены в этот язык программирования, являются исключения, шаблоны и классы коллекций.

Работа с исключениями

Windows является операционной системой, управляемой сообщениями. Это означает, что приложение, по сути, разбивается на ряд практически независимых фрагментов, связь между которыми осуществляется операционной системой. В связи с этим остро встает вопрос об обработке ошибок, поскольку при возникновении ошибки в одном из этих фрагментов достаточно сложно передать эту ошибку в вызвавшую этот фрагмент процедуру. В случае работы с классами, не имеющими прототипов в библиотеке MFC, эта проблема становится практически неразрешимой без использования механизма исключений.

Механизм исключений позволяет процедурам приложения обмениваться информацией о возникновении серьезных проблем в процессе их выполнения. Для реализации данного механизма используются три оператора языка Visual C++:

- в блок `try` помещается программный код, в котором могут возникнуть проблемы при его исполнении;
- в блоке `catch`, располагающемся непосредственно после связанного с ним блока `try`, располагается программный код, обрабатывающий информацию о возникшей ошибке;
- оператор `throw` прерывает исполнение программы и передает управление связанному с ним блоку `catch`. Если с данным оператором не связан блок `catch`, тогда управление передается модулю библиотеки MFC, выводящему сообщение об ошибке и завершающему выполнение данного приложения.

Схематически структуру программы, использующей механизм исключений, можно представить следующим образом:

```
// Проверка исключения
try
```

```
{
  . . . . . .
  if(. . . .)
    throw;  // Вызов исключения
  . . . . . .
}

// Обработка исключения
catch(...)
{
  . . . . . .
}
```

В данном примере открывается блок `try` и в нем выполняются некоторые операции. В условном операторе `if` проверяется некоторое условие вызова исключения. Если это условие выполняется, то оператор `throw` вызывает исключение и передает управление блоку `catch`, в котором и производится обработка данного исключения. Если это условие не выполняется и программа доходит до конца блока `try`, то следующий за ним блок `catch` не выполняется.

Если в блоке `try` вызывается некоторая функция, содержащая оператор `throw`, не заключенный в блок `try`, то оператор `throw` действует в этой функции как оператор `return` и управление передается в связанный с данным блоком `try` блок `catch`. Если оператор `throw` в некоторой функции не заключен в блок `try` и вызов этой функции, также, не заключен в блок `try`, то исключение передается по всей иерархии вызывающих функций, пока не будет достигнута функция, вызов которой заключен в блок `try`. После чего выполняется описанная выше процедура. Если во всей иерархии функций ни одна из них не заключена в блок `try`, то приложение выдает сообщение об ошибке.

Аргументы исключений

При работе с исключениями необходимо постоянно помнить, что при вызове исключений ему передается один аргумент, в качестве которого может выступать любая структура данных. Синтаксис блока `catch` предполагает передачу ему аргумента, указанного в скобках после ключевого слова `catch`. При этом каждый блок обрабатывает только те исключения, которым при их вызове был передан аргумент того же типа. Это позволяет не только передать блоку `catch` всю информацию, необходимую для обработки данного исключения, но и создать для каждого типа исключения свой обработчик.

Если обработка исключения представляет собой достаточно сложную операцию, а само исключение вызывается достаточно часто, то процедура обработки может быть скрыта в классе исключения, как это показано в листинге 11.1.

Листинг 11.1. Класс исключения

```
// Заголовок класса исключения

class MyException
{
protected:
    UINT  nIDSError;
public:
  MyException(int nErr)  { nIDSError = nErr;};
  ~MyException()  {};

  void  Error_Put();
};

// Реализация класса исключения
void  MyException::Error_Put()
{

  CString    buf;
  VERIFY(buf.LoadString(nIDSError));

  ::MessageBox(0, buf, "Error", MB_OK | MB_SYSTEMMODAL);
}

// Использование класса исключения
int  SomeFunc()
{
  int*  aBuffer;

  try
  {
    if((aBuffer = new int[1024]) == NULL)
    {
      MyException*  me = new MyException(ID_MEMORY);
      throw me;
    }
    else
      delete  aBuffer;
  }
```

```
  catch(MyException* me)
  {
    me-> Error_Put();
  }
}
```

В листинге 11.1 приводится описание класса `MyException`, содержащего переменную `nIDSError`, в которую записывается идентификатор строкового ресурса, содержащего текст сообщения об ошибке, и функцию `Error_Put`, выводящую сообщение об ошибке на экран. Инициализация переменной `nIDSError` производится в конструкторе класса. В функции `Error_Put` с помощью вызова функции `CString::LoadString` по идентификатору ресурса получается связанное с ним сообщение. После чего вызывается функция `MessageBox` для вывода окна сообщения с соответствующим текстом. На этом обработка исключения завершается.

Библиотека MFC сама достаточно активно использует механизм исключений. Причем многие функции пользователя оказываются вызванными функциями библиотеки MFC, заключенными в блок `try`. Некоторые функции библиотеки MFC содержат оператор `throw`, но не заключены в блок `try`. Например, упомянутая выше функция `CString::LoadString` посылает исключение с аргументом, представляющим собой объект класса `CMemoryException`. При этом неизвестно, находится ли пользовательская функция в блоке `try` и какие типы исключений он обрабатывает. Эта информация получается методом проб и ошибок.

При обработке исключений необходимо помнить, что оператор `throw` действует аналогично оператору `return` и в нем не предусмотрены операции уничтожения динамических переменных, созданных в функции, из которой производится вызов исключения, или, что еще хуже, в функции, из которой производится вызов функции, вызвавшей исключение.

Поскольку пользователь зачастую и не подозревает, какая из используемых им функций может вызвать исключение, ему следует воздержаться от использования в своих функциях динамических переменных и создавать все свои переменные в стеке. Если ему все-таки придется воспользоваться динамическими переменными, имеет смысл оформить работу с ними отдельным классом, объект которого следует размещать в стеке. Это гарантирует, что при вызове исключения будет вызван деструктор данного класса, в котором будет произведено уничтожение динамического объекта.

Механизмы исключений Visual C++

В Visual C++ существует два механизма исключений:

- исключения, вызываемые как операторы языка C++. Этот тип исключений используется в библиотеке MFC версии 3.0 и более поздних ее версиях;
- *макросы исключений*, используемые во всех версиях библиотеки MFC.

При написании новых программ, использующих библиотеку MFC, при работе с исключениями следует применять операторы C++. Использование макросов оправдано только при добавлении дополнительных возможностей в старые приложения, активно задействующие макросы.

Можно произвести преобразование программ, использующих макросы исключений в программы, применяющие для этой цели операторы C++. Способ данного преобразования и его преимущества будут описаны позднее.

Для использования исключений в программах:

1. Откройте окно **Class View** (Просмотр классов) или **Solution Explorer** (Проводник решения), выделите в нем корневую папку приложения, в котором будут использоваться исключения, и выберите команду меню **View | Property Pages** (Вид | Вкладки свойств). Появится диалоговое окно **Print Property Pages** (Вкладки свойств), изображенное на рис. 11.1.

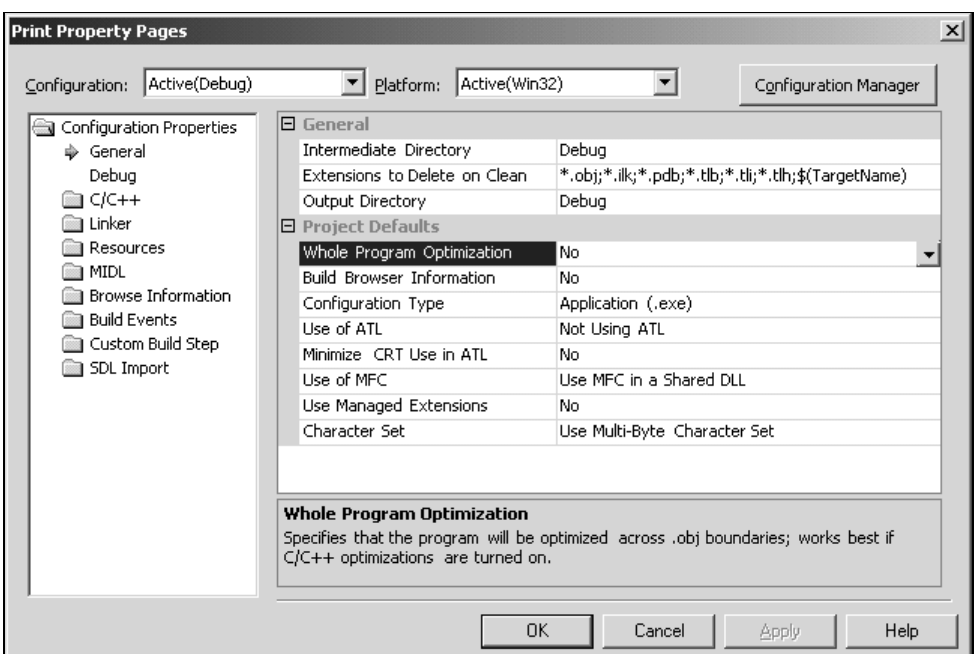


Рис. 11.1. Диалоговое окно **Print Property Pages**

2. Раскройте папку **C/C++**, выделите в ней строку **Code Generation** (Создание кода).
3. Если текстовое поле, соответствующее строке **Enable C++ Exceptions** (Разрешить исключения C++) в окне списка, содержит строку "Yes (/EHsc)", нажмите кнопку **OK**. Диалоговое окно, **Print Property Pages** (Вкладки свойств) закроется.

4. В противном случае щелкните левой кнопкой мыши на этой строке. В текстовом поле появится значок раскрывающегося списка.
5. Раскройте этот список и выделите в нем строку "Yes (/EHsc)". Диалоговое окно **Print Property Pages** (Вкладки свойств) примет вид, изображенный на рис. 11.2.

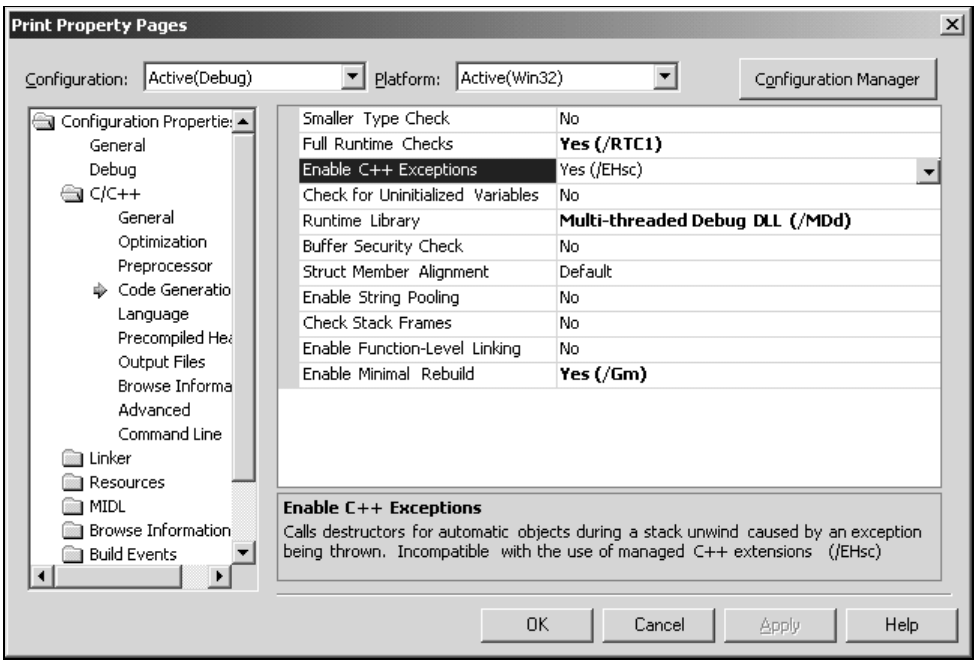


Рис. 11.2. Диалоговое окно **Print Project Settings**, вкладка **C/C++**

6. Нажмите кнопку **OK**. Диалоговое окно **Print Property Pages** (Вкладки свойств) закроется.

В каких случаях следует вызывать исключения

При вызове любой функции приложения возможны три исхода:

- **нормальное завершение:** функция может нормально завершить свою работу и возвратить управление вызвавшей ее функции. Некоторые функции возвращают значение, содержащее информацию о результатах выполнения данной функции. Эти возвращаемые значения строго определены для каждой функции и обозначают все возможные исходы ее выполнения. Возвращаемое значение может говорить об успешном завершении работы функции или о возникновении ошибки. В некоторых случаях возвращаемое значение может говорить о конкретном виде ошибки. Например, функции, позволяющие получать сообщение о состоянии объекта, могут возвращать значение, говорящее о том, что данный объект отсутствует;

- **завершение с ошибкой:** при вызове функции возникла ошибка при передаче функции аргументов или функция была вызвана в недопустимом контексте. Для обнаружения подобных ошибок в процессе отладки используются специальные макросы, одним из которых является макрос `ASSERT`;
- **возникновение особых ситуаций при выполнении программы:** особые ситуации возникают по причинам, не зависящим от данной программы. Например, вследствие ошибок ввода/вывода или недостаточного количества оперативной памяти. Именно для работы с такими особыми ситуациями и используются исключения. Применение исключений позволяет корректно завершить работу приложения практически из любой его точки.

При использовании для вызова исключений операторов C++ или макросов библиотеки MFC применяются объекты класса `CException` или производные от него классы. Библиотека MFC содержит несколько классов исключений, предназначенных для работы в определенных ситуациях:

- `CMemoryException` — недостаточно оперативной памяти;
- `CFileException` — исключения, связанные с работой с файлами;
- `CArchiveException` — исключения, связанные с работой с архивами и работой функции `Serialize`;
- `CNotSupportedException` — было вызвано устройство, не поддерживающееся в данной конфигурации операционной системы;
- `CResourceException` — исключения, связанные с работой с ресурсами;
- `CDaoException` — исключения, связанные с работой с базами данных (классами DAO);
- `CDBException` — исключения, связанные с работой с базами данных (классами ODBC);
- `COleException` — исключения, связанные с работой с объектами OLE;
- `COleDispatchException` — исключения, связанные с работой серверов автоматизации;
- `CUserException` — данное исключение выводит окно сообщения, после чего посылает сообщение `CException`.

Хотя в качестве аргумента блока `catch` может быть использована переменная практически любого типа, функции библиотеки MFC вызывают только исключения, являющиеся объектами классов, производных от класса `CException`. Чтобы перехватить исключение, вызванное функцией библиотеки MFC, следует написать блок `catch` и указать в качестве его аргумента указатель на объект класса `CException` (или на объект класса, производного от класса `CException`, например, `CMemoryException`). Объект каждого класса исключения содержит переменные, позволяющие установить причину возникновения исключения.

Например, объект класса `CFileException` включает в себя переменную `m_cause`, содержащую код причины вызова файлового исключения. В качестве кодов причины вызова исключения могут использоваться значения

CFileException::fileNotFound и CFileException::readOnly. Ниже приведен пример использования данного класса исключения.

```
try
{
  // Фрагмент программы, в котором может быть
  // вызвано исключение.
}

// Обработка исключения CFileException
catch(CFileException* theException)
{
  if(theException->m_cause == CFileException::fileNotFound)
    TRACE("File not found\n");
  theException->Delete();
}
```

Перехват и уничтожение исключений

Пользовательский блок обработки исключений должен уничтожать передаваемые ему объекты, поскольку в противном случае могут возникнуть "утечки" памяти. Блок catch должен уничтожать исключения в следующих случаях:

- когда блок catch вызывает новое исключение. Если исключение просто пересылается, его не следует уничтожать. Пересылка сообщения продемонстрирована в приведенном ниже примере.

  ```
  catch(CException* e)
  {
    if (m_bThrowExceptionAgain)
      throw; // Исключение не уничтожается
  }
  ```

- программа прекращает выполнение данной функции в блоке catch и передает управление вызвавшей ее функции.

При уничтожении объекта класса CException следует использовать функцию Delete данного объекта, а не оператор delete, поскольку исключение может не находиться в куче.

Чтобы перехватить и уничтожить исключение, следует использовать блок try. Исполнение любого фрагмента программы, который может вызвать исключение, необходимо производить в пределах блока try. Фрагмент программы, осуществляющий обработку исключения, следует поместить в блок catch. Этот фрагмент программы будет выполняться только в том случае, если в блоке try было вызвано исключение того же типа, который обрабатывается в данном бло-

ке catch. Ниже приведена структура блоков try и catch, в которых производится уничтожение объекта класса исключения.

```
// Нормальное выполнение программы
...
try
{
  // Фрагмент программы, в котором может быть вызвано исключение
}

// Обработка всех типов исключений
catch(CException* e)
{
  // Обработка исключения.
  // Аргумент "e" содержит информацию об исключении.
  e->Delete();
}
// Продолжение нормального выполнения программы
...
```

При вызове исключения управление передается первому блоку catch, который обрабатывает тот же тип исключения, что и у вызванного исключения. Последовательное расположение блоков catch, обрабатывающих различные типы исключений, позволяет произвести свою обработку каждого типа исключения. Если необходимо обработать все возможные типы исключений и нет необходимости использовать при обработке объект исключения, вместо объявления аргумента типа CException* в блоке catch можно поставить три точки (...). Такой блок полезно ставить после всех других блоков catch для перехвата не обработанных ими типов исключений. Ниже приведен пример использования нескольких блоков catch.

```
try
{
  // Фрагмент программы, в котором может быть
  // вызвано исключение.
}
catch(CFileException* e)
{
  // Обработка ошибок при работе с файлами.
}
catch(CMemoryException* e)
{
  // Обработка нехватки памяти.
}
```

```
catch(CException* e)
{
  // Обработка всех остальных системных исключений.
}
```

Уничтожение объектов в исключениях

Вызов исключения в приложении может вызвать нормальный ход выполнения программы. Поэтому необходимо постоянно следить за динамически созданными объектами и корректно уничтожать их при вызове исключения. Для этого следует придерживаться двух основных принципов:

- обрабатывать исключения локально, после чего уничтожать все объекты одним оператором;
- уничтожать все объекты в блоке `catch` перед вызовом исключения, которое будет обрабатываться за пределами данной функции.

Следование этим принципам позволит избежать проблем, возникших в данном примере:

```
void SomeFunc()  // Данная функция содержит ошибку
{
  CSomeClass* Local = new CSomeClass;

  // Фрагмент программы, в котором может быть
  // вызвано исключение.
  Local-> OtherFunc();

  // Уничтожение объекта перед выходом из функции.
  delete Local;
}
```

В приведенной выше функции переменная `Local` не уничтожается при вызове исключения в функции `SomeFunc`. Управление из этой функции передается непосредственно в обработчик исключения, расположенный за пределами данной функции, не давая функции завершить свою работу и освободить динамическую переменную. Указатель на данный объект выходит за пределы своей области действия и область памяти, занимаемая данным объектом, не будет освобождена до завершения данного приложения. Эта ситуация называется "утечкой" памяти и диагностируется программной оболочкой.

Ниже приведен текст той же самой функции, в которой решена проблема "утечки" памяти.

```
void SomeFunc()
{
  CSomeClass* Local = new CSomeClass;
```

```
  try
  {
    // Фрагмент программы, в котором может быть
    // вызвано исключение.
    Local-> OtherFunc();
  }
  catch(CException* e)
  {
    // Обработка вызванного исключения
    e-> Delete();
  }

  // Уничтожение объекта перед выходом из функции.
  delete Local;
}
```

В данном случае используются локальные блоки обработки исключений. Это позволяет произвести все действия, необходимые для обработки данного исключения в самой функции, а затем продолжить ее нормальную работу, в завершение которой будут удалены динамические переменные.

Если обработку исключения нельзя или нецелесообразно производить локально (например, это исключение посылается во многих точках приложения, а его обработка требует достаточно сложных операций), можно воспользоваться другим методом работы с динамическими объектами при вызове исключений. В этом случае блок `catch` просто транслирует объект класса исключения во внешний обработчик, но перед этим он уничтожает все динамические переменные, созданные в данной функции. Пример подобной функции приведен ниже.

```
void SomeFunc()
{
  CSomeClass* Locale = new CSomeClass;

  try
  {
    // Фрагмент программы, в котором может быть
    // вызвано исключение.
    Locale->SomeFunc();
  }
  catch(CException e)
  {
    // Уничтожение объекта перед вызовом исключения.
    delete Locale;
```

```
    // Вызов исключения для внешней обработки.
    throw;
  }

  // Уничтожение объекта перед выходом из функции
  // при нормальном завершении работы.
  delete Locale;
}
```

Механизм исключений автоматически уничтожает оконные объекты, поэтому для них не нужно вызывать деструкторы.

Вызов исключений из функций пользователя

Помимо использовавшегося выше непосредственного указания типа посылаемого исключения, MFC позволяет применять при вызове исключений специальные функции. Эти функции обеспечивают полную корректность вызова исключения соответствующего типа. В приведенном ниже примере функция `AfxThrowMemoryException` используется для вызова исключения при невозможности выделить требуемый объем оперативной памяти. Функция `SomeFunc` пытается разместить два блока памяти и вызывает исключение в случае неудачи:

```
void SomeFunc()
{
  char* szFirst = new char[ 128 ];
  if(szFirst == NULL)
    AfxThrowMemoryException();

  char* szSecond = new char[ 1024 ];
  if(szSecond == NULL)
  {
    delete szFirst;  // Уничтожение созданного объекта
    AfxThrowMemoryException();
  }

  // Некоторая обработка
  delete szFirst;
  delete szSecond;
}
```

В случае, если не удалось разместить первый блок памяти, можно просто вызвать исключение, если же не удалось разместить второй блок памяти, то необходимо сначала уничтожить первый блок памяти, и только потом можно вызы-

вать исключение. Если оба блока были размещены успешно, они освобождаются перед выходом из функции.

Если при обработке исключения не будут использоваться средства библиотеки MFC, пользователю нет необходимости применять в качестве аргументов исключения объекты соответствующих классов. В приведенном ниже примере приложение пытается проиграть звуковой файл, и вызывает исключение в случае неудачи.

```
#include "mmsystem.h"
#define WAVE_ERROR -5
void Play(CString Name)
{
  // Используемая функция Win32 API возвращает 0,
  // если файл не может быть проигран.
  // В исключение имеет тип целочисленной константы.
  if(!sndPlaySound(Name, SND_ASYNC))
    throw WAVE_ERROR;
}
```

Процедура, используемая по умолчанию в библиотеке MFC, обрабатывает только исключения, являющиеся указателями на объект класса `CException` (и производных от него классов). В данном примере механизм обработки исключений библиотеки MFC не используется.

Преобразование макросов исключений в операторы C++

В библиотеке MFC версии 3.0 и более поздних версиях макросы исключений были заменены операторами C++. В предыдущих версиях библиотеки MFC использовался макрос `CATCH`, определявший тип исключения. То есть тип исключения определялся обработчиком. В операторах C++ тип исключения определяется при его вызове. Это несоответствие может вызвать проблемы в тех редких случаях, когда тип указателя на объект исключения отличается от типа самого объекта исключения. Это продемонстрировано в приведенном ниже примере.

```
TRY
{
  THROW((CException*) new CCustomException());
}
CATCH(CCustomException, e)
{
  TRACE("Это исключение будет обработано в библиотеке MFC 2.x \n");
}
AND_CATCH(CException, e)
```

```
{
  TRACE(" Это исключение будет обработано в библиотеке MFC 3.0 \n");
}
END_CATCH
```

Этот фрагмент программы будет вести себя иначе в версии 3.0, поскольку управление всегда передается первому блоку `catch`, имеющему соответствующий тип аргумента. В результате работы макроса

```
THROW((CException*)new CCustomException());
```

будет вызвано исключение `CException*`, несмотря на то, что программист хотел вызвать исключение `CCustomException`. Макрос `CATCH` в библиотеке MFC версии 2.5 и более ранних версиях использует функцию `CObject::IsKindOf` для проверки типа в процессе выполнения программы. Поскольку выражение

```
e->IsKindOf(RUNTIME_CLASS(CException))
```

является истинным, первый блок `CATCH` перехватывает управление. В версии 3.0, использующей операторы C++ для обработки большинства исключений, управление перехватит второй блок `CATCH`, имеющий аргумент `CException`.

Такие фрагменты редко встречаются в программах. Они обычно появляются в тех случаях, когда объект исключения передается другой функции, принимающей исключение типа `CException*`, производящей некоторую обработку, и транслирующей данное исключение.

Для решения данной проблемы необходимо перенести оператор `throw` из функции в вызывающий фрагмент программы и послать исключение соответствующего типа, известного на момент посылки исключения.

Блок `catch` не может вызывать тот же самый указатель на объект класса исключения, который он получил. Следующий пример не содержал ошибок в ранних версиях, но его исполнение в версии 3.0 и более поздних версиях может привести к непредсказуемым последствиям.

```
TRY
{
  // Фрагмент программы, в котором может быть
  // вызвано исключение.
}
CATCH(CSomeException, e)
{
  THROW(e);  // Ошибка. Необходимо использовать THROW_LAST()
}
END_TRY
```

Использование макроса `THROW` в блоке обработки исключения приводит к уничтожению указателя `e`, таким образом внешний блок обработки исключения по-

лучит некорректный указатель. Вместо этого макроса необходимо использовать макрос `THROW_LAST`.

Хотя для работы со старыми программами нет необходимости преобразовывать все имеющиеся в них макросы исключений в операторы C++ (в этом нет необходимости хотя бы потому, что практически все функции библиотеки MFC используют макросы, а не операторы C++), но необходимо четко понимать, какие преимущества сулит данный переход. Основными преимуществами использования операторов C++ является то, что:

- программы, использующие операторы C++ для работы с исключениями, имеют меньший размер своих исполняемых файлов и файлов библиотек динамической компоновки;
- использование операторов C++ для работы с исключениями предоставляет дополнительные возможности, к которым относится обработка исключений практически любых типов (`int`, `float`, `char` и т. д.), в то время как макросы позволяют работать только с объектами класса `CException` и производных от него классов.

Основным различием между макросами и операторами C++, применяемыми для работы с исключениями, является то, что при использовании макросов перехваченное исключение автоматически удаляется, как только оно выходит за область своего действия. Операторы C++ не производят данной операции, поэтому их необходимо уничтожать явным образом при обработке исключения.

Кроме того, имеются некоторые различия в синтаксисе макросов и операторов C++. Эти различия можно разделить на три группы.

- Различия в аргументах макросов и объявлении исключений.

  Макрос `CATCH` имеет следующий синтаксис:

  ```
  CATCH(exception_class, exception_object_pointer_name)
  ```

  Между именем класса и указателем на объект стоит запятая.

  Заголовок блока `catch` имеет следующий синтаксис:

  ```
  catch(exception_type exception_name)
  ```

  В нем имя класса используется для определения типа имени исключения и между ними не стоит запятой.

- Различия в задании последовательности блоков обработки исключений.

  При использовании макросов для первого блока в последовательности используется макрос `CATCH` (со своими аргументами), для последующих блоков используется макрос `AND_CATCH`, а для завершения последовательности используется макрос `END_CATCH`.

  При использовании операторов C++ каждый блок обработки начинается с ключевого слова `catch` (с объявлением своего аргумента). Для макроса `END_CATCH` не существует аналога. Блок `catch` завершается закрывающей скобкой.

- Различия в способе трансляции исключения.

  Для трансляции исключения используется макрос `THROW_LAST`. Вызов оператора `throw` без аргумента имеет те же последствия.

Чтобы преобразовать программу, использующую макросы для работы с исключениями, в программу, применяющую для этого операторы C++:

1. Определите местоположение всех макросов библиотеки MFC, таких как `TRY`, `CATCH`, `AND_CATCH`, `END_CATCH`, `THROW` и `THROW_LAST`.
2. Произведите следующие замены:
   - макрос `TRY` замените ключевым словом `try`;
   - макрос `CATCH` замените ключевым словом `catch`;
   - макрос `AND_CATCH` замените ключевым словом `catch`;
   - уничтожьте макрос `END_CATCH`;
   - макрос `THROW` замените ключевым словом `throw`;
   - макрос `THROW_LAST` замените ключевым словом `throw`;
3. Внесите изменения в аргументы макросов, чтобы они приняли вид объявления объектов исключений. Например, замените выражение

   ```
   CATCH(CException, e)
   ```

   выражением

   ```
   catch(CException* e)
   ```

4. Внесите изменения в блоки обработки исключений, чтобы в них при необходимости уничтожались объекты исключений.

Ниже приведен фрагмент программы, использующей макросы.

```
TRY
{
  // Фрагмент программы, в котором может быть вызвано исключение
}
CATCH(CException, e)
{
  if (m_bPassExceptionsUp)
    THROW_LAST();
  if (m_bReturnFromThisFunction)
    return;
  // Нет необходимости уничтожать переменную e
}
END_CATCH
```

После проведения над ним описанных выше действий он будет выглядеть следующим образом:

```
try
{
  // Фрагмент программы, в котором может быть вызвано исключение
}
catch(CException* e)
{
  if (m_bPassExceptionsUp)
    throw;
  if (m_bThrowDifferentException)
  {
    e->Delete();
    throw new CMyOtherException;
  }
  if (m_bReturnFromThisFunction)
  {
    e->Delete();
    return;
  }
  e->Delete();
}
```

Совместное использование макросов и операторов C++

В одной и той же программе для работы с исключениями могут одновременно использоваться макросы и операторы C++. Однако, недопустимо одновременное использование макросов и операторов в одном и том же блоке, поскольку макросы автоматически уничтожают объект класса исключения при выходе из блока обработки, в то время как операторы C++ не производят этой операции. Поэтому смешение макросов и операторов может привести или к "утечке" памяти, если объект класса исключения не будет уничтожен, или к нарушению работы памяти при попытке повторного уничтожения данного объекта. В приведенном ниже примере дальнейшее использование указателя на объект класса исключения невозможно.

```
TRY
{
  TRY
  {
    // Фрагмент программы, в котором может быть
    // вызвано исключение.
```

```
  }
  CATCH(CException, e)  // "Внутренний" блок catch
  {
    throw;  // Недопустимая попытка вызвать исключение и передать
            // управление во внешний  блок catch, расположенный ниже
  }
  END_CATCH
}
CATCH(CException, e)  // "Внешний" блок catch
{
  // Указатель e нельзя использовать,
  // поскольку он был уничтожен во внутреннем блоке catch.
}
END_CATCH
```

Проблема возникает потому, что указатель `e` был уничтожен при выходе из "внутреннего" блока `CATCH`. Использование макроса `THROW_LAST` вместо оператора `throw` позволит "внешнему" блоку `CATCH` получить корректный указатель.

```
TRY
{
  TRY
  {
    // Фрагмент программы, в котором может быть
    // вызвано исключение.
  }
  CATCH(CException, e)  // "Внутренний" блок catch
  {
    THROW_LAST();  // Вызывается исключение и управление
                   // передается внешнему блоку catch.
  }
  END_CATCH
}
CATCH(CException, e)  // "Внешний" блок catch
{
  // Указатель e корректен, поскольку использован
  // макрос THROW_LAST().
}
END_CATCH
```

Запрещается трансляция исключения из блока `try`, расположенного в блоке макроса `CATCH`. Выполнение приведенного ниже фрагмента программы вызовет ошибку.

```
TRY
{
  // Фрагмент программы, в котором может быть
  // вызвано исключение.
}
CATCH(CException, e)
{
  try
  {
    throw;  // Ошибка. При выполнении этой операции
            // переменная e (транслируемое исключение) будет уничтожена
  }
  catch(CException exception)
  {
  }
}
END_CATCH
```

Шаблоны

При написании программ часто приходится создавать функции или классы для выполнения некоторых действий над объектами разных типов, причем, во многих случаях над объектами различных типов необходимо производить одни и те же операции. Например, необходимо разработать класс, выводящий на экран содержимое массива в виде графика. Без использования шаблонов пришлось бы создавать специальный класс для каждого возможного типа переменных, хранящихся в данном массиве, и практически полностью повторять тексты функций в каждом из классов. Использование шаблонов позволяет написать один класс, который сможет работать с массивами всех возможных типов. Эта идея была заложена в концепцию шаблонов, описанную в рабочих документах ISO WG21/ANSI X3J16.

Для задания шаблона используется ключевое слово `template`. Оператор `template`, формат которого приведен ниже, позволяет задавать абстрактные типы данных при объявлении и описании функций и классов.

```
template < [typelist] [, [ arglist ]] > declaration
```

Аргументом оператора `template` является список абстрактных типов (имеющих формат `class identifier` или `typename identifier`) или других объектов, ис-

пользуемых в теле шаблона. В поле `declaration` размещается объявление функции или класса.

Объявление класса с использованием шаблона ничем не отличается от объявления класса без использования шаблона за тем исключением, что в данном классе при объявлении типов его переменных, аргументов функций и возвращаемых значений может использоваться идентификатор абстрактного класса, объявленный в аргументе оператора `template`. Пример объявления класса с использованием шаблона приведен ниже.

```
// Заголовок класса, использующего шаблон
template <class Type, int n> class Demo
{
  public:
  Type  buffer[n];
  Type  Some_Func(int);
};

// Реализация класса, использующего шаблон
template <class Type, int n>
Type Demo<Type,i>::Some_Func(int i)
{
  return  buffer[i];
};
```

Чтобы создать объект данного класса в программе, необходимо использовать объявление переменной, подобное приведенному ниже.

```
Demo<int, 12>  ClassInst;
```

Понятие шаблона

Шаблоны представляют собой механизм создания функций и классов с использованием абстрактных типов переменных с возможностью применения вместо этих абстрактных типов любых допустимых типов переменных. Использование шаблона позволяет создать один класс, способный работать с переменными различных типов, вместо того, чтобы создавать множество классов, по одному на каждый возможный тип переменной.

Например, для создания функции, сохраняющей тип возвращаемой величины при определении максимальной из двух величин, можно написать ряд перегруженных функций, подобных приведенным ниже.

```
// Выбор максимального из двух целых чисел
int max(int a, int b)
```

```
{
  return (a > b) ? a : b;
}

// Выбор максимального из двух коротких целых чисел
short max(short a, short b)
{
  return (a > b) ? a : b;
}

// Выбор максимального из двух действительных чисел
float max(float a, float b)
{
  return (a > b) ? a : b;
}
// и т. д....
```

С использованием шаблона все эти функции могут быть заменены одной функцией, подобной приведенной ниже.

```
template <class Type> Type max(Type a, Type b)
{
  return (a > b) ? a : b;
}
```

Это позволяет уменьшить размер программы и облегчить программирование приложений при сохранении проверки корректности используемых типов данных.

Практически во всех случаях можно обойтись и без использования шаблонов, однако их применение имеет следующие преимущества:

- использование шаблонов облегчает процесс написания программ. Вместо того чтобы писать одинаковые функции и классы для различных типов, достаточно написать одну обобщенную функцию или один обобщенный класс;
- использование шаблонов повышает читабельность программы, поскольку обеспечивает возможность применения абстрактных типов данных;
- использование шаблонов обеспечивает проверку типов данных. Поскольку типы данных, с которыми работает данный шаблон, становятся известными при компиляции программы, компилятор может производить проверку на совместимость используемых в функции типов данных.

Во многих отношениях шаблоны аналогичны макросам условной трансляции, заменяющих шаблон переменной указанным типом. Однако существуют принципиальные различия между макросом, подобным приведенному ниже,

```
#define max(i, j) (((i) > (j)) ? (i) : (j))
```

и следующим шаблоном:

```
template<class Type>
Type max (Type i, Type j)
{
  return ((i > j) ? i : j)
}
```

При работе с макросом возникнут следующие проблемы:

- транслятор не может проверить совместимость типов аргументов макроса. Он выполняется без специальной проверки типов аргументов;
- переменные `i` и `j` оцениваются дважды. Если, например, любой из этих аргументов декрементируется, то он будет декрементирован дважды;
- поскольку макросы раскрываются препроцессором, то ошибки трансляции будут указаны для операторов раскрытого макроса, а не для самого макроса. При отладке макрос будет представляться в раскрытом виде, что затрудняет сопоставление исходного текста программы с ее отладочным текстом.

Многие функции, применяющие указатель на переменные типа `void`, могут быть записаны с использованием шаблонов, поскольку пустые указатели часто применяются для того, чтобы позволить функции работать с данными неизвестного типа. При использовании пустых указателей компилятор не может определить тип передаваемых аргументов и, следовательно, не может проверить совместимость типов аргументов при перегрузке операторов, в конструкторах и деструкторах классов.

Использование шаблонов позволяет создавать функции и классы, работающие с типизированными данными. При написании функций они имеют абстрактные типы, которые преобразуются в конкретные типы в процессе компиляции. При этом для каждого типа аргумента, применяемого при вызове данной функции или класса в файле, содержащем описание данной функции или класса, создается версия данной функции или класса, использующая аргументы данного типа.

Такой подход позволяет транслятору рассматривать функции и классы, включающие абстрактные типы, как прототипы функций или классов для создания их версий для конкретных типов данных, используемых в данном приложении. Применение шаблонов повышает читабельность функций, поскольку позволяет не создавать специальных версий данных функций для сложных типов, таких как объекты классов или структур.

Шаблоны функций

Шаблоны функций используются для замены набора функций, имеющих один и тот же текст, но работающих с аргументами, имеющими различный тип. Пример шаблона функции приведен ниже:

```
template <class Type>
void SwapIt(Type& x, Type& y)
```

```
{
  Type  c;
  c = a;
  a = b;
  b = c;
}
```

Эта функция позволяет своим аргументам обмениваться значениями. Шаблон может использоваться для создания функций, аргументами которых могут быть не только переменные типов `int` и `long`, но и объекты типов, созданных пользователем. Функция `SwapIt` может иметь в качестве своих аргументов, даже, объекты классов, если в них определены конструктор, не имеющий аргументов, и оператор присваивания.

Кроме того, шаблон функции не позволит производить обмен значениями между объектами, имеющими различный тип, поскольку компилятор имеет информацию о типе переменных `a` и `b` на момент трансляции.

Вызов функции, использующей шаблон, практически ничем не отличается от вызова нормальной функции. Например:

```
int  i, j;
char  k;
SwapIt(i, j);  // OK
SwapIt(i, k);  // Ошибка. Использованы различные типы аргументов
```

Возможно явное задание типа аргументов в шаблоне функции. Например:

```
template<class Type>
void SomeFunc(Type)
{
  ...
}
void OtherFunc(short k)
{
  SomeFunc<long>(k);  // Вызывает функцию SomeFunc(long)
}
```

При явном задании типа аргумента в случае необходимости производится соответствующее преобразование типов ее аргументов. В приведенном выше примере переменная `k`, имеющая тип `short`, будет преобразована к типу `long`.

В Visual C++ версии 5.0 и более поздних может использоваться новый синтаксис для явного задания типов в шаблонах функций. Например:

```
template<class Type> void Some_Func(Type v)
{
  ...
};
```

```
//Явное задание типа аргумента char в функции Some_Func:
template<>
void Some_Func<char>(char v)
{
  ...
}

// Другая форма явного задания типа аргумента double в функции: Some_Func:
template<>
void Some_Func(double v)
{
  ...
}
```

Может быть использован и старый синтаксис:

```
// Старая форма задания типа аргумента char в функции: Some_Func:
void Some_Func(char)
{
  ...
}
```

При первом вызове функции, использующей шаблон с новым типом аргумента, транслятор создает версию данной функции, в которой абстрактный тип заменяется данным конкретным типом. Эта функция вызывается всякий раз, когда осуществляется вызов данной функции с данным типом аргумента. Если в различных модулях создается несколько идентичных версий функции с одинаковыми параметрами, в исполняемом файле останется только одна версия.

Если тип аргумента функции не задан явно в шаблоне функции, как это было описано выше, преобразование абстрактного типа аргумента к другому типу невозможно. В приведенном ниже примере

```
template<class Type>
void SomeFunc(Type, int)
{
  ...
};
int  i;
char  c;
SomeFunc(i, c);
```

преобразование переменной `i` к типу `Type` может вызвать ошибку, однако допустимо преобразование переменной `c` к типу `int`.

Шаблоны классов

Шаблоны классов используются для замены набора классов, имеющих один и тот же набор переменных и функций, но в котором эти переменные, аргументы и возвращаемые значения функций имеют различный тип. Пример шаблона класса приведен ниже:

```
template <class Type, int n>
class Some_Class
{
  public:
  Some_Class(void);
  ~Some_Class(void);
  void  Some_Func(Type*);
  private:
  Type  array[n];
};
```

В этом примере шаблон класса имеет два параметра, первый из которых имеет тип `Type`, а второй является целочисленной переменной `n`. Параметру `Type` может быть передано значение любого типа, включая структуру или класс. Параметру `n` может быть передано значение любой целочисленной константы. Поскольку переменная `n` представляет собой константу, определяемую в процессе компиляции, пользователь может создать массив с размером `n`, используя стандартное объявление массива.

Объявление членов класса, использующего шаблон, несколько отличается от объявления членов класса, не использующих шаблон. Так, текст функции `Some_Func`, являющейся членом описанного выше класса, может иметь вид:

```
template <class Type, int n>
void  Some_Class< Type, n >::Some_Func(Type* a)
{
  for(int i=0; i < n; i++)
    array[i] = a[i];
}
```

Хотя в конструкторах и деструкторах имя шаблона класса упоминается дважды, параметры шаблона должны полностью указываться только один раз.

```
template <class Type, int n>
Some_Class< Type, n >::Some_Class(void)
{
  // Конструктор класса
```

```
}
template <class Type, int n>

Some_Class< Type, n >::~Some_Class(void)
{
  // Деструктор класса
}
```

В отличие от шаблонов функций, при создании конкретных версий классов пользователь должен в явном виде указать передаваемые аргументы шаблона класса. Для создания объекта класса `TempClass` могут быть использованы следующие операторы:

```
Some_Class< float, 6 >  First;        // OK
Some_Class< char, items++ >  Second;  // Ошибка, второй оператор
                                      // должен быть константой.
```

Для шаблона класса (или функции) не создается никакого исполнительного кода, пока транслятор не встретит объявления данного класса с конкретными параметрами. Более того, исполнительный код для функций — членов класса создается только в том случае, если эта функция вызывается в данной версии класса. Шаблон класса сначала определяется, а затем для него создается необходимая версия класса. Компилятор не создает версии класса для данного типа аргументов, пока этот класс не будет вызван с этим типом аргументов.

В Visual C++ версии 5.0 и более поздних версиях может использоваться новый синтаксис для явного задания типов в шаблонах классов. Например:

```
template<class Type>
class Some_Class
{
  ...
};
//Явное задание параметра int в шаблоне класса Some_Class
template<> class Some_Class<int>
{
  ...
};
```

Может быть использован и старый синтаксис:

```
//Задание типа char для класса Some_Class
class Some_Class<char>
{
  ...
};
```

При работе с шаблонами классов необходимо особое внимание обращать на положение угловых скобок (<>). Нужно таким образом разместить круглые скобки в выражениях, чтобы за угловые скобки не могли быть ошибочно приняты такие операторы, как >> и ->. Например, выражение

```
Some_Class< float, a > b ? a : b >  X;
```

должно быть заменено выражением

```
Some_Class< float, (a > b ? a : b) >  X;
```

Кроме того, необходимо обратить внимание на макросы, использующие угловые скобки в качестве аргументов шаблонов.

Как уже отмечалось, чтобы в программе появилась версия класса, использующая данный тип аргументов, необходимо, чтобы транслятор встретил объявление данного класса с данным типом аргумента. Поскольку трансляция приложения производится по файлам, необходимо, чтобы текст функций данного класса и его вызов располагались в одном файле. В противном случае данная версия класса не будет создана, а при вызове ее из другого файла компоновщик выдаст соответствующее сообщение об ошибке.

Для исключения подобных ситуаций создан механизм, позволяющий указать транслятору, какие типы аргументов будут использоваться в данном классе. Например, подобное объявление класса `Some_Class` в его файле реализации создаст версию класса, работающую с целочисленными переменными и способного хранить шесть переменных.

```
Template  class Some_Class<int, 6>;
```

Это выражение создает соответствующую версию класса, не создавая самого объекта данного класса. При этом программный код генерируется для всех функций класса.

Если необходимо создать программный код только для конструктора данного класса, используется следующее выражение.

```
Template  Some_Class <int, 6>::Some_Class (void);
```

Использование ключевого слова `extern` позволяет избежать автоматического создания всех функций — членов данного класса. Например:

```
extern  template  class Some_Class<int, 6>;
```

Аналогично, в качестве внешних могут быть объявлены отдельные функции данного класса:

```
extern  template  Some_Class<int, 6>:: Some_Class(void);
```

Ключевое слово `extern` может быть применено только к тем функциям, которые объявлены за пределами объявления класса. Функции, объявленные внутри объявления класса, считаются встроенными и для них всегда создается программный код.

Работа с классами коллекций

Библиотека MFC предлагает широкий выбор классов для работы с группами объектов. Эти классы могут быть условно разделены на две группы:

- классы, созданные с использованием шаблонов;
- классы, созданные без использования шаблонов.

Последняя группа классов представлена для обеспечения совместимости со старыми программами, поскольку она используется в библиотеке MFC, начиная с версии 1.0.

Виды классов коллекций

Классы коллекций различаются между собой по видам и по типу своих элементов. Вид класса определяет способ организации и хранения элементов в классе. В библиотеке MFC содержатся классы трех видов: списков, массивов и карты отображений (иногда называемые словарями). Это позволяет выбрать способ хранения информации, максимально соответствующий поставленной задаче.

Краткое описание каждого вида классов приведено ниже.

- Списки

  Списки представляют собой упорядоченный, неиндексированный набор элементов, организованных в связанный список. В списке возможен непосредственный доступ к его первому, последнему и текущему элементу. Для поиска всех остальных элементов необходимо произвести последовательный поиск, основываясь на хранящихся в каждом элементе списка указателях на предыдущий и последующий элементы. Эта структура позволяет сравнительно просто добавлять элементы в любую позицию списка.

- Массивы

  Массивы представляют собой упорядоченный набор объектов, каждому из которых соответствует целочисленный индекс. Массив может расширяться в процессе выполнения программы.

- Карты отображений (иногда называемые словарями)

  В данном наборе объектов для доступа к хранящемуся объекту используется ключевой объект.

Простейшим способом создания класса коллекции, хранящего объекты любого типа, проверяющие соответствие типа хранящихся объектов, является использование содержащихся в библиотеке MFC стандартных классов коллекций, использующих шаблоны. В табл. 11.1 приведен список этих классов коллекций.

Таблица 11.1. *Классы коллекций, использующие шаблоны*

	Массив	**Список**	**Карта отображений**
Коллекции объектов любого типа	`CArray`	`CList`	`CMap`

***Таблица 11.1** (окончание)*

	Массив	Список	Карта отображений
Коллекции указателей на объекты любого типа	CTypedPtrArray	CTypedPtrList	CtypedPtrMap

Если приложение включает классы коллекций, не использующие шаблоны, то нет необходимости в преобразовании их в новый стандарт, однако новые классы коллекций лучше создавать с применением шаблонов. В табл. 11.2 приведен список классов коллекций, не использующих шаблоны.

***Таблица 11.2.** Классы коллекций, не использующие шаблоны*

Массив	Список	Карта отображений
CObArray	CObList	CMapPtrToWord
CByteArray	CPtrList	CMapPtrToPtr
CDWordArray	CStringList	CMapStringToOb
CPtrArray		CMapStringToPtr
CStringArray		CMapStringToString
CWordArray		CMapWordToOb
CUIntArray		CMapWordToPtr

При выборе класса коллекции для своего приложения необходимо сопоставить характеристики различных видов классов и другие особенности их реализации. Эти характеристики включают в себя:

- особенности различных видов классов такие, как порядок расположения объектов, индексирование и производительность. Эти параметры приведены в табл. 11.3;
- использование шаблонов C++;
- возможность работы с архивами для сохранения объектов данных классов;
- возможность проведения диагностики хранимых объектов;
- проверка типов при работе с данным классом.

В таблице 11.3 приведены характеристики имеющихся видов классов коллекций:

- столбцы 2 и 3 описывают упорядоченность и возможность доступа по индексу к хранящемуся объекту. Под упорядоченностью понимается возможность введения понятия предыдущего и последующего объекта в наборе;
- столбцы 4 и 5 описывают скорость выполнения определенных операций с объектами данных классов. Важность каждого из этих параметров зависит от конкретного приложения.

Таблица 11.3. *Характеристики классов коллекций*

Вид класса	Упорядочен	Индексируется	Скорость вставки элемента	Скорость поиска элемента
Список	Да	Нет	Быстрая	Медленная
Массив	Да	Целыми числами	Медленная	Быстрая
Карта отображений	Да	По ключу	Быстрая	Быстрая

В табл. 11.4 приведены характеристики различных классов коллекций, содержащихся в библиотеке MFC.

Таблица 11.4. *Характеристики различных классов коллекций*

Класс	Использует шаблоны C++	Может работать с архивом	Имеет встроенную диагностику	Проверяет тип данных
`CArray`	Да	Да[1]	Да[1]	Нет
`CByteArray`	Нет	Да	Да	Да[3]
`CDWordArray`	Нет	Да	Да	Да[3]
`CList`	Да	Да[1]	Да[1]	Нет
`CMap`	Да	Да[1]	Да[1]	Нет
`CMapPtrToPtr`	Нет	Нет	Да	Нет
`CMapPtrToWord`	Нет	Нет	Да	Нет
`CMapStringToOb`	Нет	Да	Да	Нет
`CMapStringToPtr`	Нет	Нет	Да	Нет
`CMapStringToString`	Нет	Да	Да	Да[3]
`CMapWordToOb`	Нет	Да	Да	Нет
`CMapWordToPtr`	Нет	Нет	Да	Нет
`CObArray`	Нет	Да	Да	Нет
`CObList`	Нет	Да	Да	Нет
`CPtrArray`	Нет	Нет	Да	Нет
`CPtrList`	Нет	Нет	Да	Нет
`CStringArray`	Нет	Да	Да	Да[3]
`CStringList`	Нет	Да	Да	Да[3]
`CTypedPtrArray`	Да	Условно[2]	Да	Да

***Таблица 11.4** (окончание)*

Класс	Использует шаблоны C++	Может работать с архивом	Имеет встроенную диагностику	Проверяет тип данных
`CTypedPtrList`	Да	Условно[2]	Да	Да
`CTypedPtrMap`	Да	Условно[2]	Да	Да
`CUIntArray`	Нет	Нет	Да	Да[3]
`CWordArray`	Нет	Нет	Да	Да[3]

[1] Для использования архива необходимо непосредственно вызвать для объекта класса коллекции функцию `Serialize`. Для вывода диагностического сообщения необходимо непосредственно вызвать функцию `Dump`. Нельзя использовать выражения `ar << collObj` для работы с архивом или `dmp << collObj` для вывода диагностических сообщений.

[2] Возможность работы с архивом зависит от используемого вида класса коллекции. Например, если массив типизированных указателей основан на классе `CObArray`, то он может работать с архивом. Если он основан на классе `CPtrArray`, то он не может работать с архивом. Как правило, классы "`Ptr`" не могут работать с архивом.

[3] Значение "Да" в этом столбце означает, что класс коллекции, не использующий шаблона, проверяет только свой класс. Так, например, класс `CByteArray` проверяет, что он хранит байты, но при хранении в нем символов никаких дополнительных проверок не производится.

Классы коллекций, использующие шаблоны

Классы коллекций, использующие шаблоны, появились в библиотеке MFC версии 3.0 и могут применяться в последующих версиях данной библиотеки. Использование шаблонов для задания типа переменных, хранимых в классе коллекций, обеспечивает простой пользовательский интерфейс с объектами данных классов и более корректную проверку типов.

Простые классы коллекций имеют своим базовым классом класс `CObject`, поэтому они способны работать с архивом, могут создаваться динамически в процессе выполнения программы и обладают другими свойствами класса `CObject`. Классы коллекций указателей на объекты требуют от пользователя указать их базовый класс, в качестве которого может выступать любой класс коллекций указателей на объекты, не использующий шаблонов, из определенных в библиотеке MFC. В качестве базовых классов могут выступать такие классы, как `CPtrList` или `CPtrArray`. Новый класс коллекций наследует функции своего базового класса, однако при их вызове производится проверка на совместимость применяемых объектов классов.

Для использования простых классов коллекций необходимо знать тип данных, хранимых в данных классах, и параметры, которые необходимо указать при объявлении данных классов.

Простые классы массивов и списков CArray и CList имеют два параметра: TYPE и ARG_TYPE. В этих классах могут храниться любые данные, имеющие тип, задаваемый параметром TYPE:

- базовые типы данных C++, такие как int, char и float;
- структуры и классы C++;
- другие, определяемые пользователем типы.

Для простоты и повышения эффективности работы с данным классом пользователь может в параметре ARG_TYPE задать тип аргументов функций. Обычно в параметре ARG_TYPE задается ссылка на тип, указанный в параметре TYPE. Например:

```
CArray<short, short>  Some_Array;
CList<CSome_Class, CSome_Class&>  Some_List;
```

В первом примере объявлен целочисленный массив Some_Array, а во втором примере объявлен список Some_List, хранящий объекты класса CSome_Class. Некоторые функции — члены класса коллекции имеют аргументы, тип которых определяется значением параметра ARG_TYPE. Ниже приведен пример использования функции CArray::Add.

```
CArray<int, int>  Some_Array;
Some_Array.Add(5);
```

Простые классы карт отображений CMap имеют четыре параметра: KEY, ARG_KEY, VALUE и ARG_VALUE. Классы карт отображений позволяют хранить любые типы данных. Они представляют собой массивы, в которых доступ к элементу возможен по любому ключу, а не только по целочисленному. Параметр KEY определяет тип данных, используемых в качестве ключей для доступа к хранящимся в объекте класса данным. Если параметр KEY определяет структуру или класс, то в параметре ARG_KEY обычно дается ссылка на тип данных, заданный в параметре KEY. Параметр VALUE определяет тип данных, хранящихся в объекте данного класса. Если параметр VALUE определяет структуру или класс, то в параметре ARG_VALUE дается ссылка на тип данных, заданный в параметре VALUE. Например:

```
CMap< int, int, SOME_STRUCT, SOME_STRUCT& >  First_Map;
CMap< CString, LPCSTR, CMyClass, CMyClass >  Second_Map;
```

В первом примере в переменной First_Map хранятся объекты структуры SOME_STRUCT, доступ к которым осуществляется по целочисленным ключам. Во втором примере в переменной Second_Map хранятся объекты класса CMyClass, доступ к которым осуществляется по ключам, представляющим собой объекты класса CString, причем при запросе объекта, хранящегося в объекте данного класса, возвращается ссылка на этот объект. В качестве ключа в данном примере может выступать текстовый запрос, а в качестве хранимых данных — ответ на этот запрос.

Поскольку параметр KEY имеет значение CString, а параметр ARG_KEY имеет значение LPCSTR, то ключи хранятся в карте отображения как объект класса

CString, но в аргументах функций таких, как SetAt соответствующие аргументы имеют тип LPCSTR, например:

```
CMap< CString, LPCSTR, CMyClass, CMyClass& >  Second_Map;
CMyClass  person;
LPCSTR lpstrName = "Иван";
Second_Map.SetAt(lpstrName, person);
```

Для использования шаблонов в классах коллекций указателей на объекты, так же, как и в случае простых классов коллекций, необходимо знать тип данных, хранимых в данных классах, и параметры, которые необходимо указать при объявлении данных классов.

Классы массивов и списков указателей на объекты CTypedPtrArray и CTypedPtrList имеют два параметра: BASE_CLASS и TYPE. В этих классах могут храниться любые данные, имеющие тип, задаваемый параметром TYPE. Они являются производными от одного из классов коллекций, не использующих шаблоны. Базовый класс задается в параметре BASE_CLASS. Для массивов в качестве базового класса используются классы CObArray или CPtrArray, а для списков — CObList или CPtrList. В том случае, когда в качестве базового класса используется класс CObList, новый класс не только наследует члены базового класса, но также объявляет несколько дополнительных функций или операторов, позволяющих проверять корректность используемых типов переменных при доступе к функциям и переменным базового класса. При этом допустимо любое разрешенное преобразование типов. Например:

```
CTypedPtrArray<CObArray, CMyClass*>  myArray;
CTypedPtrList<CPtrList, MY_STRUCT*>  myList;
```

В первом примере массив myArray имеет в качестве базового класса CObArray. В массиве хранятся указатели на объекты класса CMyClass (причем, класс CMyClass является производным от класса CObject). При работе с данным массивом можно использовать любые функции — члены класса CObArray или использовать новые функции GetAt и ElementAt, проверяющие тип своих аргументов, или оператор [], также проверяющий тип.

Во втором примере объявляется список указателей myList, имеющий в качестве базового класса CPtrList. В списке хранятся указатели на объекты структуры MY_STRUCT. Класс, имеющий в качестве базового класса класс CPtrList, используется для хранения указателей на объекты, не имеющие своим базовым классом класс CObject. Класс CTypedPtrList содержит следующие функции, проверяющие тип своих аргументов: GetHead, GetTail, RemoveHead, RemoveTail, GetNext, GetPrev и GetAt.

Класс карты отображений указателей на объекты CTypedPtrMap имеет три параметра шаблона: BASE_CLASS, KEY и VALUE. Параметр BASE_CLASS определяет базовый класс данного класса: CMapPtrToWord, CMapPtrToPtr, CMapStringToPtr, CMapWordToPtr, CMapStringToOb и т. д. Параметр KEY аналогичен параметру KEY в классе CMap: он определяет тип переменной, используемой в качестве ключа

для поиска информации. Параметр `VALUE` аналогичен параметру `VALUE` в классе `CMap`: он определяет тип объекта, хранимого в карте отображений. Например:

```
CTypedPtrMap<CMapPtrToPtr, CString, MY_STRUCT*>    myPtrMap;
CTypedPtrMap<CMapStringToOb, CString, CMyObject*>  myObjectMap;
```

Первый пример представляет собой объект класса карты отображений, производного от класса `CMapPtrToPtr`, использующего в качестве ключей объекты класса `CString` и хранящего указатели на объекты структуры `MY_STRUCT`. Для получения указателя по ключу может быть использована функция `Lookup`, проверяющая тип своих аргументов. Оператор `[]` позволяет получить хранимый указатель и добавить его, если он не найден. При поиске в карте отображений может быть использована функция `GetNextAssoc`, проверяющая тип своих аргументов.

Второй пример представляет собой объект класса карты отображений, производного от класса `CMapStringToOb`, использующего в качестве ключей объекты класса `CString` и хранящего указатели на объекты класса `CMyObject`. При работе с данным классом могут быть использованы те же функции, что и в предыдущем примере.

Если указанный в параметре тип переменной является классом или структурой, а не указателем или ссылкой на данный тип, этот класс или структура должны иметь копирующий конструктор.

Классы `CArray`, `CList` и `CMap` используют семь глобальных функций, которые можно перегрузить в производных от них классах. Три из этих функций используются для создания, удаления и сохранения элементов коллекций.

Функции `ConstructElements` и `DestructElements` вызываются членами класса для добавления или удаления элемента в коллекцию. В табл. 11.5 показано, какие функции — члены класса вызывают данные глобальные функции.

Таблица 11.5. *Использование функций* `ConstructElements` *и* `DestructElements`

Глобальная функция	Вызывается непосредственно	Вызывается опосредованно

`ConstructElements`	`CArray::SetSize`
	`CArray::InsertAt`
		`CList::AddHead`
		`CList::AddTail`
		`CList::InsertBefore`
		`CList::InsertAfter`
		`CMap::operator [ ]`

`DestructElements`	`CArray::SetSize`
	`CArray::RemoveAt`
	`CList::RemoveAll`
	`CMap::RemoveAll`
		`CList::RemoveHead`
		`CList::RemoveTail`
		`CList::RemoveAt`
		`CMap::RemoveKey`

Данные функции перегружаются в том случае, когда их поведение по умолчанию не соответствует требованиям пользовательского класса. По умолчанию функция `ConstructElements` обнуляет всю область памяти, выделенную для данного элемента, и вызывает конструктор для каждого элемента. По умолчанию функция `DestructElements` вызывает деструктор для каждого элемента.

В большинстве случаев перегрузка функции `ConstructElements` бывает необходимой, если при сохранении элемента в классе надо вызвать конструктор (или другую функцию инициализации) для самого элемента или для одного из его членов. Перегрузка функции `DestructElements` осуществляется в том случае, когда при уничтожении объекта требуется произвести дополнительные действия, такие как освобождение памяти в куче, при его уничтожении. Пример перегрузки функции `ConstructElements` приведен ниже.

```
class CMyClass : public CObject
{
  ...
};

CArray< CMyClass, CMyClass& >  Some_Array;

template <>
void AFXAPI ConstructElements <CMyClass> (CMyClass* pNewElements, int
nCount)
{
  for (int i = 0; i < nCount; i++, pNewElements++)
  {
    // Непосредственный вызов конструктора класса CMyClass
    new(pNewElements)CMyClass;
  }
}
```

Перегруженная функция проходит по всем объектам класса `CMyClass` и для каждого из них вызывает конструктор. Вместо того чтобы создавать новый объект класса `CMyClass` в куче, для его создания используется специальный оператор `new`.

Классы `CArray`, `CList` и `CMap` вызывают функцию `SerializeElements` для записи хранящихся в них элементов на диск и чтения их с диска. По умолчанию функция `SerializeElements` производит побитовую запись объектов в архив и побитовое чтение их из архива.

Если класс коллекций является производным от класса `CObject` и при реализации класса используется макрос `IMPLEMENT_SERIAL`, то процесс программирования чтения и записи информации существенно упрощается:

```
class CMyClass : public CObject
{
  ...
};

CArray< CMyClass, CMyClass& >  Some_Array;
```

```
template <> void AFXAPI ConstructElements <CMyClass> (CArchive& ar,
    CMyClass* pNewElements, int nCount)
{
  for (int i = 0; i < nCount; i++, pNewElements++)
  {
    // Запись и чтение каждого объекта класса CMyClass
    pNewElements->Serialize(ar);
  }
}
```

Классы коллекций, не использующие шаблоны

Библиотека MFC содержит также классы коллекций, не использующие шаблоны, эти классы содержались еще в первой версии данной библиотеки. Они используются для хранения данных, имеющих типы `CObject*`, `UINT`, `DWORD` и `CString`. Кроме того, могут быть использованы классы коллекций объектов классов, производных от класса `CObject` (такие как `CObList`). Кроме того, библиотека MFC предоставляет классы коллекций для хранения таких простейших типов, как `UINT` и пустые указатели (`void*`).

Существует два пути использования классов коллекций, не применяющих шаблонов для надежного хранения данных:

- использовать в случае необходимости преобразование типов;
- создавать производные классы на их основе.

Чтобы применять преобразование типов при работе с классами коллекций, не использующими шаблонов.

- Применяйте данные классы, такие как `CWordArray`, непосредственно.

  Например, класс `CWordArray` позволяет хранить любые 32-разрядные значения. Они могут быть преобразованы к любому нужному типу.

  Класс `CObList` позволяет хранить указатели на объекты класса `CObject`. После извлечения указателя на объект из списка он может быть преобразован к любому совместимому типу указателя. Например, если в объекте класса `CObList` хранится указатель на объект класса `CSomeClass`, то полученный из списка указатель необходимо преобразовать к типу `CSomeClass*`. Пример данного преобразования приведен ниже:

  ```
  class CSomeClass : public CObject
  {
    ...
  };

  CSomeClass* lpObject = new CSomeClass(...);
  CObList myList;
  ```

```
myList.AddHead(lpObject);   // Явное преобразование типа не требуется
CSomeClass* lpOtherObject = (CSomeClass*)myList.GetHead();
```

Такое преобразование позволяет добиться удовлетворительного результата во многих практических случаях. Если необходимо добавить в класс дополнительные функции или обеспечить более жесткую проверку типов используемых переменных, применяйте шаблоны для задания типов или воспользуйтесь описанной ниже процедурой.

- Создайте собственный класс коллекций на основе одного из существующих классов, не использующих шаблоны.

При создании собственного класса коллекций пользователь может добавить в него функции проверки типов данных при доступе к функциям базового класса.

Например, при создании на базе класса `CObList` пользовательского класса, хранящего указатели на объект класса `CSomeClass`, в него могут быть добавлены функции `AddHeadPerson` и `GetHeadPerson`, как это показано ниже.

```
class CSomeClassList : public CObList
{
public:
  void AddHeadElement(CSomeClass* element) {AddHead(element);}

  const CSomeClass* GetHeadElement()  {return (CSomeClass*)GetHead();}
};
```

Эти функции обеспечивают проверку типа аргумента и преобразование возвращаемого значения к типу `CSomeClass`.

В данный класс могут быть добавлены и новые функции, расширяющие возможности базового класса.

Доступ к элементам классов коллекций

Классы массивов, определенные в библиотеке MFC, независимо от того, используют они шаблоны или нет, применяют целочисленные индексы для доступа к своим элементам. Классы списков и карт отображений используют индикатор типа `POSITION` для указания позиции внутри коллекции. Для доступа к одному или нескольким элементам данной коллекции необходимо сначала установить позицию индикатора, а затем несколько раз передать это значение объекту класса коллекции и запросить следующий элемент. Объект класса коллекций не отслеживает информацию о ходе поиска. Эта информация хранится в индикаторе позиции, а позиция хранит информацию о соседних с указанным элементах.

Для поиска информации в массиве используется функция `GetAt`. Пример ее использования приведен ниже.

```
CTypedPtrArray<CObArray, CSomeClass*>  Some_Array;
for(int i = 0; i < Some_Array.GetSize();i++)
{
  CSomeClass* lpElement = Some_Array.GetAt(i);
  ...
}
```

В этом примере определен массив указателей на объект класса `CSomeClass`. Этот массив является производным от класса `CObArray`, не использующего шаблоны. Функция `GetAt` возвращает указатель на объект класса `CSomeClass`. Класс `CTypedPtrArray` содержит также перегруженный оператор `[]`, позволяющий использовать привычный синтаксис работы с массивами. Задействуя этот оператор, выражение в цикле можно записать следующим образом:

```
CSomeClass* lpElement = Some_Array[ i ];
```

Этот оператор существует как в `const`, так и в обычной версиях. Оператор версии `const`, используемой в массивах констант, может применяться только в правой части оператора присваивания.

В поиске информации в списке участвуют функции `GetHeadPosition` и `GetNext`. Пример их использования приведен ниже:

```
CTypedPtrList<CObList, CSomeClass*>  Some_List;
POSITION  pos = Some_List.GetHeadPosition();
while(pos != NULL)
{
  CSomeClass* lpElement = Some_List.GetNext(pos);
  ...
}
```

В этом примере определен список указателей на объект класса `CSomeClass`. Объявление списка во многом аналогично объявлению массива, рассмотренному выше, но в качестве базового класса используется класс `CObList`. Функция `GetNext` возвращает указатель на объект класса `CSomeClass`.

Для поиска информации в карте отображений при неизвестном значении ключа функция `GetStartPosition` применяется для начальной установки индикатора, а функция `GetNextAssoc` — для последовательного доступа к элементам. Пример их использования приведен ниже:

```
CMap<CString, LPCTSTR, CSomeClass*, CSomeClass*>  Some_Map;
POSITION pos = Some_Map.GetStartPosition();
while(pos != NULL)
{
```

```
  CSomeClass* lpElement;
  CString string;
  // Получаем значение ключа (string) и элемента (lpElement)
  Some_Map.GetNextAssoc(pos, string, lpElement);
  // Используем полученные элемент и ключ
}
```

В этом примере используется простой шаблон класса карты отображений (вместо класса типизированных указателей на объекты), задействующий в качестве ключей объекты класса `CString` и хранящий объекты класса `CSomeClass`. При использовании функций доступа, таких как `GetNextAssoc`, объект класса возвращает указатели на объекты класса `CSomeClass`. Если применяемый при этом класс карты отображений не использует шаблона, то возвращаемый данной функцией указатель на объект класса `CObject` должен быть преобразован в указатель на объект класса `CSomeClass`. Пример использования подобного класса приведен ниже. Следует обратить внимание на то, что второй аргумент функции `GetNextAssoc` преобразуется к типу `CObject*&`.

```
CMapStringToOb Some_Map;  // Класс коллекции, не использующий  шаблонов

POSITION pos = Some_Map.GetStartPosition();
while(pos != NULL)
{
  CSomeClass* lpElement;
  CString string;
  // Получаем значение ключа (string) и элемента (lpElement)
  Some_Map.GetNextAssoc(pos, string, (CObject*&) lpElement);
  ASSERT(lpElement->IsKindOf(RUNTIME_CLASS(CSomeClass)));
  // Используем полученные элемент и ключ
  ...
}
```

Чтобы получить информацию по соответствующему ключу, используется функция `Lookup`, первый аргумент которой представляет собой ключ, по которому необходимо получить информацию, а второй — ссылку на соответствующий элемент в карте отображения. Для этой цели может применяться и перегруженный оператор `[ ]`.

Удаление элементов классов коллекций

Чтобы удалить все элементы в классе коллекции, хранящей объект класса `CObject` (или объекты классов, производных от класса `CObject`), можно получить доступ к каждому элементу данного класса и уничтожить его. Однако элементы в классе коллекций могут разделяться в том случае, когда коллекция хранит указатели на объекты. Если другие элементы коллекции или другие пе-

ременные программы хранят указатели на те же самые объекты, то их удаление без соответствующей проверки может привести к непредсказуемым результатам.

Чтобы уничтожить все элементы списка:

1. Используйте функции `GetHeadPosition` и `GetNext` для последовательного получения доступа ко всем элементам списка.
2. Используйте оператор `delete` для уничтожения объекта, связанного с каждым из элементов списка.
3. Вызовите функцию `RemoveAll` для удаления всех элементов списка после того, как будут уничтожены все связанные с ними объекты.

Ниже приведен пример уничтожения всех объектов класса `CSomeClass` из списка. Список содержит указатели на объекты класса `CSomeClass`, размещенные в куче.

```
CTypedPtrList<CObList, CSomeClass*>  Some_List;
POSITION pos = Some_List.GetHeadPosition();

while(pos != NULL)
{
  delete Some_List.GetNext(pos);
}
Some_List.RemoveAll();
```

Функция `RemoveAll` удаляет все элементы из списка. Функция `RemoveAt` удаляет отдельный элемент.

Обратите внимание на различие между удалением объекта, связанного с элементом списка, и удалением самого элемента. Удаление элемента из списка приводит к удалению только содержащегося в списке указателя на данный объект, сам же объект продолжает существовать. При уничтожении объекта он прекращает свое существование и освобождает занимаемую им память. Поэтому, после удаления объекта необходимо немедленно удалить все указатели на него для предотвращения их использования.

Чтобы удалить все элементы в массиве:

1. Используйте функцию `GetSize` и целочисленный индикатор для получения доступа ко всем элементам массива.
2. Используйте оператор `delete` для уничтожения объекта, связанного с каждым из элементов массива.
3. Вызовите функцию `RemoveAll` для удаления всех элементов массива после того, как будут уничтожены все связанные с ними объекты.

Ниже приведен пример уничтожения всех объектов класса `CSomeClass` в массиве. Массив содержит указатели на объекты класса `CSomeClass`, размещенные в куче.

```
CArray<CSomeClass*, CSomeClass*>  Some_Array;
for(int i = 0; i < Some_Array.GetSize(); i++)
  delete Some_Array.GetAt(i);
Some_Array.RemoveAll();
```

Так же как и при работе со списком, функция `RemoveAll` удаляет все элементы массива, а функция `RemoveAt` удаляет отдельный элемент.

Чтобы удалить все элементы в карте отображений:

1. Используйте функции `GetStartPosition` и `GetNextAssoc` для получения доступа ко всем элементам карты отображений.
2. Используйте оператор `delete` для уничтожения ключа и/или объекта, связанного с каждым из элементов карты отображений.
3. Вызовите функцию `RemoveAll` для удаления всех элементов карты отображений после того, как будут уничтожены все связанные с ними объекты.

Ниже приведен пример уничтожения всех элементов в объекте класса `CMap`. Каждый элемент карты отображений имеет строковый ключ и хранит объект класса `CSomeClass` (производного от класса `CObject`).

```
CMap<CString, LPCSTR, CSomeClass*, CSomeClass*>  Some_Map;
  ...
// Добавьте в карту отображений несколько элементов
  ...
// Теперь уничтожьте элементы
POSITION pos = Some_Map.GetStartPosition();
while(pos != NULL)
{
  CSomeClass*  lpElement;
  CString string;
  // Получение значения ключа (string) и объекта (lpElement)
  Some_Map.GetNextAssoc(pos, string, lpElement);
  delete lpElement;
}
// Функция RemoveAll уничтожает все ключи
Some_Map.RemoveAll();
```

Функция `RemoveAll` удаляет все элементы в карте отображения, а функция `RemoveKey` удаляет отдельный элемент по ключу.

Использование классов коллекций

Классы коллекций могут использоваться для создания различных полезных структур данных. Ниже будут приведены примеры использования класса `CTypedPtrList` для создания стеков и очередей.

Поскольку стандартный список имеет "голову", то этот класс может быть легко использован для создания стека. Стек иногда называют "магазинной" памятью по аналогии с магазином стрелкового оружия, где последний вставленный патрон используется первым. Для добавления элементов в стек может прриме-

няться функция `AddHead`, а для удаления элементов из стека — функция `RemoveHead`.

Чтобы создать класс стека, создайте новый класс, производный от существующего класса списка библиотеки MFC и добавьте в него дополнительные функции, отвечающие за функционирование стека. Ниже приведен пример такого класса.

```
class CBullet : public CObject
{
  ...
};
class CStack : public CTypedPtrList< CObList, CBullet* >
{
public:
  // Добавление элемента в вершину стека
  void Push(CBullet* newBullet)  { AddHead(newBullet); }

  // Чтение элемента из стека
  CBullet* Peek()  { return IsEmpty() ? NULL : GetHead(); }

  // Удаление элемента из стека
  CBullet* Pop()  { return RemoveHead(); }
};
```

В данном примере использованы функции — члены класса `CObList`, доступ к которым существует у пользователя независимо от того, имеют они отношение к стеку или нет.

Поскольку стандартный список имеет как "голову", так и "хвост", то этот класс может быть легко использован для создания очереди. Принцип работы очереди очевиден: первым пришел — первым обслужили. Новые объекты добавляются в конец очереди, поэтому для добавления элементов в очередь может использоваться функция `AddTail`, а для удаления элементов из очереди — функция `RemoveHead`.

Чтобы создать класс очереди, создайте новый класс, производный от существующего класса списка библиотеки MFC, и добавьте в него дополнительные функции, отвечающие за функционирование очереди. Ниже приведен пример такого класса.

```
class CSomeClass : public CObject
{
  ...
};
```

```
class CQueue : public CTypedPtrList< CObList, CSomeClass* >
{
public:
  // Переход в конец списка
  void AddToEnd(CSomeClass* newPerson)
    { AddTail(newPerson); }  // Конец очереди

  // Первый элемент в очереди
  CSomeClass* GetFromFront()
    { return IsEmpty() ? NULL : RemoveHead(); }
};
```

В данном случае добавление новых элементов производится в "хвост" списка, а чтение элементов производится из ее "головы" с одновременным удалением читаемого элемента.

Глава 12

Многозадачность на основе потоков Windows

Графическая оболочка Windows разрабатывалась как среда с *кооперативной многозадачностью* (cooperative multitasking) без разделения времени. Это означало, что если некоторая функция получила управление, то она не отдаст его до тех пор, пока не завершит свою работу, или ее работа не будет прервана извне. Если эта функция осуществляет достаточно сложные вычисления над большим объемом данных, подобная многозадачность может привести к практическому зависанию системы. Эта особенность сохранилась и после превращения графической оболочки в операционную систему.

Выходом из этой ситуации является использование потоков. Потоки (threads) обладают свойством *вытесняющей многозадачности* (preemptive multitasking). Это означает, что каждому потоку в соответствии с его приоритетом выделяется квант времени для его выполнения. По истечении данного кванта времени выполнение процесса прерывается и вызывается процедура распределения квантов времени, которая, при отсутствии других, более приоритетных претендентов, может опять выделить квант времени тому же потоку.

Приложение Win32 состоит из одного или более процессов. Под *процессом* (process) можно понимать некоторую исполняемую программу. В соответствии с другим определением под процессом понимают исполняемое приложение. Например, при двойном щелчке по значку **Paint** запускается процесс, в котором исполняется приложение стандартного графического редактора. В одном процессе могут быть запущены один или несколько потоков.

Поток представляет собой основную программную единицу, которой операционная система предоставляет процессорное время. В потоке может выполняться любой программный код, являющийся составной частью данного процесса, включая программный код, который в настоящее время выполняется другим процессом. Под потоком можно понимать ветвь исполнения процесса. При запуске приложения Paint операционная система создает процесс и запускает на исполнение основной поток данного процесса. При завершении работы данного потока завершается и процесс. Основной поток приложения запускается операционной системой и передается ей в качестве адреса функции, запускающей данный поток. Обычно это адрес функций `main` или `WinMain`.

Пользователь может создавать в приложении дополнительные потоки, если в этом возникает необходимость. Эта необходимость может возникнуть в том случае, когда в приложении можно выделить фоновые процессы, выполнение которых необходимо для правильной работы приложения, но пользователю нет

необходимости дожидаться их окончания для продолжения работы с приложением. Все потоки в приложениях, использующих библиотеку MFC, применяют объекты класса `CWinThread` или производных от него классов. В большинстве случаев пользователю нет необходимости самому создавать эти объекты. Ему достаточно вызвать функцию `AfxBeginThread`, которая сама создаст необходимый объект и запустит поток на исполнение.

Библиотека MFC позволяет работать с двумя видами потоков: *интерфейсными* и *рабочими*. Интерфейсные потоки обычно используются для обработки команд и сообщений пользователя. Рабочие потоки используются для выполнения задач, не требующих вмешательства пользователя. Например, для фоновых вычислений. Win32 API не различает эти виды потоков. Ему необходимо только знать начальный адрес потока, по которому необходимо произвести его запуск на исполнение. Интерфейсный поток содержит собственный *цикл обработки сообщений*, позволяющий ему обрабатывать сообщения и команды пользователя. Примером класса интерфейсного потока может служить класс `CWinApp`, производный от класса `CWinThread` и осуществляющий обработку событий и сообщений, поступающих в приложение.

Особое внимание следует уделить ситуации, когда несколько потоков имеют доступ к одному и тому же объекту. Для этого необходимо произвести их синхронизацию. Методы синхронизации будут описаны ниже в данной главе. При написании и отладке приложений, в которых используется несколько потоков, необходимо убедиться, что ни к одному из имеющихся объектов приложения не может быть осуществлен одновременный доступ из нескольких потоков.

Независимая работа потоков

В данном разделе будут рассмотрены создание различных типов потоков и работа с ними.

Создание рабочего потока

Одной из областей применения потоков является создание рабочих потоков. Они обычно используются для запуска фоновых задач, выполнение которых занимает достаточно много времени, и в процессе выполнения которых у пользователя может возникнуть потребность продолжить работу с данным приложением. К таким задачам относятся различные вычисления и фоновая печать документов.

Создание рабочего потока является довольно-таки простой задачей. Необходимо выполнить всего две операции: создать исполняющую функцию потока и запустить поток. При этом нет необходимости создавать класс, производный от класса `CWinThread`. В большинстве случаев этот класс может использоваться без перегрузки его функций, однако, иногда такая перегрузка необходима.

Для создания рабочего потока необходимо вызвать функцию `AfxBeginThread` и передать ей в качестве аргументов следующую информацию:

- адрес исполняющей функции потока;
- значение аргумента исполняющей функции потока.

Кроме того, данная функция имеет еще четыре необязательных аргумента.

- Приоритет потока.

  По умолчанию используется нормальный приоритет. Возможные значения приоритета перечислены при описании функции `::SetThreadPriority`.

- Предполагаемый размер стека, используемого потоком.

  По умолчанию у создаваемого потока создается стек того же размера, что и у вызывающего потока.

- Значение `CREATE_SUSPENDED`, если поток создается в приостановленном состоянии.

  По умолчанию в этом аргументе передается нулевое значение, означающее нормальное начало работы потока.

- Атрибуты безопасности потока.

  По умолчанию для потока выбираются те же атрибуты безопасности, что и для вызывающего потока.

Функция `AfxBeginThread` создает и инициализирует объект класса `CWinThread`, запускает его на исполнение и возвращает его адрес пользователю, чтобы он мог на него ссылаться. Если в процессе выполнения функции `AfxBeginThread` возникает ошибка, то в процессе выхода из функции производится проверка, что все объекты, созданные до этого в процессе выполнения данной функции, будут соответствующим образом уничтожены.

В *исполняющей функции потока* производятся все действия, которые необходимо выполнить в данном потоке. При вызове данной функции поток начинает свою работу, а при выходе из функции поток завершает свою работу. Эта функция имеет следующий формат:

```
UINT MyThreadFunction(LPVOID pParam);
```

Аргументом данной функции является 32-разрядная величина. Значение, передаваемое функции в этом аргументе, совпадает со значением второго аргумента функции `AfxBeginThread`, используемой для создания данного потока. Исполняющая функция может интерпретировать это значение любым способом. Оно может быть интерпретировано как скалярная величина, как указатель на объект структуры, содержащей несколько параметров, или может вообще игнорироваться. Если аргумент функции представляет собой указатель на объект структуры, то данный объект может использоваться не только для передачи данных в процедуру, но и для передачи обработанных данных приложению.

При завершении своей работы функция должна возвратить значение типа `UINT`, указывающее на то, почему данная функция завершила свою работу. Обычно

возврат нулевого значения говорит о нормальном завершении функции, а все остальные значения — о возможных ошибках. Эти значения определяются пользователем при написании исполняющей функции. Некоторые потоки могут подсчитывать число вызовов объектов определенных классов и возвращать число этих вызовов.

Ниже приведен пример исполняющей функции потока и ее вызова из приложения:

```
// Рабочая функция потока
UINT MyThreadProc(LPVOID pParam)
{
  CMyObject* lpObject = (CMyObject*)pParam;

  if (lpObject == NULL  |  |
    !lpObject->IsKindOf(RUNTIME_CLASS(CMyObject)))
      return 1;  // Проверка типа объекта

  // Работа с объектом
  return 0;  // Успешное завершение потока
}

// Вызов в одной из функций приложения
...
CMyObject*  lpMyObject = new CMyObject;
AfxBeginThread(MyThreadProc, lpMyObject);
...
```

Создание интерфейсных потоков

Поток для реализации пользовательского интерфейса обычно используется при организации ввода информации независимо от выполнения основного потока приложения. Основной поток приложения (производный от класса `CWinApp`) запускается вместе с приложением и завершается вместе с ним.

Для создания интерфейсного потока необходимо создать пользовательский класс, производный от класса `CWinThread`. При объявлении и при реализации данного класса необходимо использовать макросы `DECLARE_DYNACREATE` и `IMPLEMENT_DYNACREATE`. В этом классе надо перегрузить несколько функций. В табл. 12.1 приведены эти функции и действия, которые они должны совершать.

Таблица 12.1. *Перегружаемые функции — члены класса* `CWinThread`

Имя функции	Назначение
`ExitInstance`	Осуществляет уничтожение объектов после завершения потока. Обычно перегружается

Таблица 12.1 *(окончание)*

Имя функции	Назначение
`InitInstance`	Производит инициализацию экземпляра потока. Должна перегружаться
`OnIdle`	Производит обработку во время ожидания процесса. Обычно не перегружается
`PreTranslateMessage`	Производит фильтрацию сообщений перед их передачей функциям `TranslateMessage` и `DispatchMessage`. Обычно не перегружается
`ProcessWndProcException`	Перехватывает необработанные исключения, вызванные сообщениями и обработчиками команд потока. Обычно не перегружается
`Run`	Исполняющая функция потока. Содержит цикл обработки сообщений. Очень редко перегружается

Для создания интерфейсного потока необходимо вызвать функцию `AfxBeginThread` и передать ее в качестве аргумента имя класса, производного от класса `CWinThread`. Кроме того, данная функция имеет еще четыре необязательных аргумента.

- Приоритет потока.

  По умолчанию используется нормальный приоритет. Возможные значения приоритета перечислены при описании функции `::SetThreadPriority`.

- Предполагаемый размер стека, используемого потоком.

  По умолчанию у создаваемого потока создается стек того же размера, что и у вызывающего потока.

- Значение `CREATE_SUSPENDED`, если поток создается в приостановленном состоянии.

  По умолчанию в этом аргументе передается нулевое значение, означающее нормальное начало работы потока.

- Атрибуты безопасности потока.

  По умолчанию для потока выбираются те же атрибуты безопасности, что и для вызывающего потока.

Функция `AfxBeginThread` выполняет основную работу по созданию потока. Она создает новый объект пользовательского класса, инициализирует его, используя информацию, предоставленную пользователем, и вызывает функцию `CWinThread::CreateThread`. Если в процессе выполнения функции `AfxBeginThread` возникает ошибка, то в процессе выхода из функции производится проверка, что все объекты, созданные до этого в процессе выполнения данной функции, будут соответствующим образом уничтожены.

Прекращение работы потока

Поток может прекратить свою работу по двум причинам: потому, что его исполняющая функция завершила свою работу, или потому, что его работа была прервана извне. Если текстовый процессор использует поток для осуществления фоновой печати, то поток может завершить свою работу, распечатав весь имеющийся текст, или может быть прерван пользователем по той или иной причине.

Для рабочего потока оба случая ничем не отличаются друг от друга, поскольку любой выход из исполнительной функции означает прекращение работы потока. Различие состоит исключительно в том, что для прерывания потока он должен получить некоторую информацию извне, а для нормального завершения никакой дополнительной информации не требуется. Обычно о нормальном завершении потока говорит нулевое возвращаемое значение его исполняющей функции. Для прекращения работы рабочего потока могут быть использованы оператор `return` или функция `AfxEndThread`, вызываемая в исполняемой функции.

При нормальном завершении интерфейсного потока в той функции класса, где завершается его работа, после выполнения всех необходимых операций вызывается функция `::PostQuitMessage`, единственным аргументом которой является код завершения потока. Как и в случае рабочего потока, нулевое возвращаемое значение, обычно, означает нормальное завершение работы потока. Для прерывания работы интерфейсного потока необходимо вызвать функцию `AfxEndThread` в одной из функций класса потока. В качестве аргумента данной функции передается код завершения потока. Эта функция прекращает работу потока, освобождает стек потока и все библиотеки динамической компоновки, связанные с потоком, и уничтожает объект класса потока.

Функция `AfxEndThread` должна вызываться из завершаемого потока. Если необходимо завершить данный поток из другого потока, ему должно быть передано соответствующее сообщение, в результате обработки которого поток завершает свою работу.

Для получения кода завершения любого потока необходимо вызвать функцию `::GetExitCodeThread`. Первым аргументом этой функции является дескриптор потока (хранящийся в переменной `m_hThread` объекта класса `CWinThread`), а вторым — указатель на 32-разрядную величину, в которой хранится код завершения потока. Если указанный поток еще не завершил свою работу, во втором код завершения имеет значение `STILL_ACTIVE`.

Получение кода завершения потока требует выполнения дополнительных операций. По умолчанию при завершении потока соответствующий объект класса `CWinThread` уничтожается. Это означает, что доступ к переменной `m_hThread` объекта класса `CWinThread` становится невозможен, поскольку сам объект больше не существует. Чтобы избежать этой ситуации предлагаются два метода.

❒ Наиболее предпочтительный метод состоит в следующем:

- присвойте переменной `m_bAutoDelete` значение `FALSE`. Это позволит избежать автоматического уничтожения объекта класса `CWinThread` после завершения соответствующего потока;

- используйте значение переменной `m_hThread` после завершения работы потока;
- после использования данной переменной самостоятельно уничтожьте объект класса `CWinThread`.

❒ Альтернативный метод состоит в следующем:

- сохраните дескриптор окна в отдельной переменной;
- после создания потока с использованием функции `::DuplicateHandle` скопируйте переменную `m_hThread` в другую переменную и используйте ее для вызова функции `GetExitCodeThread`.

  Этот метод позволяет воспользоваться автоматическим уничтожением объекта класса потока и узнать причину его завершения. При использовании данного метода необходимо убедиться в том, что при копировании дескриптора процесс еще не завершил свою работу. Надежнее всего при этом использовать флаг `CREATE_SUSPENDED` при вызове функции `AfxBeginThread`, сохранить дескриптор, а затем возобновить его исполнение функцией `::ResumeThread`.

Использование любого из этих методов позволит определить причину завершения работы потока.

Взаимодействие между потоками

Одновременная работа нескольких потоков в одном приложении требует более внимательного отношения к проблеме обеспечения безопасности данных, чем при работе с одним потоком в приложении. Поскольку потоки представляют собой независимые процессы, которые могут обрабатывать одни и те же данные, необходимо убедиться в том, что при доступе к данным они всегда будут корректными. Некоторые операции с данными приводят к их временной некорректности. Например, при изменении содержимого записи в базе данных необходимо внести изменения во все ее поля, чтобы запись стала корректной.

Исходя из требований уменьшения размера и повышения быстродействия программ объекты библиотеки MFC располагают средствами для безопасной работы с потоками. Этими средствами располагают только классы. Это означает, что двум различным потокам не разрешается работать с одним и тем же объектом класса `CString`. Каждый из них должен работать с собственным объектом данного класса. Если же работа двух потоков с одним объектом необходима, следует воспользоваться механизмом синхронизации Win32, например, критическими секциями. Библиотека классов использует критические секции для защиты глобальных структур данных, используемых, например, при выделении памяти при отладке.

Потоки, созданные каким-либо другим образом, кроме наследования от класса `CWinThread`, не могут получить доступ к объектам библиотеки MFC, расположенным в другом потоке. Другими словами, чтобы получить доступ к объектам библиотеки MFC, расположенным в другом потоке, необходимо создать данный поток с использованием средств библиотеки MFC.

Взаимодействие между потоком и приложением

Как правило, поток имеет доступ только к тому объекту библиотеки MFC, в котором он был создан. Это происходит потому, что временные и постоянные карты дескрипторов Windows хранятся в локальной памяти потока для защиты от одновременного доступа нескольких потоков. Например, рабочий поток не может произвести вычисления и вызвать функцию `UpdateAllViews` в соответствующем документе, чтобы отобразить все внесенные в него изменения во всех связанных с ним окнах.

Эта операция не вызовет никаких изменений, поскольку карта преобразования объектов класса `CWnd` в объекты дескрипторов `HWND` является локальной для первичного потока. Это означает, что один поток может содержать преобразование дескриптора Windows в объект C++, но в другом потоке тот же самый дескриптор будет соответствовать другому объекту C++. Изменения, произведенные в одном потоке, не будут отображаться в другом.

Существует несколько путей решения этой проблемы. Первым из них является передача рабочему потоку индивидуальных дескрипторов (таких как `HWND`), а не объектов C++. После этого рабочий поток добавляет этот объект в свою временную карту вызовом функции `FromHandle`. Можно добавить этот объект и в постоянную карту потока, вызвав функцию `Attach`, однако это можно делать, если есть гарантия, что данный объект просуществует дольше, чем сам поток.

Другим способом решения данной проблемы является создание новых пользовательских сообщений, соответствующих определенным ситуациям, о которых рабочий поток должен извещать вызвавшее его приложение, и посылка их в главное окно с использованием функции `::PostMessage`. Этот метод аналогичен способу передачи сообщений между двумя приложениями за исключением того, что в данном случае оба потока используют одно и то же адресное пространство.

Использование классов синхронизации

При работе с несколькими потоками постоянно встает проблема синхронизации доступа потоков к общему ресурсу. Одновременный доступ двух и более потоков к одним и тем же данным может привести к нежелательным и непредсказуемым последствиям. Например, один поток может производить обновления содержимого объекта структуры, а другой считывать содержимое данного объекта. В этом случае трудно предположить, какие данные получит поток при чтении обновляемой структуры. Это могут быть новые данные, старые данные или смесь новых и старых данных.

Библиотека MFC содержит несколько классов синхронизации доступа к данным, позволяющим решить эту проблему.

В типичном приложении, использующем несколько потоков, для хранения разделяемого ими ресурса используется специальным образом организованный класс. Использование данного класса позволяет пользователю обойтись без использования специальных функций синхронизации при написании приложения.

Все необходимые функции включены и скрыты в классе, позволяя пользователю сосредоточиться на решении поставленной задачи, а не на обеспечении безопасности работы с данными.

Возьмем в качестве примера приложение, работающее с тремя окнами связного списка счетов. Это приложение позволяет вывести на экран три окна, в которые выводится один и тот же счет, но работа может вестись только с одним окном одновременно. После завершения работы со счетом он сохраняется в архиве.

Для реализации подобной системы следует использовать три класса синхронизации:

- поскольку приложение работает не более чем с тремя окнами счетов, используется класс `CSemaphore`, задающий максимальное число объектов в приложении. При попытке открыть четвертое окно счета приложение или будет ждать, пока не будет закрыто одно из открытых окон счетов, или аварийно завершит свою работу;
- для ограничения доступа к объекту при внесении в него изменений используется класс `CCriticalSection`, позволяющий в данный момент времени вносить изменения только в одном из окон счетов;
- после внесения изменений в счет используется объект класса `CEvent` для запуска остальных потоков, находившихся до этого в режиме ожидания. После этого данные посылаются в архив.

Чтобы создать класс, полностью скрывающий все функции синхронизации, необходимо, прежде всего, добавить в него соответствующие объекты классов синхронизации. В предыдущем примере в класс представления должен быть добавлен объект класса `CSemaphore`, в класс связного списка — объект класса `CCriticalSection`, а в класс документа — объект класса `CEvent`.

После этого необходимо вызвать соответствующие функции синхронизации в каждом из этих классов. Это означает, что все функции — члены классов, изменяющие данные, хранящиеся в классе или осуществляющие доступ к этим данным, должны создать объекты классов `CSingleLock` или `CMultiLock` и вызывать для этих объектов функцию `Lock`.

Если заблокированный объект выходит из области своего действия или уничтожается, то деструктор объекта вызывает функцию `Unlock` и освобождает ресурс. Пользователь может самостоятельно вызывать функцию `Unlock`, если в этом возникнет необходимость.

Создание подобных классов позволяет использовать их в приложениях, работающих с несколькими потоками, так же просто и надежно, как и в приложениях, имеющих единственный поток. Недостатком данного подхода является некоторое замедление работы с данными классами по сравнению с их версиями, в которые не добавлены объекты синхронизации. Кроме того, если имеется вероятность того, что один и тот же объект может быть уничтожен различными потоками, этот подход может привести к нежелательным результатам. В этом случае лучше использовать отдельные объекты синхронизации.

Шесть классов, используемых в библиотеке MFC для работы с несколькими потоками, могут быть разделены на две категории: классы синхронизации работы (`CSyncObject`, `CSemaphore`, `CMutex`, `CCriticalSection` и `CEvent`) и классы синхронизации доступа (`CMultiLock` и `CSingleLock`).

Классы синхронизации работы используются в том случае, когда доступ к ресурсу должен контролироваться для обеспечения целостности данного ресурса. *Классы синхронизации доступа* к объекту используются для получения доступа к контролируемому ресурсу.

Чтобы определить, какой из классов синхронизации следует применить в каждом конкретном случае, необходимо ответить на несколько вопросов.

- Должно ли приложение ждать возникновения некоторого события перед тем, как получить доступ к данному ресурсу? (Например, получения данных из коммуникационного порта перед тем, как записать их в файл.) Если да, то используйте класс `CEvent`.
- Могут ли несколько потоков одного приложения получить одновременный доступ к данному ресурсу? (Например, приложение позволяет открыть до пяти окон, связанных с одним документом.) Если да, то используйте класс `CSemaphore`.
- Может ли несколько приложений использовать данный ресурс? (Например, в том случае, когда ресурс расположен в библиотеке динамической компоновки.) Если да, то используйте класс `CMutex`. Если нет, используйте класс `CCriticalSection`.

Объекты класса `CSyncObject` нельзя использовать непосредственно. Он является базовым классом для четырех других классов синхронизации.

Решение вопроса о выборе класса синхронизации доступа, используемого в данном приложении, не представляет особой проблемы. Если приложение должно контролировать доступ только к одному разделяемому ресурсу, используется объект класса `CSingleLock`, если же необходимо контролировать доступ к нескольким разделяемым ресурсам, используется класс `CMultiLock`.

Простейший пример работы с потоками

В качестве примера работы с потоками рассмотрим демонстрационное приложение Sync, запускающее два рабочих потока и обеспечивающее их синхронизацию. Полный текст этого приложения можно найти в одноименной папке на дискете, прилагаемой к данной книге.

Чтобы создать демонстрационное приложение, работающее с потоками:

1. Создайте заготовку диалогового приложения по методике, описанной *в главе 1*, и назовите его Sync.

2. В окне **Resource View** (Просмотр ресурсов) выделите идентификатор ресурса `IDD_SYNC_DIALOG`, расположенный в папке **Dialog** (Диалог), и нажмите клавишу <F4>. Раскроется окно **Properties** (Свойства).
3. Щелкните левой кнопкой мыши на строке **Language** (Язык). В соответствующем текстовом поле появится значок раскрывающегося списка.
4. Раскройте этот список и выделите в нем строку **Russian** (Русский).
5. Дважды щелкните левой кнопкой мыши на идентификаторе ресурса `IDD_SYNC_DIALOG` и введите в текстовое поле **Caption** (Заголовок) окна **Properties** (Свойства) заголовок "Потоки".
6. Закройте окно **Output** (Окно вывода) и растяните заготовку диалогового окна.
7. Перетащите кнопки **Cancel** (Отмена) и **OK** в нижнюю часть заготовки диалогового окна.
8. Щелкните левой кнопкой мыши на статическом тексте в заготовке диалогового окна и введите в текстовое поле **Caption** (Заголовок) окна **Properties** (Свойства) заголовок "Результат".
9. Перетащите статический текст в верхнюю часть заготовки диалогового окна, и отцентрируйте его по горизонтали, предварительно приведя его рамку в соответствие с размерами текста в ней.
10. В заготовке диалогового окна выделите кнопку **Cancel** (Отмена) и введите в текстовое поле **Caption** (Заголовок) окна **Properties** (Свойства) заголовок "Отмена".
11. В окне **Toolbox** (Инструментарий) выберите элемент управления **Button** (Кнопка) и поместите его справа от кнопки **Отмена**.
12. Введите в текстовое поле **Caption** (Заголовок) окна **Properties** (Свойства) заголовок "Старт", а в тестовое поле раскрывающегося списка **ID** (Идентификатор ресурса) того же окна введите идентификатор ресурса `IDC_START`.
13. В окне **Properties** (Свойства) нажмите кнопку **ControlEvents** (События элементов управления). Раскроется список событий, генерируемых данным элементом управления.
14. В раскрывшемся списке выделите строку `BN_CLICKED`. В соответствующем текстовом поле появится значок раскрывающегося списка.
15. Раскройте этот список и выделите в нем единственную строку. В приложение добавится функция обработки данного сообщения.
16. В окне **Toolbox** (Инструментарий) выберите элемент управления **Check Box** (Флажок) и поместите его над кнопкой **Отмена**.
17. Введите в текстовое поле **Caption** (Заголовок) окна **Properties** (Свойства) заголовок "Синхронизировать", а в текстовое поле раскрывающегося списка **ID** (Идентификатор ресурса) того же окна введите идентификатор ресурса `IDC_SYNC`.
18. В окне **Toolbox** (Инструментарий) выберите элемент управления **Edit Control** (Текстовое поле) и поместите его между статическим текстом и флажком **Синхронизировать**.

19. Растяните текстовое поле на все свободное место и отцентрируйте его по горизонтали.
20. Введите в текстовое поле раскрывающегося списка **ID** (Идентификатор ресурса) окна **Properties** (Свойства) идентификатор ресурса `IDC_OUTPUT`.
21. Выделите строку **ReadOnly** (Только чтение). В соответствующем текстовом поле появится значок раскрывающегося списка.
22. Раскройте этот список и выделите в нем значение `TRUE`.
23. Повторите п.п. 24 и 25 для строк **Multiline** (Несколько строк), **VerticalScroll** (Вертикальная полоса прокрутки), **HorisontalScroll** (Горизонтальная полоса прокрутки), **ClientEdge** (Рамка) и **AutoVScroll** (Автоматическая прокрутка по вертикали). В результате произведенных действий заготовка диалогового окна примет вид, изображенный на рис. 12.1.

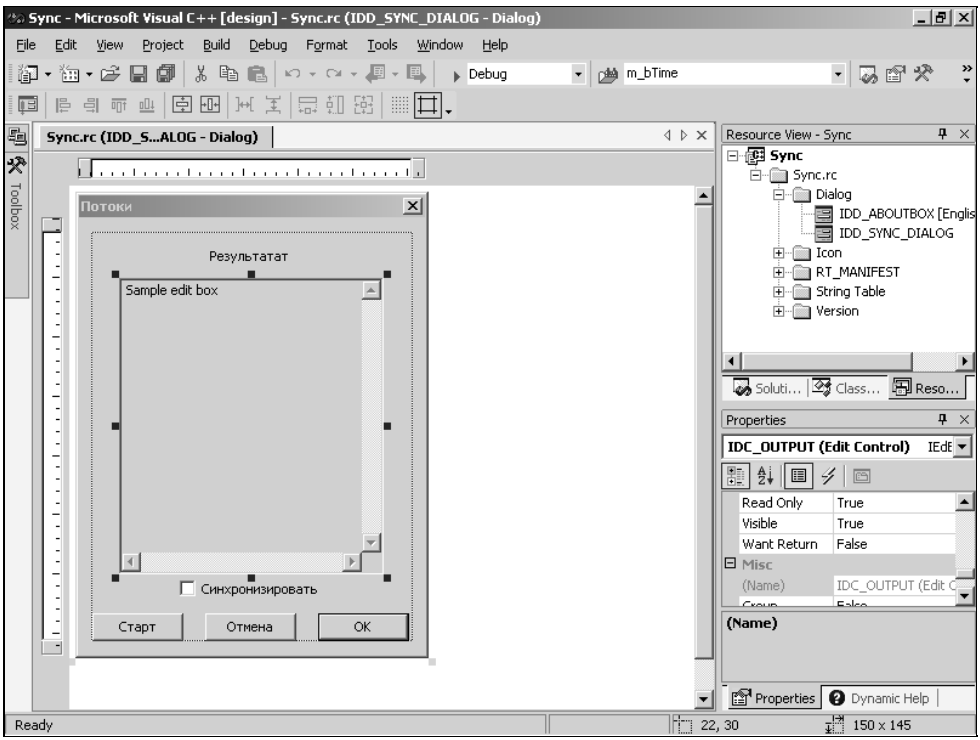


Рис. 12.1. Окончательный вид заготовки диалогового окна

24. Щелкните правой кнопкой мыши в окне редактирования ресурса и выберите в появившемся контекстном меню команду **Add Variable** (Добавить переменную). Появится диалоговое окно **Add Member Variable Wizard - Sync** (Мастер добавления переменной в класс), изображенное на рис. 12.2.

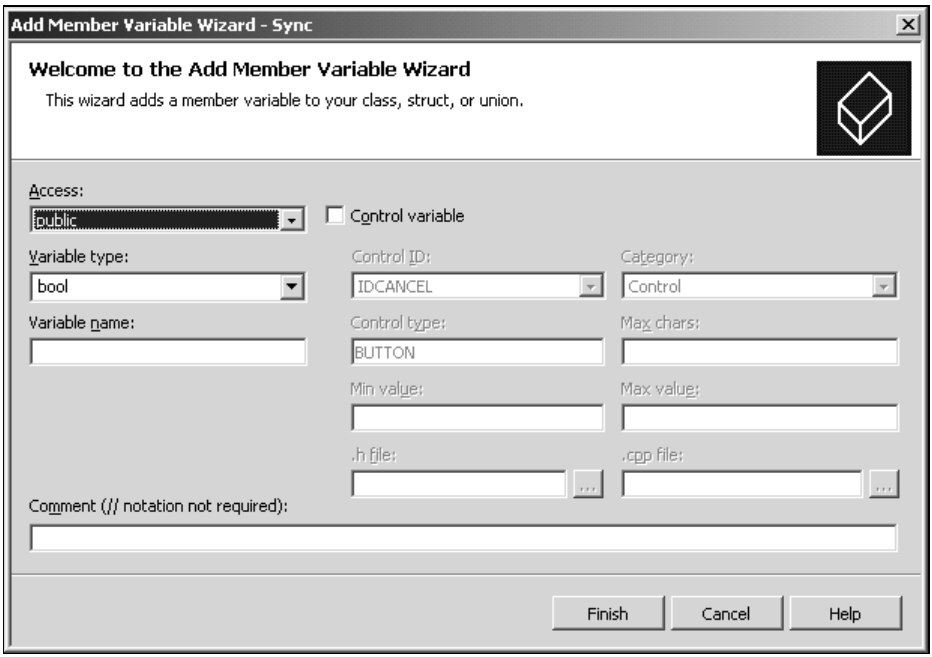


Рис. 12.2. Диалоговое окно **Add Member Variable Wizard - Sync**

25. Установите флажок **Control variable** (Связь с элементом управления) и выберите в раскрывающемся списке **Control ID** (Идентификатор элемента управления) идентификатор ресурса `IDC_SYNC`.
26. В раскрывающемся списке **Category** (Категория) выделите строку **Value** (Переменная), введите в текстовое поле **Variable name** (Имя переменной) идентификатор переменной `m_bSync` и нажмите кнопку **Finish** (Готово).
27. Повторите п.п. 24—26, сопоставив текстовому полю `IDC_OUTPUT` идентификатор переменной `m_Output`, но в данном случае оставьте содержимое текстового поля **Category** (Категория) без изменений.
28. Откройте окно **Class View** (Просмотр классов), в нем раскройте папку **Sync** и выделите в ней папку `CSyncDlg`.
29. В окне **Properties** (Свойства) нажмите кнопку **Overrides** (Перегружаемые функции базовых классов). Раскроется список виртуальных функций базовых классов класса `CSyncDlg`.
30. В этом списке выделите строку `OnCancel`. В соответствующем текстовом поле появится значок раскрывающегося списка.
31. Раскройте этот список и выделите в нем единственную строку. В класс `CSyncDlg` будет добавлена соответствующая функция обработки сообщения.
32. Повторите п.п. 29 и 30 для функции `OnOK`.

33. В окне **Class View** (Просмотр классов) щелкните правой кнопкой мыши на папке `CSyncDlg` и выберите в появившемся контекстном меню команду **Add | Add Variable** (Добавить | Добавить переменную). Появится диалоговое окно **Add Member Variable Wizard** (Мастер добавления переменной в класс).
34. В раскрывающемся списке **Variable type** (Тип переменной) выделите тип переменной `int`, введите в текстовое поле **Variable name** (Имя переменной) идентификатор переменной `m_Count` и нажмите кнопку **Finish** (Готово). Эта переменная будет добавлена в класс `CSyncDlg`.
35. Повторите п. 33 для добавления в класс `CSyncDlg` переменной `m_Critic`, имеющей тип `CCriticalSection`.
36. В окне **Class View** (Просмотр классов) дважды щелкните левой кнопкой мыши на папке `CSyncDlg`. Откроется окно редактирования файла SyncDlg.h.
37. Откройте окно редактирования файла SyncDlg.cpp и измените в нем функции обработки сообщений класса `CSyncDlg` в соответствии с текстом листинга 12.1.

Листинг 12.1. Функции обработки сообщений класса `CSyncDlg`

```
// Запуск потоков на исполнение
void CSyncDlg::OnBnClickedStart(void)
{
  // Считывание состояния диалогового окна
  UpdateData();

  // Запуск рабочих потоков
  AfxBeginThread(TextThreadProc, this);
  AfxBeginThread(TextThreadProc, this);
}

// Выход из диалогового окна
void CSyncDlg::OnCancel(void)
{
  // Проверка завершения работы потоков
  if(m_Count)
    return;

  // Вызов метода базового класса
  CDialog::OnCancel();
}

// Выход из диалогового окна
void CSyncDlg::OnOK(void)
```

```
{
  // Проверка завершения работы потоков
  if(m_Count)
    return;

  // Вызов метода базового класса
  CDialog::OnOK();
}
```

38. Перед реализацией класса `CAboutDlg` вставьте текст листинга 12.2.

Листинг 12.2. Рабочая функция потока

```
// Рабочая функция потока
UINT  TextThreadProc(LPVOID param)
{
  int      i;
  CString    szTemp;
  CSyncDlg*    pt = (CSyncDlg*) param;

  CSingleLock    FirstLock(&(pt->m_Critic));

  pt->m_Count++;  // Увеличение значения счетчика потоков

  for(i=0; i < 20; i++)
  {
    if(pt-> m_bSync)
      FirstLock.Lock();  // Блокировка доступа
    if(!pt-> m_bSync  |  |  FirstLock.IsLocked()) // Проверка блокировки
    {
      // Вывод строки
      szTemp.Format("Очень ");
      pt->m_Output.ReplaceSel(szTemp);
      Sleep(10);
      szTemp.Format("длинная ");
      pt->m_Output.ReplaceSel(szTemp);
      Sleep(10);
      szTemp.Format("строка ");
      pt->m_Output.ReplaceSel(szTemp);
      Sleep(10);
```

```
      szTemp.Format("номер ");
      pt->m_Output.ReplaceSel(szTemp);
      Sleep(10);
      szTemp.Format("%d \r\n", i);
      pt->m_Output.ReplaceSel(szTemp);
    }
    Sleep(100);  // Ожидание
    if(pt-> m_bSync)
      FirstLock.Unlock();  // Разблокирование доступа
  }
  pt->m_Count--;  // Уменьшение значения счетчика потоков
  return  0;
}
```

39. Нажмите клавишу <F5> и запустите приложение на исполнение. Появится диалоговое окно **Потоки**.
40. Нажмите кнопку **Старт**. Диалоговое окно **Потоки** примет вид, изображенный на рис. 12.3.

Рис. 12.3. Несинхронизированные потоки

41. Установите флажок **Синхронизировать**, и опять нажмите кнопку **Старт**. Диалоговое окно **Потоки** примет вид, изображенный на рис. 12.4.

Рис. 12.4. Синхронизированные потоки

42. Нажмите кнопку **Старт** и в процессе вывода информации на экран нажмите кнопку **Отмена** или **OK**. Диалоговое окно **Потоки** не будет закрыто.
43. Нажмите кнопку **Отмена** или **OK** по завершении вывода информации в окно. Диалоговое окно **Потоки** будет закрыто.

В данном демонстрационном приложении создаются два потока, выводящих информацию в одно и то же текстовое поле. Для того чтобы результат синхронизации был лучше виден, процесс вывода строки в рабочей функции потока разбит на несколько отдельных операций вывода. В этой функции создается объект класса `CSingleLock`, управляющий доступом к объекту класса синхронизации `CCriticalSection`. Однако этот объект используется только в том случае, если в диалоговом окне установлен флажок **Синхронизировать**. В этом случае перед выводом каждой строки текста вызывается функция `CSingleLock::Lock`, ожидающая отметки объекта класса `CCriticalSection` для передачи управления потоку. После того как объект класса `CCriticalSection` будет сброшен (заблокирован), производится проверка того, что он действительно был заблокирован. Для этого вызывается функция `CSingleLock::IsLocked` (данное приложение хорошо работает и без этой проверки, но так полагается). После вывода строки вызывается функция `CSingleLock::Unlock`, отмечающая объект класса синхронизации (снимающая блокировку).

В данном приложении потоки достаточно быстро завершают свою работу. Однако в реальном приложении работа потока может занять продолжительное время. При этом было бы нежелательно, чтобы вызвавший его поток был бы в это время закрыт. Поэтому в данном приложении осуществлена защита от

преждевременного закрытия диалогового окна. Для этого в класс диалогового окна введен счетчик числа открытых потоков. Для того чтобы замедлить процесс вывода текста, облегчив тем самым возможность закрытия окна в процессе его вывода, перед выводом каждой порции текста и перед вызовом функции `CSingleLock::Unlock` вызывается функция `Sleep`, осуществляющая задержку работы потока, хотя без последнего вызова данной функции эффект синхронизации более показателен.

Функции обработки сообщений диалогового окна предельно просты. В функции `OnBnClickedStart`, вызываемой при нажатии пользователем кнопки **Старт**, вызывается функция `UpdateData`, считывающая информацию о состоянии элементов управления в соответствующие переменные, и дважды вызывается функция `AfxBeginThread` для запуска потока вывода текста. В функциях `OnCancel` и `OnOK` производится проверка значения счетчика запущенных потоков и соответствующие методы базовых классов вызываются только в том случае, если этот счетчик имеет нулевое значение. В противном случае нажатие на эти кнопки не приводит ни к каким действиям.

Более сложный пример работы с потоками

В рассмотренном выше приложении имеется один существенный недостаток: нажатие кнопок **Отмена** и **OK** в процессе вывода текста в текстовое поле не приводит ни к каким последствиям. С одной стороны, это так и задумывалось, чтобы обеспечить нормальную работу вызванных потоков, но, с другой стороны, нажав эти кнопки, пользователь уже однозначно определил свое намерение. Если необходимо подождать завершения работы потоков, то никто и не возражает, но после их завершения необходимо закрыть диалоговое окно. (Можно было бы, конечно, прервать работу потоков, но это не так интересно, и при этом все равно, в большинстве случаев, придется дожидаться завершения работы прерываемого потока).

Чтобы внести соответствующие изменения в приложение Sync:

1. Выберите команду меню **File | Open Solution** (Файл | Открыть решение). Появится диалоговое окно **Open Solution** (Открыть решение), изображенное на рис. 12.5.
2. В окне списка раскройте папку **Sync** и дважды щелкните левой кнопкой мыши на значке **Sync**. В среду программирования будет загружено соответствующее приложение.
3. Откройте окно **Solution Explorer** (Проводник решения), раскройте в нем папку **Sync**, а в ней — папку **Source Files** (Файлы реализации).
4. Дважды щелкните левой кнопкой мыши на имени файла Sync.cpp. Откроется окно редактирования этого файла.
5. Измените функцию `InitInstance` в соответствии с текстом листинга 12.3.

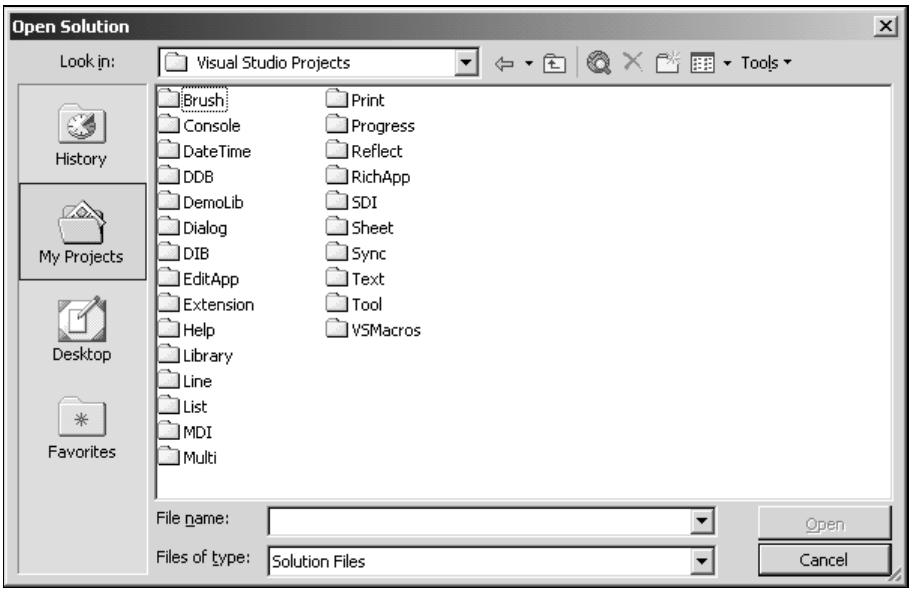


Рис. 12.5. Диалоговое окно **Open Solution**

Листинг 12.3. Функция `InitInstance`

```
// Инициализация приложения CSyncApp
BOOL CSyncApp::InitInstance()
{
  // Вызов метода базового класса
  CWinApp::InitInstance();

  AfxEnableControlContainer();

  // Создание класса диалогового окна
  CSyncDlg dlg;
  m_pMainWnd = &dlg;

  // Вызов потока синхронизации
  AfxBeginThread(SyncroThreadProc, &dlg);

  int nResponse = dlg.DoModal();
  if (nResponse == IDOK)
  {
```

```
    // Диалоговое окно было закрыто кнопкой OK
  }
  else if (nResponse == IDCANCEL)
  {
    // Диалоговое окно было закрыто кнопкой Отмена
  }

  // Поскольку диалоговое окно закрыто, функция возвращает
  // значение  FALSE, что приводит к завершению работы приложения
  // вместо запуска цикла обработки сообщений
  return FALSE;
}
```

6. Перед картой сообщений класса `CSyncApp` вставьте текст листинга 12.4.

Листинг 12.4. Рабочая функция потока синхронизации

```
extern UINT  TextThreadProc(LPVOID param);

// Рабочая функция потока синхронизации
UINT  SyncroThreadProc(LPVOID param)
{
  CSyncDlg*    pt = (CSyncDlg*) param;
  DWORD      dwResult;

  // Инициализация массива событий
  pt-> m_pEvents[0]  = CreateEvent(NULL, FALSE, FALSE, NULL);
  pt-> m_pEvents[1]  = CreateEvent(NULL, FALSE, FALSE, NULL);

  pt->m_eFinish.SetEvent();

  while(true)
  {
    // Ожидание событий диалогового окна
    dwResult = ::WaitForMultipleObjects(2, pt-> m_pEvents, FALSE,
          INFINITE);
    switch(dwResult)  // Определение события
    {
      case WAIT_OBJECT_0 + 0:  // Нажата кнопка Старт
          // Запуск рабочих потоков
```

```
            AfxBeginThread(TextThreadProc, pt);
            AfxBeginThread(TextThreadProc, pt);
            break;
        case WAIT_OBJECT_0 + 1:  // Нажата кнопка Отмена или OK
            // Ожидание завершения работы потоков
            ::WaitForSingleObject(pt->m_eFinish.m_hObject, INFINITE);
            // Уничтожение объектов событий
            CloseHandle(pt-> m_pEvents[0]);
            CloseHandle(pt-> m_pEvents[1]);
            // Закрытие диалогового окна
            ::PostMessage(pt->GetSafeHwnd(), WM_FINISH, 0, 0);
            return  0;
    }
  }
}
```

7. Откройте окно **Class View** (Просмотр классов), раскройте в нем папку **Sync**, щелкните правой кнопкой мыши на папке `CSyncDlg` и выберите в появившемся контекстном меню команду **Add | Add Variable** (Добавить | Добавить переменную). Появится диалоговое окно **Add Member Variable Wizard** (Мастер добавления переменной в класс).
8. Введите в текстовое поле раскрывающегося списка **Variable type** (Тип переменной) тип переменной `CEvent`, введите в текстовое поле **Variable name** (Имя переменной) идентификатор переменной `m_eFinish` и нажмите кнопку **Finish** (Готово). Соответствующая переменная будет добавлена в класс `CSyncDlg`.
9. В окне **Class View** (Просмотр классов) раскройте папку `CSyncDlg` и дважды щелкните левой кнопкой мыши на идентификаторе класса `CSyncDlg`. Откроется окно редактирования класса `SyncDlg.h`.
10. После строки `CEvent m_eFinish;` введите в следующий текст

    ```
    HANDLE m_pEvents[2];
    ```

11. После строки `void OnOK(void);` вставьте строку

    ```
    LRESULT OnFinish(WPARAM, LPARAM);
    ```

12. После строки `#include "afxmt.h"` вставьте строку

    ```
    #define  WM_FINISH  WM_USER + 0x100
    ```

13. Откройте окно редактирования класса `SyncDlg.cpp`.
14. Измените функции обработки сообщений класса `CSyncDlg` в соответствии с текстом листинга 12.5.

Листинг 12.5. Функции обработки сообщений класса `CSyncDlg`

```
// Запуск потоков на исполнение
void CSyncDlg::OnBnClickedStart(void)
{
  // Считывание состояния диалогового окна
  UpdateData();

  // Запуск рабочих потоков
/*
  AfxBeginThread(TextThreadProc, this);
  AfxBeginThread(TextThreadProc, this);
*/
  SetEvent(m_pEvents[0]);  // Установка события
}

// Выход из диалогового окна
void CSyncDlg::OnCancel(void)
{
/*
  // Проверка завершения работы потоков
  if(m_Count)
    return;

  // Вызов метода базового класса
  CDialog::OnCancel();
*/
  SetEvent(m_pEvents[1]);  // Установка события
}

// Выход из диалогового окна

void CSyncDlg::OnOK(void)
{
/*
  // Проверка завершения работы потоков
  if(m_Count)
    return;

  // Вызов метода базового класса
  CDialog::OnOK();
```

```
*/
  SetEvent(m_pEvents[1]);  // Установка события
}

// Функция обработки сообщения WM_FINISH
LRESULT CSyncDlg::OnFinish(WPARAM /*wParam*/, LPARAM /*lParam*/)
{
  CDialog::OnOK();
  return  0;
}
```

15. Внесите изменения в рабочую функцию потока вывода текста в соответствии с текстом листинга 12.6.

Листинг 12.6. Рабочая функция потока вывода текста

```
// Рабочая функция потока
UINT  TextThreadProc(LPVOID param)
{
  int        i;
  CString      szTemp;
  CSyncDlg*    pt = (CSyncDlg*) param;

  pt->m_eFinish.ResetEvent();  // Сброс события
  CSingleLock    FirstLock(&(pt->m_Critic));
  pt->m_Count++;         // Увеличение значения счетчика потоков
  for(i=0; i < 20; i++)
  {
    if(pt-> m_bSync)
      FirstLock.Lock();  // Блокировка доступа
    if(!pt-> m_bSync  |  |  FirstLock.IsLocked())  // Проверка блокировки
    {
      // Вывод строки
      szTemp.Format("Очень ");
      pt->m_Output.ReplaceSel(szTemp);
      Sleep(10);
      szTemp.Format("длинная ");
      pt->m_Output.ReplaceSel(szTemp);
      Sleep(10);
      szTemp.Format("строка ");
```

```
      pt->m_Output.ReplaceSel(szTemp);
      Sleep(10);
      szTemp.Format("номер ");
      pt->m_Output.ReplaceSel(szTemp);
      Sleep(10);
      szTemp.Format("%d \r\n", i);
      pt->m_Output.ReplaceSel(szTemp);
    }
    Sleep(100);  // Ожидание
    if(pt-> m_bSync)
      FirstLock.Unlock();  // Разблокирование доступа
  }
  pt->m_Count--;  // Уменьшение значения счетчика потоков
  if(!pt->m_Count)
    pt->m_eFinish.SetEvent();  // Отметка события
  return  0;
}
```

16. Измените карту сообщений класса `CSyncDlg` в соответствии с текстом листинга 12.7.

Листинг 12.7. Карта сообщений класса `CSyncDlg`

```
BEGIN_MESSAGE_MAP(CSyncDlg, CDialog)
  ON_WM_SYSCOMMAND()
  ON_WM_PAINT()
  ON_WM_QUERYDRAGICON()
  //}}AFX_MSG_MAP
  ON_BN_CLICKED(IDC_START, OnBnClickedStart)
  ON_MESSAGE(WM_FINISH,OnFinish)
END_MESSAGE_MAP()
```

17. Нажмите клавишу <F5> и запустите приложение на исполнение. Работа приложения практически не изменится, но при нажатии кнопок **Отмена** или **OK** в процессе вывода текста в диалоговое окно вывод текста будет продолжен, а при завершении этого процесса диалоговое окно будет закрыто.

Такие, казалось бы, незначительные отличия в работе приложения потребовали внесения в него существенных изменений. Это связано с тем, что вызов любой функции ожидания приостанавливает работу потока, в котором она была вызвана. Таким образом, вызов подобной функции в одной из функций класса диалогового окна приведет к приостановке работы всего диалогового окна. По-

скольку поток выводит текстовую информацию в диалоговое окно, то при помещении функции ожидания его завершения в функцию обработки сообщения диалогового окна приведет к тому, что поток диалогового окна будет ожидать завершения работы потока вывода текста, а поток вывода текста — возобновления работы потока диалогового окна для вывода в нем текста. В результате приложение зависнет.

Помещение функции ожидания завершения работы потока в главный поток приложения не исправит сложившееся положение, поскольку при вызове модального диалогового окна выполнение вызвавшего его потока приостанавливается. Для простоты можно считать, что модальное диалоговое окно работает в вызвавшем его потоке. Поэтому все функции ожидания необходимо поместить в специальный поток синхронизации. Этот поток синхронизации запускается перед вызовом функции `DoModal` класса диалогового окна и в качестве аргумента его рабочей функции передается указатель на объект данного класса.

В рабочей функции потока синхронизации производится инициализация массива дескрипторов объектов событий, определенного в классе диалогового окна. Определение данного массива в классе диалогового окна сделало его доступным во всех функциях данного приложения, а его инициализация в рабочей функции потока синхронизации объясняется тем, что в этой функции производится уничтожение созданных объектов, что облегчает контроль за их созданием и уничтожением. Этот массив пришлось помещать в заголовок данного класса вручную, поскольку мастер Add Member Variable Wizard по какой-то совершенно секретной причине не позволяет включать массивы в классы.

Для создания и инициализации объектов событий используется функция `CreateEvent`. Особое внимание следует обратить на второй и третий аргументы данной функции. Значение `FALSE` второго аргумента указывает на то, что после определения состояния данного объекта в функции ожидания он будет автоматически сброшен, а то же самое значение третьего аргумента указывает на то, что этот объект создается в неотмеченном состоянии.

После создания объектов классов синхронизации открывается бесконечный цикл наблюдения за их состоянием. В этом цикле вызывается функция `::WaitForMultipleObjects`. Первый аргумент данной функции определяет число объектов синхронизации, за состоянием которых производится наблюдение. Второй аргумент данной функции содержит массив дескрипторов этих объектов синхронизации. Значение `FALSE` третьего аргумента определяет, что данная функция возвратит управление потоку при отметке любого из объектов, за которым осуществляется наблюдение. Последний, четвертый, аргумент функции определяет, что ожидание отметки события будет продолжаться в течение неограниченного отрезка времени.

При завершении своей работы функция `::WaitForMultipleObjects` возвращает значение, определяющее, отметка какого из наблюдаемых объектов привела к возврату управления потоку. Это значение анализируется в операторе-переключателе.

Если возврат управления произошел вследствие отметки нулевого события, это означает, что пользователь нажал кнопку **Старт**. Обработка этого события заключается в запуске двух потоков вывода текста. Эту операцию можно было бы произвести и непосредственно в функции обработки сообщения диалогового окна, но в этом случае не было бы возможности продемонстрировать работу функции `::WaitForMultipleObjects`.

Если возврат управления произошел вследствие отметки первого события, это означает, что пользователь нажал кнопки **Отмена** или **OK**. В этом случае сначала вызывается функция `::WaitForSingleObject`. Первым ее аргументом является дескриптор объекта класса синхронизации, отметка которого ожидается, а вторым — величина интервала ожидания. Дескриптор объекта класса синхронизации является одним из открытых членов данного класса, а интервал ожидания в данном случае установлен бесконечным.

Вместо вызова этой функции можно было бы вставить следующий текст:

```
while(pt-> m_Count)
  Sleep(100);
```

Этот метод практически не уступает по эффективности использованию функции ожидания, но не использует объектов событий.

После завершения работы потока вывода текста в рабочей функции потока синхронизации с использованием функций `CloseHandle` уничтожаются объекты событий, дескрипторы на которые содержатся в соответствующем массиве. После этого классу диалогового окна посылается сообщение о необходимости прекращения его работы и осуществляется выход из рабочей функции потока синхронизации, что приводит к завершению работы данного потока.

Вместо посылки сообщения классу диалогового окна можно было бы непосредственно вызвать функцию `CDialog::EndDialog`, однако при попытке ее использования в приложении иногда возникали различные нештатные ситуации, а из справки по этой функции нельзя было заключить, можно ли ее использовать за пределами класса диалогового окна. Кроме того, использование сообщений является еще одним способом взаимодействия между потоками.

Сообщение посылается функцией `::PostMessage`. Первый аргумент данной функции содержит дескриптор окна, которому посылается данное сообщение. Для получения значения этого дескриптора используется функция `CWnd::GetSafeHwnd`. Вторым аргументом данной функции является идентификатор сообщения. Этот идентификатор может иметь любое свободное значение из диапазона пользовательских сообщений. Для того чтобы уменьшить вероятность конфликта значений, выбранных для различных пользовательских сообщений, данному идентификатору присваивается значение `WM_USER + 0x100`, т. е. на 256 больше, чем начальное значение диапазона значений пользовательских сообщений. Определение данного идентификатора помещено в файл заголовка класса, которому будет посылаться сообщение.

Все операции по созданию функции обработки пользовательского сообщения и включения его в карту сообщений класса, в полном соответствии с его названи-

ем, возлагаются на пользователя. Аргументы функции обработки пользовательского сообщения `OnFinish` не используются, поэтому их имена заремаркированы. Эти имена можно было бы и не указывать, но по традиции используется именно такой синтаксис. В этой функции вызывается функция `CDialog::OnOK`, завершающая работу диалогового окна и возвращающая значение `IDOK`. Хотя при этом класс диалогового окна завершает свою работу и не важно, какое значение возвратит функция обработки сообщения, синтаксис требует использования в ней оператора `return` и передачи ему целочисленного значения.

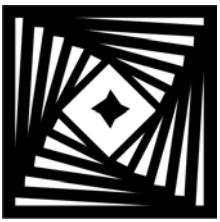
В функции обработки сообщений исходного приложения были внесены минимальные изменения. Из функции `OnBnClickedStart` были исключены функции запуска потоков и включена функция `SetEvent`, отмечающая событие, дескриптор которого передан ей в качестве аргумента. В функциях `OnCancel` и `OnOK` весь текст заменен вызовом функции `SetEvent`. Казалось бы, ее следует вызывать только в том случае, если поток вывода текста не завершил свою работу. Но в противном случае в приложении будут пропущены операции уничтожения объектов событий, дескрипторы которых содержатся в массиве.

В рабочей функции потока вывода текста прежде всего вызывается функция `CEvent::ResetEvent`. Вызов этой функции должен вызвать определенные вопросы у внимательного читателя. Дело в том, что по сведениям MSDN конструктор класса `ResetEvent` создает объект в неотмеченном состоянии, сброс которого производится автоматически. Поэтому вызов этой функции не только не нужен, но и недопустим. Однако серия проведенных опытов привела к очень интересному выводу: конструктор данного класса создает отмеченный объект, сброс которого должен производиться вручную, независимо от значений своих аргументов, определяющих эти режимы.

После декрементирования счетчика запущенных потоков производится проверка его нового значения и, если это значение нулевое, вызывается функция `CEvent::SetEvent` для отметки соответствующего объекта события.

Использование в данном приложении объекта класса `CEvent` вызвано исключительно желанием продемонстрировать принципы работы с объектом данного класса. Вместо использования данного объекта можно было бы, просто, увеличить размер массива `m_pEvents` на единицу и использовать этот дескриптор объекта синхронизации.

Глава 13



Справка в приложении

Наличие хорошо организованной справочной системы является одной из основных отличительных особенностей профессионального приложения. Отсутствие справки допустимо только в приложениях, создаваемых пользователем для решения текущих задач, и только в том случае, если он не собирается использовать, а тем более, модифицировать это приложение в дальнейшем. В противном случае ему постоянно придется вспоминать, какие функции заложены в данном приложении и как ими можно воспользоваться. В некоторых случаях вместо справочной системы можно использовать комментарии в тексте программы. Но этот способ применим только в программах, написанных для внутреннего употребления. В остальных случаях комментарии в тексте программы должны не заменять, а дополнять справочную систему.

Несмотря на особую роль справочной системы в оценке приложения потребителем многие программисты откладывают ее создание до полного завершения работы по отладке проекта. Хотя, во многих случаях, они не могут создать ее раньше вследствие некорректно выданного задания, что приводит к необходимости вносить кардинальные изменения в практически готовый проект. Создание справочной системы на более ранних стадиях приводит к необходимости написания нескольких практически независимых друг от друга справочных систем для каждой из промежуточных версий приложения.

Описание справочной системы приложения

Microsoft Windows предоставляет специальные функции и сообщения, позволяющие приложению, работающему под управлением данной операционной системы, создать свою справочную систему. Большинство подобных приложений используют различные методы доступа к справочной системе. Для каждой конкретной справки необходимо ответить на следующие вопросы.

- Каким образом она будет вызываться?
- В какое окно будет выводиться справочная информация?
- В какой форме она будет представлена?
- Каким образом справочная система будет включена в текст программы?

Ниже будут приведены возможные ответы на каждый из поставленных вопросов. Какой из них выбрать в каждом конкретном случае, решает сам пользователь.

Способы доступа к справочной системе

Различные виды справочной информации подразумевают различные способы доступа к ней. Например, для получения справки о назначении данной кнопки панели инструментов достаточно просто поместить на нее указатель мыши, поскольку в данном случае ясно, какая информация может понадобиться пользователю в данном конкретном случае. Если же пользователь хочет получить наиболее полную информацию по ключевому слову, то здесь не обойтись без вызова специального диалогового окна. Ниже перечислены основные способы выдачи запроса на получение справочной информации.

- Вызов команды меню.

  Многие приложения позволяют пользователю использовать для доступа к справочной системе приложения соответствующее меню в главном окне программы. Когда пользователь выбирает команду меню, соответствующая процедура окна получает сообщение `WM_COMMAND`, определяющее запрос к справочной системе. В ответ на этот запрос приложение выводит соответствующую информацию, например, список разделов, индекс или вводную информацию к разделу.

- Вызов с клавиатуры.

  При нажатии пользователем клавиши <F1> приложение получает соответствующее сообщение Windows. При этом система посылает сообщение `WM_HELP` окну, имеющему в данный момент фокус ввода. Если это окно является дочерним окном (например, элемент управления диалогового окна), функция `DefWindowProc` передает это сообщение родительскому окну. Если в момент нажатия клавиши <F1> фокус ввода принадлежал меню, система посылает сообщение окну, связанному с данным меню. В ответ на это сообщение приложение выводит информацию, связанную с данным окном, элементом управления или командой меню, которой принадлежал фокус ввода. Например, при выделении элемента управления в диалоговом окне и нажатии клавиши <F1> приложение выводит информацию о данном элементе управления.

  Аргумент `lParam` сообщения формата `WM_HELP` представляет собой указатель на объект структуры `HELPINFO`, содержащей детальную информацию об объекте, которому в данный момент принадлежит фокус ввода. Эта информация используется для определения характера выводимой информации. Кроме того, в объекте структуры `HELPINFO` содержатся координаты указателя мыши в момент нажатия клавиши <F1>. Эта информация также может использоваться для определения характера выводимой информации.

- Вызов с помощью мыши.

  Чтобы получить справочную информацию с помощью мыши, необходимо щелкнуть правой кнопкой в окне или левой кнопкой по кнопке в окне или элементе управления. В ответ на это приложение выведет информацию, связанную с данным окном или элементом управления.

При щелчке правой кнопкой мыши система посылает сообщение `WM_CONTEXTMENU` окну, в котором находился в это время указатель мыши. Если это окно является дочерним, то функция `DefWindowProc` передает это сообщение родительскому окну. В сообщении `WM_CONTEXTMENU` содержатся координаты указателя мыши, горизонтальная координата — в младшем слове аргумента `lParam`, а вертикальная — в старшем слове. Если пользователь щелкнул по элементу управления, то в аргументе `wParam` передается дескриптор этого элемента управления.

При щелчке левой кнопкой мыши по кнопке , расположенной в заголовке окна, система посылает сообщение `WM_HELP`. Чтобы добавить в заголовок окна кнопку со знаком вопроса, необходимо задать стиль `WS_EX_CONTEXTHELP` в функции `CreateWindowEx` при создании окна. В аргументе `lParam` сообщения `WM_HELP` имеется указатель на объект структуры `HELPINFO`, содержащей детальную информацию об объекте, которому в данный момент принадлежит фокус ввода, включая координаты указателя мыши в момент щелчка.

Кнопку со знаком вопроса рекомендуется использовать только в диалоговых окнах. Ранее для получения информации по диалоговому окну использовалась специальная кнопка в данном диалоговом окне. Теперь вместо нее рекомендуется применять кнопку со знаком вопроса в заголовке диалогового окна.

Способы представления справочной информации

После того как приложение получило запрос на вывод справочной информации, оно должно вывести ее на экран. Поскольку все базовые приложения Windows используют единый интерфейс при выводе справочной информации, настоятельно рекомендуется, чтобы все пользовательские приложения использовали этот интерфейс при выводе своей справки. Для доступа к этому интерфейсу используется функция `WinHelp`. Этой функции следует передать выводимую информацию и указать форму ее вывода.

После того как Microsoft осознала, хотя и с большой задержкой, роль Internet в современном мире и включила в состав Windows программу Internet Explorer, у нее появилась новая забота: до того, как будет принято судебное решение об изъятии данной программы из состава операционной системы, настолько глубоко интегрировать ее в Windows, чтобы это постановление стало бы технически невыполнимым. Одним из путей реализации этой задачи является перевод всех приложений на использование справочной системы в стиле Internet Explorer. Эта справочная система называется HTML Help.

Справочная информация может выводиться с использованием окон различных типов.

- Окно **Help Topics** (Справочная система).

  Это диалоговое окно, показанное на рис. 13.1, позволяет пользователю произвести поиск по ключевому слову, просмотреть оглавление или найти в справочных текстах определенное слово.

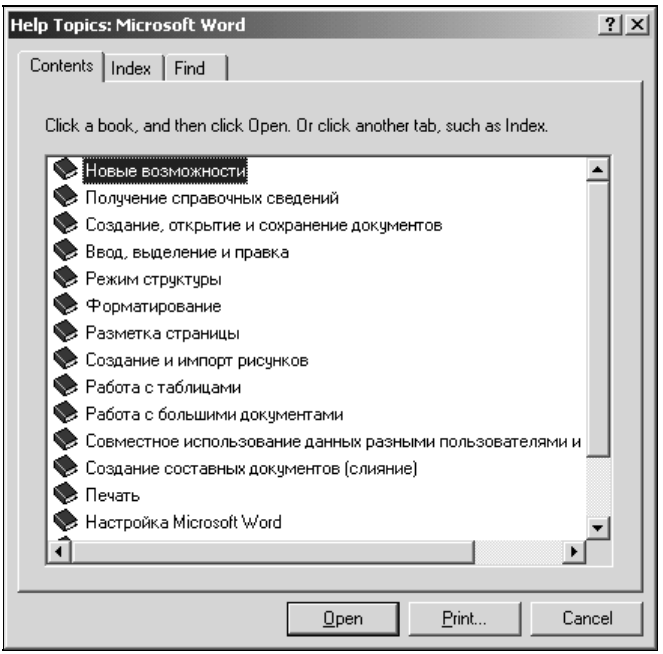


Рис. 13.1. Диалоговое окно **Help Topics**

- Обычное окно справки.

  В этом окне, показанном на рис. 13.2, выводится текст конкретной справки по интересующему вопросу. Окно можно свернуть в пиктограмму, максимизировать, изменить его размер или закрыть. Обычно это окно всегда располагается поверх всех окон, что позволяет использовать его в качестве подсказки при осуществлении некоторых действий, описанных в данном окне. Иногда окно справки называется вторичным.

- Всплывающее окно.

  Это окно, изображенное на рис. 13.3, используется для вывода справочной информации небольшого объема или кратких пояснений. Оно не имеет элементов управления и закрывается при щелчке мышью за его пределами. Это окно нельзя минимизировать или переместить.

- Окно **HTML Help** (Справка HTML).

  Это диалоговое окно, показанное на рис. 13.4, используется для вывода справочной информации в стиле Internet Explorer. Его можно рассматривать как комбинацию первых двух типов окон.

Особой формой справки является поиск слов или комбинаций слов непосредственно в тексте справки.

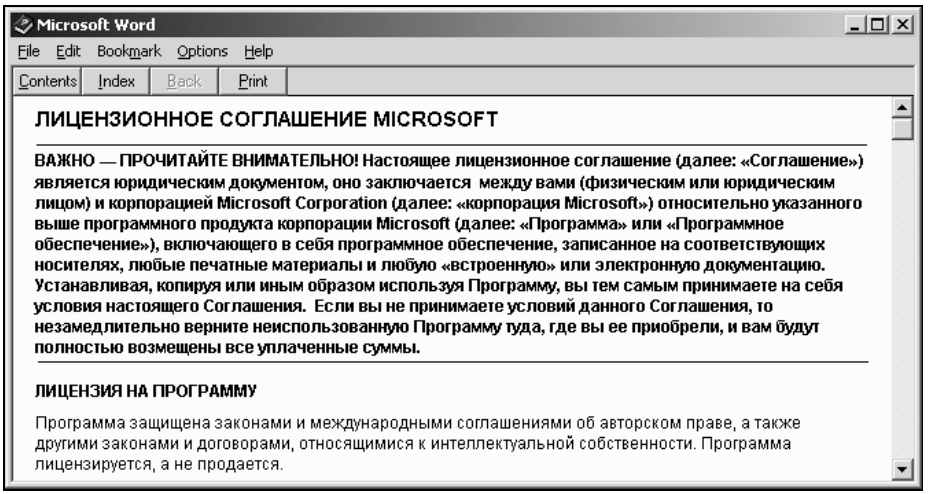


Рис. 13.2. Окно справки

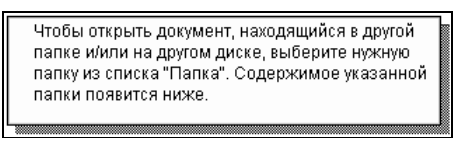


Рис. 13.3. Всплывающее окно справки

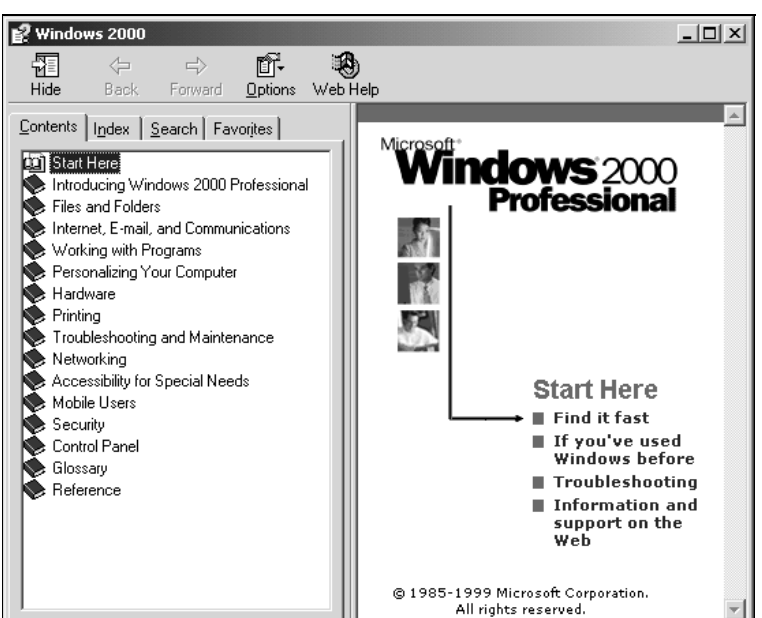


Рис. 13.4. Диалоговое окно **HTML Help**

При первой попытке раскрыть вкладку **Find** (Поиск), в окне **Help Topics** (Справочная система) запускается процесс настройки поиска. При этом в справочной системе приложения создается база данных ключевых слов по файлам справочной системы.

Чтобы настроить справочную систему приложения на поиск информации по ключевому слову:

1. Откройте окно **Help Topics** (Справочная система) и раскройте в нем вкладку **Find** (Поиск). На экране появится диалоговое окно мастера Find Setup Wizard (Мастер настройки окна поиска), изображенное на рис. 13.5.

Рис. 13.5. Диалоговое окно мастера Find Setup Wizard

2. Установите переключатель в требуемое положение и нажмите кнопку **Next** (Далее). Появится последняя вкладка мастера Find Setup Wizard (Мастер настройки окна поиска), изображенная на рис. 13.6.
3. Нажмите кнопку **Finish** (Готово). Появится диалоговое окно **Help Topics** (Справочная система), раскрытое на вкладке **Find** (Поиск), как это изображено на рис. 13.7.
4. Введите искомое слово или набор слов в текстовое поле **1. Type the word(s) you want to find** (Введите искомое слово или слова) или выберите его в окне списка **2. Select some matching words to narrow your search** (Выделите слова для ограничения поиска). В окне списка **3. Click a topic, then click Display** (Выберите нужный раздел и нажмите кнопку Display) появится список разделов, связанных с данным ключевым словом или набором слов.
5. В окне списка **3. Click a topic, then click Display** (Выберите нужный раздел и нажмите кнопку Display) выделите заголовок интересующего вас раздела и нажмите кнопку **Display** (Показать).

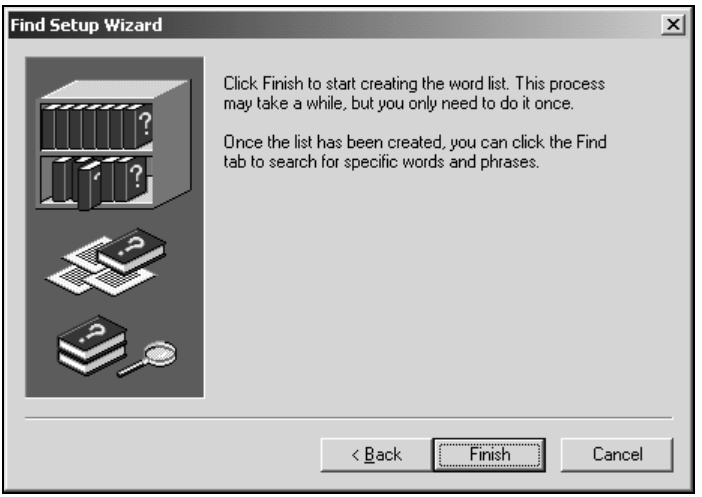


Рис. 13.6. Последняя вкладка мастера Find Setup Wizard

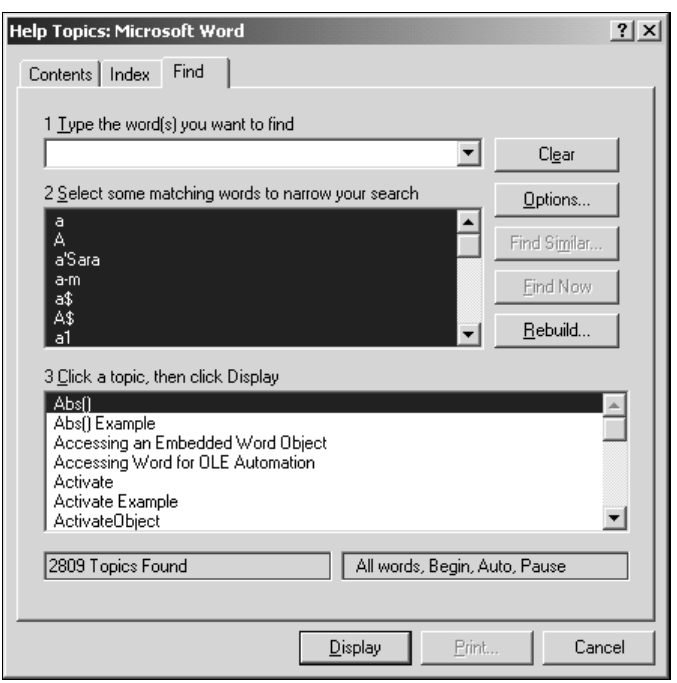


Рис. 13.7. Окно **Help Topics**, вкладка **Find**

6. На экране появится окно справки, показанное на рис. 13.8.

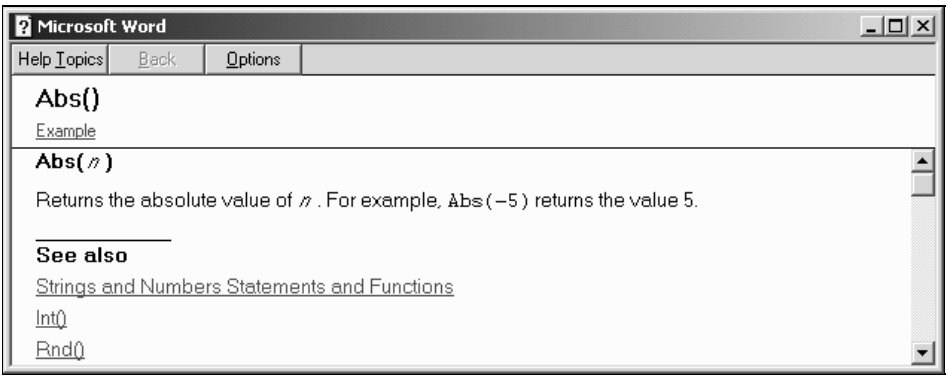


Рис. 13.8. Окно справки

При повторном открытии вкладки **Find** (Поиск) в окне **Help Topics** (Справочная система) процесс настройки пропускается и работа начинается с п. 4. Чтобы запустить процесс настройки поиска из вкладки **Find** (Поиск), нажмите кнопку **Rebuild** (Перестроить).

В окне **HTML Help** (Справка HTML) подготовка к выводу информации производится только при первом открытии вкладки **Index** (Предметный указатель), но она производится без участия пользователя.

Для вывода справочной информации необходимо создать файл, содержащий эту информацию, и передать его имя в качестве аргумента функции `WinHelp`. Файл справки должен иметь расширение hlp и соответствующий формат своего содержимого.

Данный файл содержит описание одного или нескольких разделов. Под разделом понимается самостоятельная единица информации, содержащая описание некоторого вопроса, последовательность действий, описание оглавления и т. п. Каждый раздел должен быть однозначно идентифицирован таким образом, чтобы справочная система Windows могла найти его по запросу. Для этого в ней используются идентификаторы разделов, но приложение обычно использует для этого контекстные идентификаторы (уникальные целочисленные значения).

Создатель файла справки должен в явном виде указать контекстный идентификатор раздела в секции `[MAP]` файла проекта, используемого для компиляции файла справки.

Если при задании имени файла справки в функции `WinHelp` не задан путь к данному файлу, то функция ищет файл в каталоге Help или в рабочей папке приложения. Кроме того, функция `WinHelp` может использовать имя файла справки, указанное в системном реестре по адресу:

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\Help
```

Для использования системного реестра необходимо создать переменную с тем же именем, что и файл справки. Значением этой переменной должен быть каталог, в котором расположен файл справки.

Если функция `WinHelp` не может найти файл справки, она выводит окно сообщения **Windows Help** (Справочная система Windows), изображенное на рис. 13.9.

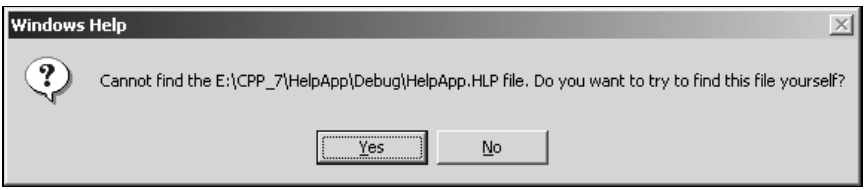


Рис. 13.9. Окно сообщения **Windows Help**

Данное окно сообщает о том, что указанный в приложении файл справки отсутствует. При нажатии кнопки **Yes** (Да) появляется диалоговое окно **Open** (Открыть), изображенное на рис. 13.10.

Рис. 13.10. Диалоговое окно **Open**

Это диалоговое окно позволяет пользователю указать операционной системе путь к файлу справки. Поскольку функция `WinHelp` сохраняет переданную ей информацию в системном реестре, при последующем запуске приложения она не будет спрашивать о местоположении данного файла справки.

Функция `WinHelp` должна вызываться в цикле обработки сообщений приложения. Ее вызов должен выглядеть следующим образом:

```
case ID_HELP:
{
```

```
    WinHelp(hWnd, "DemoApp.hlp", HELP_FINDER, 0);
    break;
  }
```

Первым аргументом данной функции является дескриптор вызывающего окна. Во втором аргументе передается путь к файлу справки, который, в данном случае, расположен в той же папке, что и само приложение. Третий аргумент указывает на то, что необходимо выводить справочную информацию в диалоговое окно справочной системы. Четвертый аргумент используется для передачи дополнительной информации, зависящей от значения третьего аргумента. Ниже приведен прототип функции `WinHelp`:

```
BOOL WinHelp(HWND hWndMain,       // дескриптор окна
             LPCTSTR lpszHelp,    // путь к файлу справки
             UINT uCommand,       // тип запроса
             DWORD dwData);       // дополнительная информация
                                  // для аргумента uCommand
```

Полный список возможных значений аргумента `uCommand` и соответствующих ему значений аргумента `dwData` приведен в справочной системе Visual C++. Некоторые из этих значений приведены в табл. 13.1.

Таблица 13.1. *Список некоторых значений аргумента* `UCommand`

Аргумент `UCommand`	**Описание**
`HELP_CONTEXT`	Выводит информацию по теме, определяемой значением аргумента `dwData`
`HELP_QUIT`	Завершает работу справочной системы
`HELP_INDEX`	Выводит индекс
`HELP_FINDER`	Выводит оглавление (требуется наличие файла CNT)
`HELP_HELPONHELP`	Выводит справку по справочной системе
`HELP_KEY`	Выводит информацию по ключевому слову, определяемому аргументом `dwData`

Для вызова приложения WinHelp в программах MFC используется функция `CWinApp::WinHelp`. Ниже приведен ее прототип:

```
virtual void WinHelp(DWORD dwData, UINT nCmd = HELP_CONTEXT);
```

Ее аргументы соответствуют третьему и четвертому аргументам функции API. Имя файла справки, как правило, совпадает с именем приложения. Если приложение называется DemoApp.exe, а файл справки — DemoApp.hlp, то для вызова приложения WinHelp можно использовать одну из следующих функций:

```
AfxGetApp->WinHelp(0, HELP_INDEX);   // старый стиль
AfxGetApp->WinHelp(0, HELP_FINDER);  // новый стиль,
                                     // требуется наличие файла .CNT
```

Примечание

Если имя файла справки отличается от имени приложения, путь к нему должен быть помещен в переменную `CWinApp::m_pszHelpFilePath`.

Ниже приведен пример использования функции `WinHelp` для получения справочной информации по операциооной системе Windows NT.

```
HWND hwnd;    // Дескриптор главного окна приложения
BOOL bResult  // используется для проверки результата
bResult = WinHelp(hWnd, "WINNT.HLP", HELP_CONTENTS, 0L);
```

В следующем примере производится открытие файла справки в Windows NT и в нем производится поиск по ключевой строке.

```
HWND hwnd;    // Дескриптор главного окна приложения
BOOL bResult  // используется для проверки результата
bResult = WinHelp(hWnd, "WINNT.HLP", HELP_KEY, (DWORD) "finding topics");
```

Для вывода диалогового окна **Help Topics** (Справочная система) при вызове функции `WinHelp` используется команда `HELP_FINDER`. Как уже говорилось выше, диалоговое окно **Help Topics** (Справочная система) позволяет пользователю выбирать разделы для вывода по ним справочной информации с использованием заголовков разделов, ключевых слов, связанных с данными темами, или слов и фраз, встречающихся в тексте данных разделов.

Данное диалоговое окно обычно выводится в ответ на выбор пользователем соответствующей команды меню. Кроме того, приложение может выводить данное диалоговое окно в ответ на нажатие пользователем клавиши <F1>, если в данный момент времени ни одно из окон, элементов управления или меню данного приложения не имеют фокуса ввода или не являются активными.

Старые приложения используют команду `HELP_CONTENTS` в функции `WinHelp` для раскрытия панели **Contents** (Содержание) в диалоговом окне **Help Topics** (Справочная система) и команду `HELP_INDEX` для раскрытия в нем панели **Index** (Предметный указатель). Теперь использовать эти команды не рекомендуется. Вместо них следует использовать команду `HELP_FINDER`.

Для вывода информации по конкретной теме в функции `WinHelp` используется команда `HELP_CONTEXT` и указывается контекстный идентификатор данного раздела. Обычно команда `HELP_CONTEXT` вызывается в ответ на выбор пользователем конкретного раздела в диалоговом окне **Help Topics** (Справочная система) и редко используется для предоставления информации по конкретному элементу управления или команде меню. С использованием данной команды пользователь может продолжить поиск информации в справочной системе, не возвращаясь в приложение.

Вызов команды `HELP_CONTEXT` приводит к вызову справочной системы Windows, позволяя пользователю продолжить поиск по другим темам в файле справки. Информация по этому сообщению обычно выводится в окне справки, имеющем заголовок, системное меню, кнопки минимизации и максимизации, главное

меню, дополнительную панель навигации, рамку изменяемого размера и рабочую область. Текст справки размещается в рабочей области окна. Для перемещения по файлу справки могут быть использованы расположенные в тексте ссылки и кнопки навигации в главном окне.

Для вывода всплывающего окна справки в качестве аргумента функции `WinHelp` необходимо указать команду `HELP_WM_HELP` или команду `HELP_CONTEXTMENU`. Чтобы вывод данной справки не препятствовал дальнейшей работе пользователя с приложением, всплывающее окно справки удаляется с экрана после первого же нажатия пользователем любой клавиши или первого же щелчка левой кнопкой мыши.

Команда `HELP_WM_HELP` используется для обработки сообщения `WM_HELP` в окне элемента управления. Поскольку большинство элементов управления передают сообщение `WM_HELP` функции `DefWindowProc`, то обработка данного сообщения производится функцией соответствующего класса диалогового окна (или функцией класса родительского окна).

Вместо того чтобы использовать отдельный контекстный идентификатор, функция обработки данного сообщения в классе диалогового окна должна передать массив, содержащий пары элементов управления и соответствующих им контекстных идентификаторов, функции `WinHelp` вместе с дескриптором элемента управления, содержащегося в переменной `hItemHandle`, являющейся членом структуры `HELPINFO`, передаваемой в качестве аргумента сообщения `WM_HELP`. Функция определяет идентификатор элемента управления, для которого посылается сообщение `WM_HELP` и использует соответствующий ему контекстный идентификатор для вывода на экран соответствующей справочной информации.

Команда `HELP_CONTEXTMENU` используется для обработки сообщения `WM_CONTEXTMENU`. Поскольку большинство элементов управления передают сообщение `WM_CONTEXTMENU` функции `DefWindowProc`, то обработка данного сообщения производится функцией соответствующего класса диалогового окна (или функцией класса родительского окна).

В функции обработки данного сообщения создается массив, содержащий пары элементов управления и соответствующих им контекстных идентификаторов. При вызове функции `WinHelp` в аргументе `wParam` передается дескриптор элемента управления. Таким образом, функция может определить контекстный идентификатор, соответствующий данному дескриптору, и вывести на экран соответствующую справочную информацию. В отличие от команды `HELP_WM_HELP`, команда `HELP_CONTEXTMENU` сначала раскрывает контекстное меню с единственной командой **What's This?** (Что это такое?). Если пользователь выбирает эту команду, функция `WinHelp` выводит информацию по данной теме. В противном случае запрос на получение справочной информации отменяется.

Для получения справки во всплывающем окне можно использовать команду `HELP_CONTEXTPOPUP`, указав ей контекстный идентификатор раздела справки. Эта команда работает аналогично команде `HELP_CONTEXT`, но использует для вывода информации всплывающее окно. Приложение может использовать эту команду в ответ на получение сообщения `WM_HELP` при выводе справочной информации по

командам меню и для окон, не являющихся элементами управления диалогового окна. Для более эффективного использования данной команды приложение должно назначить контекстные идентификаторы этим командам меню и окнам.

Контекстный идентификатор может быть назначен любому окну и любой команде меню в приложении. При посылке сообщения `WM_HELP` родительскому окну система включает в объект структуры `HELPINFO` контекстный идентификатор. Родительское окно передает этот контекстный идентификатор функции `WinHelp` для вывода соответствующего раздела справки.

Функция `SetWindowContextHelpId` позволяет назначить контекстный идентификатор окну или элементу управления, а функция `SetMenuContextHelpId` позволяет назначить контекстный идентификатор команде меню. Контекстный идентификатор окна или команды меню может быть получен с использованием функций `GetWindowContextHelpId` и `GetMenuContextHelpId` соответственно.

Справочная система Windows позволяет производить поиск по ключевому слову в файле справки. Ключевое слово представляет собой строку, с которой связаны один или несколько разделов в файле справки.

Для поиска по ключевому слову приложение использует команды `HELP_KEY`, `HELP_PARTIALKEY` или `HELP_MULTIKEY` в функции `WinHelp`. При вызове функции `WinHelp` необходимо указать используемую команду, ключевое слово, имя файла справки и дескриптор родительского окна. В любом случае, если в файле справки обнаружено только одно совпадение ключевого слова, функция `WinHelp` выводит справочную информацию по этой теме.

Поиск по ключевому слову ведется из вкладки **Index** (Предметный указатель) окна **Help Topics** (Справочная система), изображенного на рис. 13.11.

В текстовое поле **1. Type the first few letters of the word you're looking for** (Введите первые буквы искомого слова) вводится искомое ключевое слово. По мере его ввода содержимое в окне списка **2. Click the index entry you want, and the click Display** (Выберите термин или фразу и нажмите кнопку Display) прокручивается таким образом, чтобы в нем было выделено первое ключевое слово, совпадающее с введенным пользователем. Как только в окне списка **2. Click the index entry you want, and the click Display** (Выберите термин или фразу и нажмите кнопку Display) будет выделено требуемое ключевое слово, пользователь может нажать кнопку **Display** (Показать).

Если найдено несколько совпадений, функция выводит диалоговое окно **Topics Found** (Найденные разделы), изображенное на рис. 13.12. В данном диалоговом окне пользователь может выбрать раздел, по которому он хочет получить справку.

Если в файле справки не содержатся разделы, связанные с данным ключевым словом, то функция `WinHelp` или продолжает выводить предметный указатель (для команд `HELP_KEY` и `HELP_PARTIALKEY`), или выдает сообщение об ошибке (для команды `HELP_MULTIKEY`).

Приложение может производить поиск по нескольким ключевым словам при однократном вызове функции `WinHelp`. При этом различные ключевые слова разделяются точкой с запятой. (Проведение поиска по нескольким ключевым словам

невозможно в файлах справки, созданных для приложений Windows версии 3.*x*.) Кроме того, возможно проведение поиска по нескольким файлам справки, если указанный в качестве основного файл справки имеет связанный с ним файл контекста (с расширением cnt), в котором содержатся команды `:Index` или `:Link`.

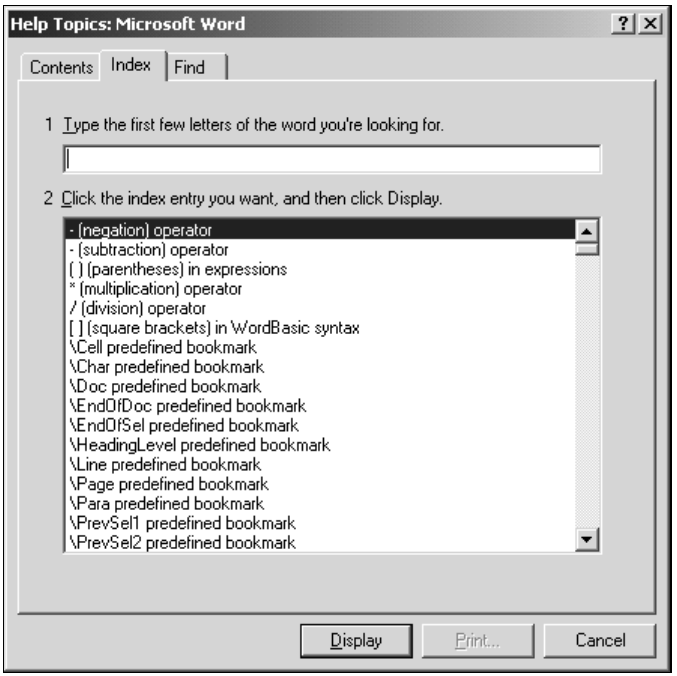


Рис. 13.11. Окно **Help Topics**, вкладка **Index**

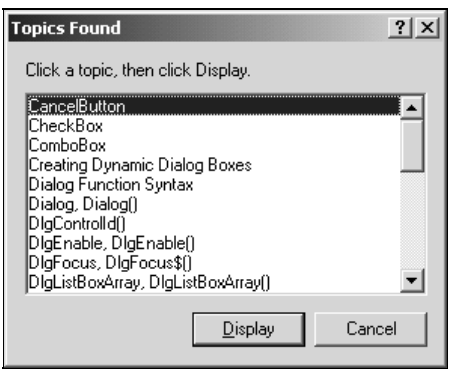


Рис. 13.12. Диалоговое окно **Topics Found**

Использование команды `HELP_KEY` в функции `WinHelp` позволяет производить поиск ключевого слова во всех файлах, указанных в данных командах. Использование в данной функции команд `HELP_MULTIKEY` и `HELP_PARTIALKEY` позволяет производить поиск во всех файлах, кроме указанных в команде `:Link`.

По умолчанию справочная система Windows использует только таблицу ключевых слов, отмеченную сноской **K** в исходном файле справки. Пользователь может указать справочной системе Windows на необходимость создания дополнительных таблиц ключевых слов, добавляя при определении ключевого слова в исходном файле справки другие сноски, отличные от **K**. (При этом следует помнить, что сноска **A** зарезервирована для других целей.) Любая дополнительная таблица ключевых слов должна быть определена с использованием ключевого слова `MULTIKEY` в разделе `[OPTIONS]` файла проекта при компиляции файла справки.

Для вывода в панели **Index** (Предметный указатель) таблицы ключевых слов, отличной от таблицы **K**, приложение может использовать команду `HELP_SETINDEX` в функции `WinHelp`.

Для проведения поиска по ключам, содержащимся в альтернативных таблицах ключевых слов, необходимо использовать команду `HELP_MULTIKEY`. Ключевое слово и таблица ключевых слов определяются объектом структуры `MULTIHELP`, передаваемым функции `WinHelp`.

При выводе справочной информации функцией `WinHelp` она использует для этого окно, определяемое сноской ">" данного раздела, окно, определяемое командой `:Base` в контекстном файле, или главное окно справочной системы. Если главное окно справочной системы в момент вызова функции `WinHelp` уже открыто для другого файла справки, функция скрывает главное окно на момент поиска. В этом случае одновременное закрытие окон **Topics Found** (Найденные разделы) и **Help Topics** (Справочная система) закрывает главное окно справочной системы.

Главное окно справочной системы Windows иногда называется также первичным окном. Справочная система Windows может выводить информацию и во вторичные окна. В отличие от первичного окна, вторичное не содержит контекстного меню. Вторичное окно может содержать панель навигации, в которую могут быть добавлены дополнительные кнопки. Кроме того, для этого окна может быть выбран режим, в котором его высота и ширина автоматически устанавливаются справочной системой Windows исходя из объема выводимой информации.

Вторичные окна справки должны определяться в разделе `[WINDOWS]` файла проекта справочной системы. При этом необходимо указать имя и, возможно, начальный размер и положение каждого окна. Чтобы сообщить справочной системе Windows о необходимости вывести данную тему во вторичном окне, следует добавить к имени файла справки угловую скобку (>) и имя вторичного окна. Полученная строка передается в качестве аргумента функции `WinHelp`.

Для изменения размера и положения первичного или вторичного окна приложение должно внести соответствующие изменения в объект структуры `HELPWININFO`

и передать команду HELP_SETWINPOS функции WinHelp. Структура HELPWININFO содержит имя окна, его размер и положение на экране.

Для обучения пользователя методам решения некоторых задач, встающих перед ним в процессе работы с приложением, используется последовательность окон, каждое из которых содержит описание одного из шагов выполнения данной задачи. Эти окна выводятся в определенной последовательности, определяющейся последовательностью описанных в них шагов. Эти окна, кроме текста, содержат кнопки, связанные с макросами TCard, позволяющие пользователю выбирать следующее окно справки в зависимости от результатов его действий, предпринятых на данный момент. Данные последовательности окон могут состоять только из вторичных окон и в них не могут находиться ссылки на другие разделы, содержащиеся в файле справки.

Вывод последовательности окон инициируется функцией WinHelp, в которой наряду с другими командами такими, как HELP_CONTEXT, указана команда HELP_TCARD. При нажатии пользователем на кнопки окна справки, вызове макросов TCard или закрытии окна справочная система Windows посылает приложению сообщение WM_TCARD. Аргумент wParam данного сообщения определяет нажатую кнопку или предпринятое пользователем действие, а аргумент lParam содержит дополнительную информацию, структура которой определяется значением аргумента wParam.

Справочная система Windows требует от приложения явной команды на свое закрытие, чтобы иметь возможность освободить все ресурсы, используемые для поиска информации в файле справки. Приложение может в любой момент закрыть справочную систему, вызвав функцию WinHelp и передав ей в качестве аргумента команду HELP_QUIT. Этот метод неприменим к всплывающим окнам справки, которые не могут закрываться приложением.

Если любое приложение вызвало функцию WinHelp, то оно должно завершить работу со справочной системой до того, как оно закроет свое главное окно (например, в ответ на сообщение WM_DESTROY в главном окне приложения). Для прекращения работы со справочной системой достаточно всего один раз вызвать функцию WinHelp независимо от того, сколько файлов справки было открыто на данный момент. Справочная система продолжает работу до тех пор, пока все приложения или библиотеки динамической компоновки (dll) не завершили с ней работу.

Чтобы закрыть последовательность окон в справочной системе Windows, необходимо одновременно передать функции WinHelp команды HELP_TCARD и HELP_QUIT. Приложение не должно закрывать последовательность окон в справочной системе, если до этого ее закрыл пользователь. Справочная система Windows извещает приложение о закрытии пользователем последовательности окон, посылая ему сообщение WM_TCARD, в котором аргумент wParam имеет значение IDCLOSE.

Формы представления справочной информации

Форма представления справочной информации в основном определяется причинами, по которым эта информация вызывается. Справочная информация может быть представлена в форме:

- *контекстной справки.* (Данная форма служит для ответа на вопросы типа "Для чего предназначена данная кнопка?" или "Что нужно ввести в это текстовое поле?");
- *справки о методике решения задачи.* (Данная форма служит для получения описания алгоритма действий при решении конкретной задачи, решаемой данным приложением. Например, о том, как распечатать документ);
- *справки по ключевому слову.* (Данная форма служит для предоставления развернутой информации по конкретному вопросу).

В приложениях Windows используются два основных типа контекстной справки. Первый из них при нажатии клавиши <F1> вызывает систему WinHelp, используя контекстный идентификатор, соответствующий активному объекту.

Второй вызывается при нажатии комбинации клавиш <Shift>+<F1>. В этом режиме указатель мыши приобретает форму курсора справки (комбинация стрелки и знака вопроса) и для получения справки по конкретному элементу управления пользователю нужно установить на нем курсор справки и щелкнуть левой кнопкой мыши.

Нажатие пользователем клавиши <F1> позволяет получить справку по активному окну или команде меню. При этом приложение не переводится в специальный режим вывода справочной информации. Эта информация исчезает при первом же активном действии пользователя (нажатии клавиши или щелчке мыши).

Справка по методике решения задачи и справка по ключевому слову иногда объединяются под общим наименованием *командной справки*, поскольку в большинстве случаев для их вызова используется команда меню.

Программирование справочной системы

Вопрос о способах программирования справочной системы приложения является наиболее важным для программиста, поскольку именно эта задача полностью возлагается на его плечи. Поэтому ответ на него вынесен в отдельный раздел.

Для обеспечения возможности работы со справочной системой в приложении необходимо при его создании установить флажок **Context-sensitive Help** (Контекстная справка) во вкладке **Advanced Features** (Дополнительные возможности) диалогового окна **MFC Application Wizard - Help** (Мастер создания приложений MFC), как это показано на рис. 13.13.

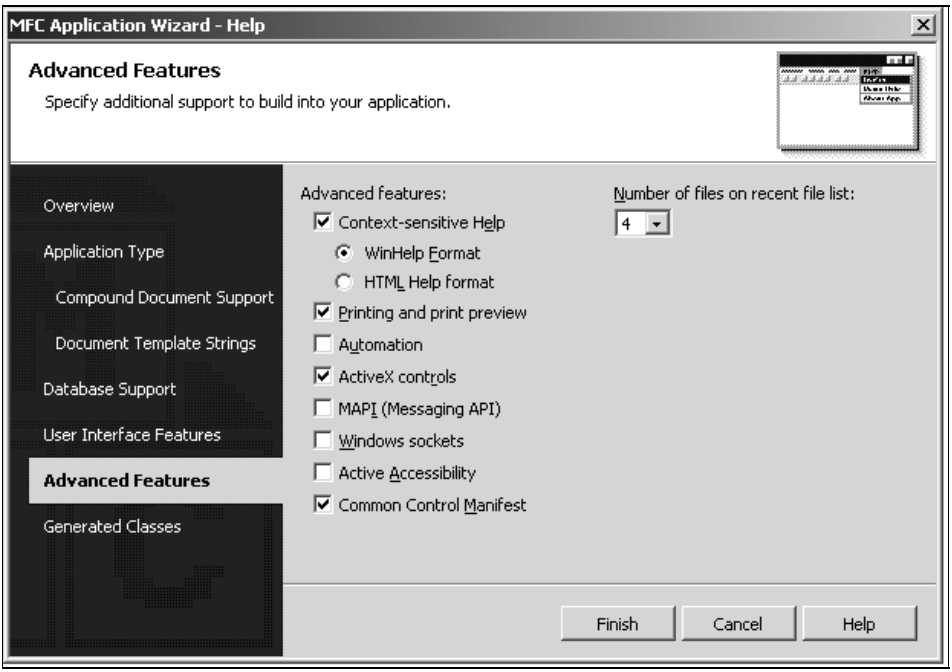


Рис. 13.13. Установка флажка **Context-sensitive Help**

Компоненты справочной системы

При создании справочной системы используется большое количество различных файлов. Большинство из них являются промежуточными для создания файла справки, имеющего расширение hlp. Ниже приведены расширения файлов, создаваемых мастером AppWizard и компилятором справки.

- h — файл заголовка (header file) содержит определения идентификаторов ресурсов и идентификаторы разделов справки, которые будут использоваться в программах.
- hm — файл адресации справок (help mapping file) содержит идентификаторы разделов справки. Файл, имеющий имя приложения и данное расширение, генерируется всякий раз при компиляции приложения.
- rtf — файл расширенного текстового формата (rich text format) содержит тексты справок по каждому разделу.
- cnt — файл таблицы содержания, используемый для подготовки вкладки **Contents** (Предметный указатель) в диалоговом окне **Help Topics** (Справочная система). (Файл, имеющий имя приложения и данное расширение, необходимо поставлять в составе приложения вместе с файлом hlp).

- hpj — файл проекта справочной системы (help project file), имеющий имя приложения и данное расширение, объединяет файлы с расширениями hm и rtf, используемые при создании файла hlp.

В процессе использования приложения справочная система создает и другие типы файлов. Все эти файлы имеют имя приложения, а их расширения перечислены ниже.

- gid — файл конфигурации, как правило, скрытый.
- fts — файл поиска по всему тексту, который создается при поиске раздела в текстах справки.
- ftg — файл группового списка поиска по всему тексту, который также создается при поиске раздела в текстах справки.

Рассмотрим подробнее файл проекта справочной системы, представляющий собой обычный текстовый файл, имеющий формат ASCII, который может быть создан в любом текстовом редакторе. Он форматируется как INI-файл. В табл. 13.2 приведено описание наиболее часто используемых разделов этого файла. Для получения более детальной информации о файле проекта справочной системы можно обратиться к интерактивной справке приложения Microsoft Help Workshop, который обычно и используется для работы с этим файлом.

Таблица 13.2. *Наиболее часто используемые разделы файла проекта справочной системы*

Раздел	Описание
[OPTIONS]	Определяет режим компиляции файла справки
[FILES]	Определяет файлы формата RTF, используемые при создании файла HLP
[WINDOWS]	Определяет первичное окно справочной системы. Требуется при реализации контекстной справки
[ALIAS]	Сопоставляет один контекстный идентификатор другому

Ниже приведен исходный вид простейшего файла проекта справочной системы:

```
[OPTIONS]
ERRORLOG = DemoApp.err
CONTENTS = HID_CONTENTS
TITLE = DemoApp Help

[FILES]
DemoApp.rtf

[WINDOWS]
Main = "DemoApp Help",(190,100,560,630),,,(192,192,192)
```

В этом файле в разделе `[OPTIONS]` определен файл, в который будут записываться все ошибки, предупреждения и примечания компоновщика справочной системы, определена ссылка на идентификатор корневого раздела справочной системы и определен заголовок справочной системы. В разделе `[FILES]` содержится имя текстового файла справочной системы, а в разделе `[WINDOWS]` определены параметры главного окна справочной системы.

Идентификаторы разделов используются для связи текста справки со справочной системой.

Кроме этого, мастер MFC Application Wizard при создании многооконного приложения (без поддержки баз данных и технологии OLE) при установке флажка **Context-sensitive Help** (Контекстная справка) внесет в остальные файлы проекта следующие изменения:

- в карту сообщений класса `CMainFrame` будут добавлены макросы обработки команд `ID_HELP_FINDER` и `ID_DEFAULT_HELP`. Для обработки сообщений в этих макросах будут использованы функции — члены класса `CMDIFrameWnd`;
- в панель инструментов будет добавлена кнопка **Help** (Помощь);
- в обе версии меню, создаваемые мастером AppWizard (в меню, появляющемся при отсутствии в приложении открытых окон документа, и в меню, появляющемся при наличии в приложении открытого окна документа), добавляется команда **Help Topics** (Справочная система);
- добавляются команды обработки нажатия на клавишу <F1> (`ID_HELP`) и на комбинацию клавиш <Shift>+<F1> (`ID_CONTEXT_HELP`);
- помещаемое по умолчанию в строку состояния сообщение **Ready** (Готово) будет заменено сообщением **For Help, press F1** (Для получения справки нажмите F1);
- в таблицу строковых ресурсов, расположенную в файле с именем приложения и расширением rc, добавляется строковый ресурс `AFX_IDS_HELPMODEMESSAGE`, содержащий текст **Select an object on which to get Help** (Укажите объект, по которому хотите получить справку). Этот текст выводится в строку состояния при посылке команды `ID_CONTEXT_HELP`;
- в таблицу строковых ресурсов будет добавлено сообщение, выводимое в строку состояния при выделении команды меню **Help Topics** (Справочная система);
- в проект включается файл afxcore.rtf, содержащий текст справочной информации по стандартным командам меню;
- если во вкладке **Advanced Features** (Дополнительные возможности) диалогового окна **MFC Application Wizard - Help** (Мастер создания приложений MFC) пользователь не сбросил флажок **Printing and print preview** (Печать и предварительный просмотр печати), в проект включается файл afxprintf.rtf, содержащий текст справочной информации по процессу печати и предварительного просмотра документа;

- в проект будут включены двадцать два файла графических изображений в формате BMP, используемые в качестве иллюстраций в различных разделах справки.

Обработка сообщений справочной системы

Когда пользователь выбирает команду в меню **Help** (Помощь) или нажимает кнопку **Help** (Помощь) в диалоговом окне, система, как и всегда в подобном случае, получает сообщение `WM_COMMAND`. Это сообщение обрабатывается соответствующей функцией обработки сообщения, вызывающей систему WinHelp. Таким образом, вызывается справка по ключевому слову или по методике решения задачи.

При щелчке правой кнопкой мыши в окне посылается сообщение `WM_CONTEXTMENU`. В ответ на это сообщение на экране должно появиться контекстное меню, в котором пользователю необходимо указать действие, которое он желает произвести. При работе со справочной системой это меню, как правило, содержит только одну команду **What's This?** (Что это такое?), при выборе которой выводится контекстная справка по указанному элементу управления.

Большинство приложений Windows позволяют пользователю нажать клавишу <F1> для получения контекстной информации. При нажатии этой клавиши приложение способно определить, какой объект (окно, элемент управления, команда меню и т. д.) имеет в это время фокус ввода, считать связанную с ним справочную информацию и передать ее приложению WinHelp, выводящему эту информацию на экран. Для этого активному окну или команде меню посылается сообщение `WM_HELP`. В ответ на это сообщение приложение и выводит справочную информацию об активном окне, элементе управления или команде меню.

Идентификаторы контекстных ссылок определяются в разделе `[MAP]` файла hpj. В этот раздел помещаются инструкции `#define` и `#include`, а также инструкция присваивания. Во многих случаях в качестве идентификаторов контекстных ссылок используются идентификаторы ресурсов программы. При этом в раздел `[MAP]` помещается всего одна инструкция `#include`, как это показано ниже:

```
#include DemoApp.h
```

В этом случае необходимо использовать одни и те же идентификаторы для имен меток перехода в файле формата RTF и имен ресурсов. Другой подход основан на использовании раздела `[ALIAS]`, в котором идентификаторы ресурсов связываются с именами меток перехода, используемых в файле формата RTF. Ниже приведен пример такого подхода:

```
[ALIAS]
ID_FIRST=HID_FIRST
ID_SECOND=HID_SECOND
ID_THIRD=HID_THIRD
ID_HELP=HID_DEMO_MENU
IDM_ABOUT=HID_MAIN_INDEX
IDM_EXIT=HID_APP_EXIT
```

```
[MAP]
ID_FIRST=0x9000
ID_SECOND=0x9001
ID_THIRD=0x9002
ID_HELP=32771
IDM_ABOUT=104
IDM_EXIT=105
```

Если требуется получить справку по элементу управления диалогового окна, то класс элемента управления пересылает сообщение классу родительского диалогового окна. Класс диалогового окна должен содержать функции обработки сообщений `WM_HELP` и `WM_CONTEXTMENU`.

В аргументе `lParam` сообщения `WM_HELP` передается указатель на объект структуры `HELPINFO`. В переменной `hItemHandle` данного объекта хранится дескриптор элемента управления, по которому был произведен щелчок правой кнопкой мыши, а в переменной `iCtrlId` — его идентификатор. Этот дескриптор необходимо передать функции `WinHelp` вместе с командой `HELP_WM_HELP`, именем файла справки и указателем на массив идентификаторов.

Ниже приведен фрагмент цикла обработки сообщений приложения, выводящий контекстную справку по команде меню:

```
case WM_HELP:
{
  HELPINFO & helpInfo = * reinterpret_cast<LPHELPINFO>(lParam);
  if(helpInfo.iContextType == HELPINFO_MENUITEM)
  {
    WinHelp(hWnd, "DemoApp.hlp", HELP_CONTEXT, helpInfo.iCtrlId);
  }
  break;
}
```

Функция `WinHelp` осуществляет поиск в массиве переданного ей идентификатора элемента управления и по нему определяет контекстный идентификатор раздела в файле справки. Затем функция передает полученный контекстный идентификатор справочной системе Windows, которая отыскивает связанную с ним тему и выводит ее текст во всплывающее окно. Если элемент управления, на котором был произведен щелчок левой кнопкой мыши, имеет идентификатор −1, то система ищет следующий элемент управления (тот, к которому перейдет фокус ввода при нажатии клавиши табуляции, если в данный момент он принадлежит данному элементу управления) и использует его идентификатор для поиска контекстного идентификатора. Поэтому статический текст следует располагать в файле ресурсов перед идентификатором элемента управления, к которому он относится. То есть создавать в редакторе ресурсов статический текст до создания соответствующего элемента управления.

Нажатие клавиши <F1> при наличии в приложении активного меню или диалогового окна обычно приводит к посылке команды с идентификатором `ID_HELP`. Связь между данной клавишей и командой устанавливается в таблице акселераторов. Команда `ID_HELP` может также посылаться кнопкой диалогового окна, имеющей данный идентификатор.

Команда `ID_HELP` обрабатывается как обычная команда, пока она не вызовет свою функцию обработки. Если приложение имеет справочную систему, команда `ID_HELP` обрабатывается функцией `CWinApp::OnHelp`. Поскольку предусмотренная по умолчанию обработка команд не позволяет определить необходимый для получения справочной информации контекст, данная команда всегда направляется в объект приложения, в котором вызывается обычная процедура поиска запрошенной справочной информации.

Функция `CWinApp::OnHelp` производит следующие действия:

- проверяет наличие активного вызова функции `AfxMessageBox`, в котором указан идентификатор справки. Если в данный момент имеется активное окно сообщения, вызывается система WinHelp с контекстом, связанным с данным окном сообщения;
- если в приложении отсутствует активное окно сообщения, функция `CWinApp::OnHelp` посылает сообщение `WM_COMMANDHELP` активному окну. Сообщение `WM_COMMANDHELP` является внутренним сообщением библиотеки MFC, принимаемым активным окном при запросе справочной информации. Когда приложение получает это сообщение, оно может вызвать функцию `CWinApp::WinHelp` с контекстом, соответствующим внутреннему состоянию окна. Если это окно не вызовет систему WinHelp, то же самое сообщение посылается родительскому окну активного окна и эта процедура продолжается до тех пор, пока данное сообщение не будет обработано или не будет достигнуто окно верхнего уровня в иерархии;
- если сообщение остается необработанным, вызывается справочная информация, предусмотренная по умолчанию. Для этого главному окну программы посылается сообщение `ID_DEFAULT_HELP`. Эта команда обычно обрабатывается функцией `CWinApp::OnHelpIndex`.

Использование команды `HELP_CONTEXTMENU` приводит к тому, что справочная система Windows выводит контекстное меню перед выводом справочной информации. Это меню определено в системе и позволяет пользователю выводить справочную информацию об отдельном элементе управления или вызывать диалоговое окно **Help Topics** (Справочная система).

Работа функции `WinHelp` при обработке сообщения `WM_CONTEXTMENU` аналогична ее работе при обработке сообщения `WM_HELP` за двумя исключениями:

- в сообщении `WM_CONTEXTMENU` используется аргумент `wParam`, в котором передается дескриптор элемента управления, посылающего сообщение;
- в качестве аргумента функции используется команда `HELP_CONTEXTMENU`, а не `HELP_WM_HELP`.

Ниже приведен пример создания контекстной справки в диалоговом окне.

```
LRESULT CALLBACK MyDlgProc(HWND hwndDlg, UINT uMsg, WPARAM wParam,
                                                    LPARAM lParam)
{
  // Создание массива идентификаторов элементов управления
  // и контекстных идентификаторов.
  static  DWORD  aIDs[] =
  {
    IDC_GO, HIDD_GO,
    IDOK, HIDD_CONTEXT_OK,
    IDCANCEL, HIDD_CONTEXT_CANCEL,
    0, 0
  };

  // Вызов компонентов справочной системы
  switch (uMsg)
  {
    case WM_HELP:
      WinHelp(((LPHELPINFO) lParam)->hItemHandle,
      "MyHelp.hlp", HELP_WM_HELP, (DWORD) (LPSTR) aIDs);
      break;
    case WM_CONTEXTMENU:
      WinHelp((HWND) wParam, "MyHelp.hlp",
      HELP_CONTEXTMENU, (DWORD) (LPVOID) aIDs);
      break;
    // Обработка других сообщений.
    ...
  }
  return FALSE;
}
```

Приложение может задать размер, позицию и режим отображения вторичного окна справки при использовании в функции `WinHelp` команды `HELP_SETWINPOS` и внесении соответствующих изменений в значения переменных передаваемого при этом объекта структуры `HELPWININFO`. Переменные данной структуры позволяют задать имя окна, его размер, положение и режим отображения.

Ниже приведен пример изменения параметров вторичного окна с именем "help_wnd". Это имя должно быть определено в разделе `[WINDOWS]` файла проекта справочной системы.

```
BOOL DoWindowSize(VOID)
{
```

```
  HANDLE    hWnd;
  LPHELPWININFO  lpHwi;
  WORD    Size;
  char*    szWndName = "help_wnd";

  // Выделение и блокировка памяти
  // для объекта структуры LPHELPWININFO
  Size  = sizeof(HELPWININFO) + lstrlen(szWndName);
  hWnd  = GlobalAlloc(GHND, Size);
  lpHwi  = (LPHELPWININFO) GlobalLock(hWnd);

  // Заполнение объекта структуры LPHELPWININFO
  lpHwi-> wStructSize  = Size;
  lpHwi-> x    = 64;          // горизонтальная позиция
  lpHwi-> y    = 64;          // вертикальная позиция
  lpHwi-> dx    = 256;        // ширина
  lpHwi-> dy    = 256;        // высота
  lpHwi-> wMax    = SW_SHOW;  // режим отображения
  lstrcpy(lpHwi-> rgchMember, szWndName);  // имя окна

  // Вызов справочной системы
  WinHelp(hwnd, "MyHelp.hlp", HELP_SETWINPOS, (DWORD) lpHwi);

  // Освобождение памяти
  GlobalUnlock(hWnd);
  GlobalFree(hWnd);

return;
}
```

При вызове контекстной справки с использованием комбинации клавиш <Shift>+<F1>, соответствующих команд меню или кнопок панели инструментов вызывается команда `ID_CONTEXT_HELP`. При наличии в приложении активного меню или модального диалогового окна для обработки данной команды не используется стандартный фильтр сообщений, поэтому данную команду можно применять только во время работы главного цикла обработки сообщений приложения (`CWinApp::Run`).

После вызова подобной справки указатель мыши приобретает форму курсора справки при отображении над всеми областями приложения даже в том случае, когда при отображении в данной области указатель мыши имеет особую форму (например, над толстой рамкой окна). Пользователь может использовать мышь

или клавиатуру для выбора команды. Вместо выполнения команды выводится справка по данному элементу управления.

Кроме того, пользователь может щелкнуть по любому видимому объекту на экране, например по кнопке панели инструментов, и получить справку по данному объекту. В этом режиме используется функция `CWinApp::OnContextHelp`. При этом все клавиши на клавиатуре, не связанные с клавишами, обеспечивающими доступ к меню, блокируются. Передача команд производится через функцию `PreTranslateMessage`, позволяя использовать предопределенные комбинации клавиш и получать справку по данной команде.

Если в данном режиме работы предусмотрена особая реакция или особая трактовка команд в функции `PreTranslateMessage`, то перед выполнением этих операций следует проверить значение переменной `m_bHelpMode` объекта класса `CWinApp`. При использовании функции `PreTranslateMessage` в объектах класса `CDialog` она, например, вызывает функцию `IsDialogMessage`, что препятствует использованию клавиш перемещения по элементам управления диалогового окна в данном режиме получения контекстной справки. Кроме того, в цикле обработки сообщения продолжает вызываться функция `CWinApp::OnIdle`.

При выборе пользователем команды меню выводится справка по данной команде (с использованием описанной ниже команды `WM_COMMANDHELP`). Если пользователь щелкнет левой кнопкой мыши в области окна приложения, то функция `OnContextHelp` посылает сообщение `WM_HELPHITTEST` окну, в котором был произведен щелчок. Если данное окно возвращает ненулевое значение, то это значение используется в качестве контекста для справки. Если возвращается нулевое значение, то функция `OnContextHelp` посылает это сообщение родительскому окну (и так далее, пока не будет возвращен контекст справки или не будет достигнуто верхнее окно в иерархии). Если контекст справки не может быть определен, главному окну приложения посылается сообщение `ID_DEFAULT_HELP`, обычно обрабатываемое функцией `CWinApp::OnHelpIndex`.

Сообщение `WM_HELPHITTEST` представляет собой внутреннее сообщение библиотеки MFC, посылаемое активному окну, в котором был произведен щелчок мыши при нажатой комбинации клавиш <Shift>+<F1>. В ответ на данное сообщение окно возвращает значение типа `DWORD`, содержащее идентификатор справки, используемый в функции `WinHelp`.

Русификация файла ресурсов

При создании Visual C++ был определен достаточно узкий список языков, которые предполагалось использовать при создании приложений с использованием данной системы программирования. Выбор языка ресурсов производится в раскрывающемся списке **Resource language** (Язык ресурсов), расположенном во вкладке **Application Type** (Тип приложения) диалогового окна **MFC Application Wizard - Help** (Мастер создания приложений MFC). Пока что он позволяет выбрать только английский язык, но есть надежда, что этот список будет впослед-

ствии расширен и позволит выбирать немецкий, испанский, французский и итальянский языки.

Кроме того, в дистрибутив Visual C++ включены ресурсы, необходимые для создания приложений на японском, корейском и китайском языках. Однако в Visual C++ 6.0 по непонятной причине соответствующие файлы ресурсов игнорировались при компиляции приложения и в результате в приложении одновременно использовались три языка: английский, выбранный язык приложения и язык версии Windows, в которой запускалось данное приложение. Поскольку имеющаяся версия Visual Studio.NET работает только в английской версии Windows 2000 и позволяет выбрать только английский язык приложения, то набор возможных вариантов сокращается до одного.

Выбор языка ресурсов в мастере MFC Application Wizard (Мастер создания приложений MFC) и предлагаемые ресурсы для создания приложений на японском, корейском и китайском языках влияют только на язык, на котором будет выводиться некоторый ограниченный набор системных сообщений. Остальные системные сообщения, независимо ни от чего, будут выводиться на английском языке или на языке операционной системы. Следует помнить, что пользователь может самостоятельно выбирать язык для создаваемых им ресурсов из практически неограниченного списка языков.

Для профессиональных программистов, имеющих опыт работы с продуктами фирмы Microsoft, одновременный вывод в одном окне сообщений на разных языках является привычным и они не обращают на это внимание, тем более что в качестве языка приложения они обычно выбирают английский, и, таким образом, приложение использует только два языка. Однако у неподготовленных пользователей подобная ситуация может вызывать нарекания. Поэтому желательно использовать в приложении только один язык.

Конечно, лучше всего было бы выбрать в качестве языка ресурсов английский язык. Это сразу бы сняло вопрос о появлении сообщений на других языках (если, конечно, работать в нелокализованной версии операционной системы, которая, как правило, работает надежнее и имеет расширенный набор возможностей). Однако многие пользователи с этим не согласны и считают, что вся информация в приложении должна выводиться на русском языке.

В этом случае возникает одно маленькое затруднение: корпорация Microsoft с такой скоростью "печет" новые операционные системы, что ей некогда заниматься их полноценной локализацией. В результате диалоговое окно вашего локализованного приложения будет иметь вид, подобный окну, изображенному на рис. 13.14.

Если вы думаете, что это касается только продуктов, создаваемых независимыми программистами, посмотрите на диалоговое окно **Параметры** приложения MS Word 7.0 в операционной системе Windows 2000 (рис. 13.15). Самое интересное, что это и многие другие диалоговые окна данного приложения выглядят подобным образом не только в английской, но и в русской версии Windows 2000.

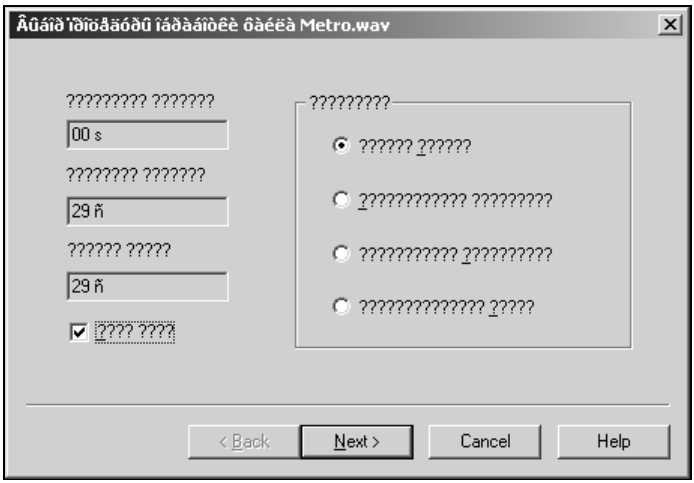


Рис. 13.14. Диалоговое окно русифицированного приложения в Windows 2000

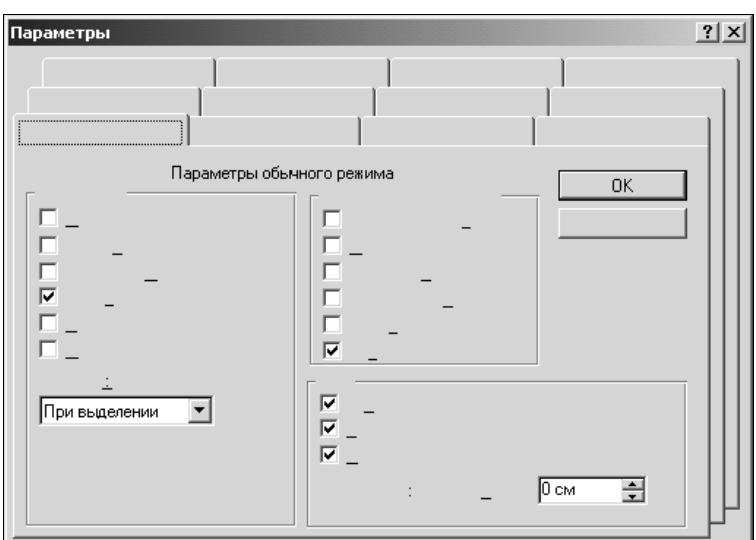


Рис. 13.15. Диалоговое окно **Параметры** приложения MS Word 7.0 в Windows 2000

Если вас не пугают проблемы с поддержкой требуемых шрифтов, вы можете смело приступать к русификации своего приложения. Если предположить, что создаваемое приложение будет работать под управлением русской версии Windows, то задачей программиста является исключение из него английских текстов. Часть этих текстов содержится в файле ресурсов приложения, другие

же содержатся в системных файлах ресурсов, которые должны включаться, но не включаются в файл ресурсов приложения (то есть оператор включения данных файлов присутствует, а результаты — отсутствуют).

Изменения в файл ресурсов приложения могут быть внесены по отдельности для каждого из ресурсов с использованием редактора соответствующего ресурса. Однако все эти изменения проще произвести непосредственно в файле ресурсов.

Файл ресурсов многооконного приложения, позволяющего печатать документы и не работающего с базами данных и компонентами OLE, должен быть включен в файлы afxres.rc и afxprint.rc, содержащие строковые ресурсы и описание диалоговых окон на выбранном языке приложения. Первый из них содержит заголовки и тексты для стандартных диалоговых окон открытия и закрытия файлов, всевозможные сообщения об ошибках, описание диалогового окна создания нового документа и описание некоторых ресурсов, не связанных с локализацией (курсоров и битовых образов).

Как выяснилось, независимо от применяемого системного файла ресурсов, приложение использует для вывода сообщений об ошибках ресурсы операционной системы, и, в случае русской версии Windows, в них нет необходимости вносить какие-либо изменения.

Второй файл (afxprint.rc) содержит описание окон печати и предварительного просмотра печати, строковые ресурсы для данных окон и таблицу акселераторов.

Чтобы приложение использовало ресурс, содержащийся в системном файле ресурсов, этот ресурс должен быть включен непосредственно в файл ресурса приложения. Кроме того, в сам файл ресурса приложения необходимо внести некоторые изменения, связанные с его локализацией. Ниже будет приведен локализованный файл ресурса приложения Help, который можно с минимальными изменениями использовать для локализации любого многооконного приложения, позволяющего распечатывать документы и не работающего с базами данных и объектами OLE.

Прежде, чем приступить к внесению изменений в файл приложения, необходимо создать приложение. Текст демонстрационного приложения Help, имеющего справочную систему, содержится в одноименной папке на дискете, прилагаемой к данной книге. Чтобы самостоятельно создать это приложение:

1. Выполните все операции, указанные в *главе 1* для создания многооконного приложения с именем Help, но в диалоговом окне **MFC Application Wizard** (Мастер создания приложений MFC) не нажимайте кнопку **Finish** (Готово).
2. Вместо этого раскройте в нем вкладку **Advanced Features** (Дополнительные возможности), установите в ней флажок **Context-sensitive Help** (Контекстная справка) и только после этого нажмите кнопку **Finish** (Готово).
3. Откройте файл ресурса данного приложения для редактирования в текстовом режиме. Для чего выберите команду **File | Open | File** (Файл | Открыть | Файл) или нажмите соответствующую кнопку на панели инструментов. Появится диалоговое окно **Open** (Открыть).

4. В окне списка выделите имя файла Help.rc, раскройте список кнопки **Open** (Открыть), как это показано на рис. 13.16, и выделите в нем команду **Open With** (Открыть с помощью). Появится диалоговое окно **Open With** (Открыть с помощью), изображенное на рис. 13.17.

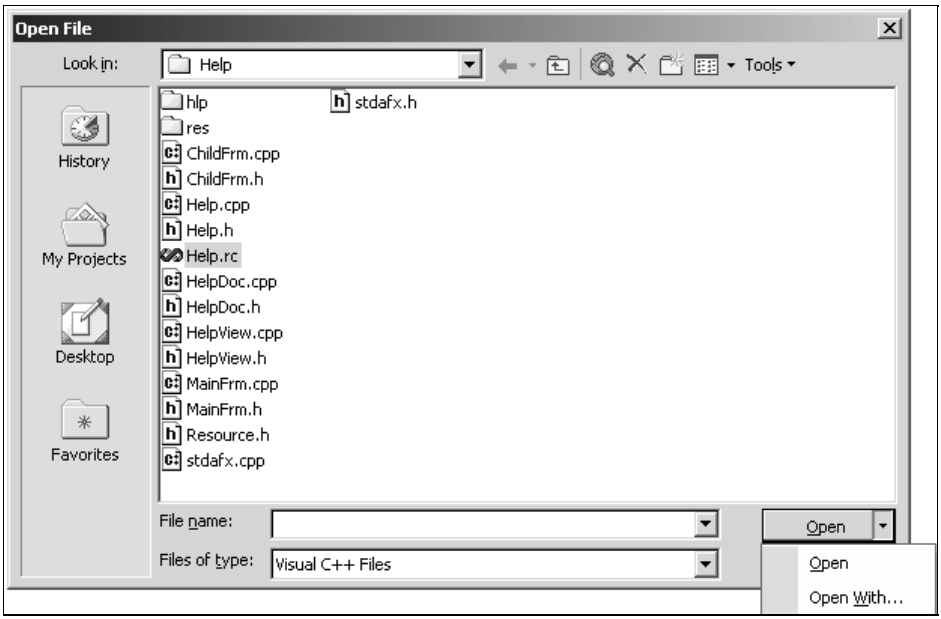


Рис. 13.16. Диалоговое окно **Open**

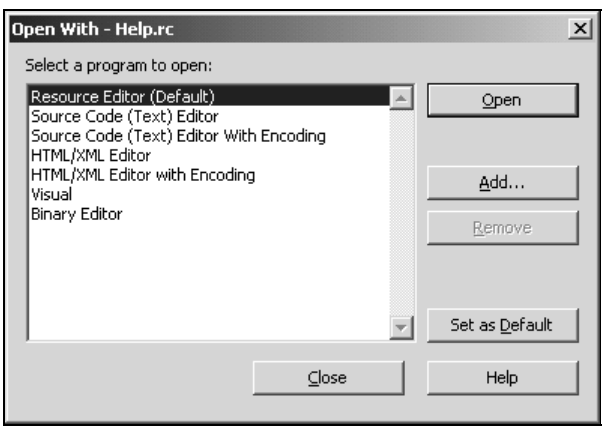


Рис. 13.17. Диалоговое окно **Open With - Help.rc**

5. В окне списка **Select a program to open** (Выделите программу, используемую для открытия файла) выделите строку **Source Code (Text) Editor** (Текстовый редактор исходного кода) и нажмите кнопку **Open** (Открыть). Откроется окно редактирования файла ресурсов в текстовом виде.
6. Измените текст открывшегося файла ресурсов в соответствии с листингом 13.1.

Листинг 13.1. Русифицированный файл ресурсов

```
// Файл ресурсов, созданный Microsoft Developer Studio.
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS
/////////////////////////////////////////////////////////////////////////////
// Создан из ресурса TEXTINCLUDE 2.

#include "afxres.h"
/////////////////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

#ifdef APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////////////////////
// TEXTINCLUDE

1 TEXTINCLUDE
BEGIN
  "resource.h\0"
END

2 TEXTINCLUDE
BEGIN
  "#include ""afxres.h""\r\n"
  "\0"
END

3 TEXTINCLUDE
BEGIN
    "#define _AFX_NO_SPLITTER_RESOURCES\r\n"
    "#define _AFX_NO_OLE_RESOURCES\r\n"
    "#define _AFX_NO_TRACKER_RESOURCES\r\n"
```

```
    "#define _AFX_NO_PROPERTY_RESOURCES\r\n"
    "\r\n"
    "#if !defined(AFX_RESOURCE_DLL)  |  |  defined(AFX_TARG_ENU)\r\n"
    "#ifdef _WIN32\r\n"
    "LANGUAGE LANG_RUSSIAN, SUBLANG_DEFAULT\r\n"
    "#pragma code_page(1251)\r\n"
    "#endif //_WIN32\r\n"
    "#include ""res\\Help.rc2""  // non-Microsoft Visual C++ edited    re-
sources\r\n"
    "#endif\r\n"
    "\0"
END

#endif    // APSTUDIO_INVOKED

/////////////////////////////////////////////////////////////////////////////
// Icon

// Значок, имеющий наименьшее значение идентификатора, располагается
// первым, чтобы обеспечить использование значка приложения
// во всех системах

#if !defined(AFX_RESOURCE_DLL)  |  |  defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_RUSSIAN, SUBLANG_DEFAULT
#pragma code_page(1251)
#endif //_WIN32
IDR_MAINFRAME           ICON         "res\\Help.ico"
IDR_HelpTYPE            ICON         "res\\HelpDoc.ico"
#endif
/////////////////////////////////////////////////////////////////////////////
// Bitmap

IDR_MAINFRAME           BITMAP      "res\\Toolbar.bmp"

/////////////////////////////////////////////////////////////////////////////
// CMiniFrameWnd Bitmap

AFX_IDB_MINIFRAME_MENU      BITMAP  DISCARDABLE     "res\\minifwnd.bmp"
```

```
/////////////////////////////////////////////////////////////////////////////
// CCheckListBox Bitmaps

AFX_IDB_CHECKLISTBOX_95     BITMAP  DISCARDABLE     "res\\95check.bmp"

/////////////////////////////////////////////////////////////////////////////

/////////////////////////////////////////////////////////////////////////////
// Cursors

AFX_IDC_CONTEXTHELP     CURSOR  DISCARDABLE     "res\\help.cur"

#ifndef _AFX_NO_SPLITTER_RESOURCES
AFX_IDC_SMALLARROWS     CURSOR  DISCARDABLE     "res\\sarrows.cur"
AFX_IDC_HSPLITBAR       CURSOR  DISCARDABLE     "res\\splith.cur"
AFX_IDC_VSPLITBAR       CURSOR  DISCARDABLE     "res\\splitv.cur"
AFX_IDC_NODROPCRSR      CURSOR  DISCARDABLE     "res\\nodrop.cur"
#endif //!_AFX_NO_SPLITTER_RESOURCES

#ifndef _AFX_NO_TRACKER_RESOURCES
AFX_IDC_TRACKNWSE       CURSOR  DISCARDABLE     "res\\trcknwse.cur"
AFX_IDC_TRACKNESW       CURSOR  DISCARDABLE     "res\\trcknesw.cur"
AFX_IDC_TRACKNS         CURSOR  DISCARDABLE     "res\\trckns.cur"
AFX_IDC_TRACKWE         CURSOR  DISCARDABLE     "res\\trckwe.cur"
AFX_IDC_TRACK4WAY       CURSOR  DISCARDABLE     "res\\trck4way.cur"
AFX_IDC_MOVE4WAY        CURSOR  DISCARDABLE     "res\\move4way.cur"
#endif //!_AFX_NO_TRACKER_RESOURCES

#ifndef _AFX_NO_MOUSE_RESOURCES

AFX_IDC_MOUSE_PAN_N  CURSOR  DISCARDABLE  "res\\im_pann.cur"
AFX_IDC_MOUSE_PAN_S  CURSOR  DISCARDABLE  "res\\im_pans.cur"
AFX_IDC_MOUSE_PAN_E  CURSOR  DISCARDABLE  "res\\im_pane.cur"
AFX_IDC_MOUSE_PAN_W  CURSOR  DISCARDABLE  "res\\im_panw.cur"
AFX_IDC_MOUSE_PAN_NE  CURSOR  DISCARDABLE  "res\\im_panne.cur"
AFX_IDC_MOUSE_PAN_NW  CURSOR  DISCARDABLE  "res\\im_pannw.cur"
AFX_IDC_MOUSE_PAN_SE  CURSOR  DISCARDABLE  "res\\im_panse.cur"
AFX_IDC_MOUSE_PAN_SW  CURSOR  DISCARDABLE  "res\\im_pansw.cur"
AFX_IDC_MOUSE_PAN_HORZ  CURSOR  DISCARDABLE  "res\\im_panh.cur"
AFX_IDC_MOUSE_PAN_VERT  CURSOR  DISCARDABLE  "res\\im_panv.cur"
AFX_IDC_MOUSE_PAN_HV  CURSOR  DISCARDABLE  "res\\im_panhv.cur"
```

```
// anchor bitmaps
AFX_IDC_MOUSE_ORG_HORZ  CURSOR  DISCARDABLE  "res\\im_orgh.cur"
AFX_IDC_MOUSE_ORG_VERT  CURSOR  DISCARDABLE  "res\\im_orgv.cur"
AFX_IDC_MOUSE_ORG_HV  CURSOR  DISCARDABLE  "res\\im_orghv.cur"
#endif

AFX_IDC_MAGNIFY         CURSOR  DISCARDABLE     "res\\magnify.cur"

/////////////////////////////////////////////////////////////////////////////
// Toolbar

IDR_MAINFRAME TOOLBAR   16, 15
BEGIN
    BUTTON      ID_FILE_NEW
    BUTTON      ID_FILE_OPEN
    BUTTON      ID_FILE_SAVE
  SEPARATOR
    BUTTON      ID_EDIT_CUT
    BUTTON      ID_EDIT_COPY
    BUTTON      ID_EDIT_PASTE
    SEPARATOR
    BUTTON      ID_FILE_PRINT
    BUTTON      ID_APP_ABOUT
    BUTTON      ID_CONTEXT_HELP
END

#if !defined(AFX_RESOURCE_DLL)  |  |  defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_RUSSIAN, SUBLANG_DEFAULT
#pragma code_page(1251)
#endif //_WIN32
/////////////////////////////////////////////////////////////////////////////
// Menu

IDR_MAINFRAME MENU
BEGIN
    POPUP "&Файл"
    BEGIN
        MENUITEM "Созд&ать...\tCtrl+N",         ID_FILE_NEW
        MENUITEM "&Открыть...\tCtrl+O",         ID_FILE_OPEN
```

```
        MENUITEM SEPARATOR
        MENUITEM "&Настройка принтера...",      ID_FILE_PRINT_SETUP
        MENUITEM SEPARATOR
        MENUITEM "&Текущий документ",           ID_FILE_MRU_FILE1, GRAYED
        MENUITEM SEPARATOR
        MENUITEM "&Закрыть",                    ID_FILE_CLOSE
        MENUITEM "В&ыход",                      ID_APP_EXIT
    END
    POPUP "&Вид"
    BEGIN
        MENUITEM "Панель &инструментов",        ID_VIEW_TOOLBAR
        MENUITEM "Панель &состояния",           ID_VIEW_STATUS_BAR
    END
    POPUP "&?"
    BEGIN
        MENUITEM "&Вызов справки",              ID_HELP_FINDER
        MENUITEM SEPARATOR
        MENUITEM "&О программе",                ID_APP_ABOUT
    END
END

IDR_HelpTYPE MENU
BEGIN
    POPUP "&Файл"
    BEGIN
        MENUITEM "Созд&ать...\tCtrl+N",         ID_FILE_NEW
        MENUITEM "&Открыть...\tCtrl+O",         ID_FILE_OPEN
        MENUITEM "&Закрыть",                    ID_FILE_CLOSE
        MENUITEM "&Сохранить\tCtrl+S",          ID_FILE_SAVE
        MENUITEM "Сохранить &как...",           ID_FILE_SAVE_AS
        MENUITEM SEPARATOR
        MENUITEM "&Печать...\tCtrl+P",          ID_FILE_PRINT
        MENUITEM "Пред&варительный просмотр",   ID_FILE_PRINT_PREVIEW
        MENUITEM "&Настройка принтера...",      ID_FILE_PRINT_SETUP
        MENUITEM SEPARATOR
        MENUITEM "&Текущий документ",           ID_FILE_MRU_FILE1, GRAYED
        MENUITEM SEPARATOR
        MENUITEM "В&ыход",                      ID_APP_EXIT
    END
    POPUP "&Правка"
```

```
    BEGIN
        MENUITEM "&Отменить\tCtrl+Z",           ID_EDIT_UNDO
        MENUITEM SEPARATOR
        MENUITEM "&Вырезать\tCtrl+X",           ID_EDIT_CUT
        MENUITEM "&Копировать\tCtrl+C",         ID_EDIT_COPY
        MENUITEM "Вст&авить\tCtrl+V",           ID_EDIT_PASTE
    END
    POPUP "&Вид"
    BEGIN
        MENUITEM "Панель &инструментов",        ID_VIEW_TOOLBAR
        MENUITEM "Панель &состояния",           ID_VIEW_STATUS_BAR
    END
    POPUP "&Окно"
    BEGIN
        MENUITEM "&Новое",                      ID_WINDOW_NEW
        MENUITEM "&Каскадировать",              ID_WINDOW_CASCADE
        MENUITEM "&Разделить",                  ID_WINDOW_TILE_HORZ
        MENUITEM "&Упорядочить",                ID_WINDOW_ARRANGE
    END
    POPUP "&?"
    BEGIN
        MENUITEM "&Вызов справки",              ID_HELP_FINDER
        MENUITEM SEPARATOR
        MENUITEM "&О программе",                ID_APP_ABOUT
    END
END

/////////////////////////////////////////////////////////////////////////////
// Accelerator

IDR_MAINFRAME ACCELERATORS
BEGIN
  "N",            ID_FILE_NEW,            VIRTKEY,CONTROL
  "O",            ID_FILE_OPEN,           VIRTKEY,CONTROL
  "S",            ID_FILE_SAVE,           VIRTKEY,CONTROL
  "P",            ID_FILE_PRINT,          VIRTKEY,CONTROL
  "Z",            ID_EDIT_UNDO,           VIRTKEY,CONTROL
  "X",            ID_EDIT_CUT,            VIRTKEY,CONTROL
  "C",            ID_EDIT_COPY,           VIRTKEY,CONTROL
  "V",            ID_EDIT_PASTE,          VIRTKEY,CONTROL
```

```
  VK_BACK,        ID_EDIT_UNDO,           VIRTKEY,ALT
  VK_DELETE,      ID_EDIT_CUT,            VIRTKEY,SHIFT
  VK_INSERT,      ID_EDIT_COPY,           VIRTKEY,CONTROL
  VK_INSERT,      ID_EDIT_PASTE,          VIRTKEY,SHIFT
  VK_F6,          ID_NEXT_PANE,           VIRTKEY

  VK_F6,          ID_PREV_PANE,           VIRTKEY,SHIFT
  VK_F1,          ID_CONTEXT_HELP,        VIRTKEY,SHIFT
  VK_F1,          ID_HELP,                VIRTKEY
END

// Таблица акселераторов предварительного просмотра печати
AFX_IDR_PREVIEW_ACCEL ACCELERATORS LOADONCALL MOVEABLE
BEGIN
  VK_NEXT,    AFX_ID_PREVIEW_NEXT,    VIRTKEY, NOINVERT
  VK_PRIOR,   AFX_ID_PREVIEW_PREV,    VIRTKEY, NOINVERT
  VK_ESCAPE,  AFX_ID_PREVIEW_CLOSE,   VIRTKEY, NOINVERT
END

/////////////////////////////////////////////////////////////////////////////
// Dialog

IDD_ABOUTBOX DIALOG   0, 0, 235, 55
CAPTION "О программе"
STYLE DS_MODALFRAME  |  WS_POPUP  |  WS_CAPTION  |  WS_SYSMENU
FONT 8, "MS Sans Serif"
BEGIN
  ICON            IDR_MAINFRAME,IDC_STATIC,11,17,20,20
  LTEXT           "Help версия 1.0",IDC_STATIC,40,10,119,8,
                   SS_NOPREFIX
  LTEXT           "Copyright (C) 2001",IDC_STATIC,40,25,119,8
  DEFPUSHBUTTON   "OK",IDOK,178,7,50,14,WS_GROUP
END

/////////////////////////////////////////////////////////////////////////////
// Standard Dialogs

#ifndef _AFX_NO_NEWTYPEDLG_RESOURCES
AFX_IDD_NEWTYPEDLG DIALOG DISCARDABLE  9, 26, 183, 70
STYLE DS_MODALFRAME  |  WS_POPUP  |  WS_CAPTION  |  WS_SYSMENU  |  0x04
```

```
CAPTION "New"
FONT 8, "MS Shell Dlg"
BEGIN
  LTEXT           "&Тип создаваемого документа",IDC_STATIC,6,5,123,8,NOT
WS_GROUP
  LISTBOX         AFX_IDC_LISTBOX,6,15,125,49,WS_VSCROLL  |  WS_TABSTOP
  DEFPUSHBUTTON   "OK",IDOK,137,6,40,14
  PUSHBUTTON      "&Отмена",IDCANCEL,137,23,40,14
  PUSHBUTTON      "&Справка ...",ID_HELP,137,43,40,14
END
#endif // !_AFX_NO_NEWTYPEDLG_RESOURCES

AFX_IDD_PRINTDLG DIALOG DISCARDABLE  6, 18, 133, 95
STYLE DS_MODALFRAME  |  WS_POPUP  |  WS_CAPTION  |  0x04
FONT 8, "MS Shell Dlg"
BEGIN
  CTEXT           "Печать",IDC_STATIC,0,9,133,8
  CTEXT           "на",IDC_STATIC,0,19,133,8
  CTEXT           "",AFX_IDC_PRINT_PRINTERNAME,0,29,133,8
  CTEXT           "",AFX_IDC_PRINT_PORTNAME,0,39,133,8
  CTEXT           "",AFX_IDC_PRINT_DOCNAME,0,49,133,8
  CTEXT           "",AFX_IDC_PRINT_PAGENUM,0,59,133,8
  PUSHBUTTON      "Отмена",IDCANCEL,46,74,40,14
END

// Диалоговое окно панели инструментов предварительного просмотра печати
AFX_IDD_PREVIEW_TOOLBAR DIALOG PRELOAD DISCARDABLE  0, 0, 330, 16
STYLE WS_CHILD  |  0x04
FONT 8, "MS Shell Dlg"
BEGIN
  PUSHBUTTON      "&Печать...",AFX_ID_PREVIEW_PRINT,2,2,44,12
  PUSHBUTTON      "С&ледующая",AFX_ID_PREVIEW_NEXT,48,2,44,12
  PUSHBUTTON      "Пр&едыдущая",AFX_ID_PREVIEW_PREV,94,2,44,12
  PUSHBUTTON      "",AFX_ID_PREVIEW_NUMPAGE,140,2,44,12
  PUSHBUTTON      "&Крупнее",AFX_ID_PREVIEW_ZOOMIN,186,2,44,12
  PUSHBUTTON      "&Мельче",AFX_ID_PREVIEW_ZOOMOUT,232,2,44,12
  PUSHBUTTON      "&Закрыть",AFX_ID_PREVIEW_CLOSE,278,2,44,12
END

/////////////////////////////////////////////////////////////////////////////
//
```

```
// Version
//
VS_VERSION_INFO     VERSIONINFO
  FILEVERSION       1,0,0,1
  PRODUCTVERSION    1,0,0,1
 FILEFLAGSMASK 0x3fL
#ifdef _DEBUG
 FILEFLAGS 0x1L
#else
 FILEFLAGS 0x0L
#endif
 FILEOS 0x4L
 FILETYPE 0x1L
 FILESUBTYPE 0x0L
BEGIN
  BLOCK "StringFileInfo"
  BEGIN
    BLOCK "040904B0"
    BEGIN

      VALUE "CompanyName",     "\0"
      VALUE "FileDescription", "Help Application\0"
      VALUE "FileVersion",     "1.0.0.1\0"
      VALUE "InternalName",    "Help\0"
      VALUE "LegalCopyright",  "Copyright (C) 2001\0"
      VALUE "LegalTrademarks", "\0"
      VALUE "OriginalFilename","Help.EXE\0"
      VALUE "ProductName",     "Help Application\0"
      VALUE "ProductVersion",  "1.0.0.1\0"
    END
  END
  BLOCK "VarFileInfo"

  BEGIN
    VALUE "Translation", 0x409, 1200
  END
END

/////////////////////////////////////////////////////////////////////////////
// DESIGNINFO
```

```
//
#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO

BEGIN
  IDD_ABOUTBOX, DIALOG
  BEGIN
    LEFTMARGIN, 7
    RIGHTMARGIN, 228
    TOPMARGIN, 7
    BOTTOMMARGIN, 48
  END
END
#endif    // APSTUDIO_INVOKED

/////////////////////////////////////////////////////////////////////////////
// String Table
//
STRINGTABLE DISCARDABLE
BEGIN
#ifndef _AFX_NO_APPMENU_RESOURCES
  AFX_IDS_OPENFILE        "Открытие документа"
  AFX_IDS_SAVEFILE        "Сохранение документа"
  AFX_IDS_ALLFILTER       "Все файлы (*.*)"
  AFX_IDS_UNTITLED        "Новый"
  AFX_IDS_HIDE,           "&Скрыть"
#endif // !_AFX_NO_APPMENUSTRING_RESOURCES
END
// Строковые ресурсы, используемые при печати
STRINGTABLE DISCARDABLE
BEGIN
  // используются в диалоговых окнах
  AFX_IDS_PRINTONPORT    "на %1"
  AFX_IDS_ONEPAGE      "&Одна стр." // для кнопки окна
                                    // предварительного просмотра
                                    // печати
  AFX_IDS_TWOPAGE      "&Две стр."  // для кнопки окна
                                    // предварительного просмотра
                                    // печати
  AFX_IDS_PRINTPAGENUM  "Стр. %u"
  AFX_IDS_PREVIEWPAGEDESC  "Страница %u\nСтраницы %u-%u\n"
```

```
  // используются при печати в файл
  AFX_IDS_PRINTDEFAULTEXT  "prn"      // расширение, используемое
                                      // по умолчанию
  AFX_IDS_PRINTDEFAULT  "Output.prn"  // имя файла, используемое
                                      // по умолчанию
  AFX_IDS_PRINTFILTER    "Файлы печати (*.prn) | *.prn | Все файлы (*.*) |
                         *.* |  | "
  AFX_IDS_PRINTCAPTION  "Печать в файл"
  AFX_IDS_PRINTTOFILE    "файл %1"
END
STRINGTABLE

BEGIN
//  Приложения, не ориентированные на MAC, удаляют следующие две строки
  IDR_MAINFRAME           "Help"
  IDR_HelpTYPE           "\nHelp\nHelp\n\n\nHelp.Document\nHelp.Document"
END
STRINGTABLE

BEGIN
  AFX_IDS_APP_TITLE       "Help"
    AFX_IDS_IDLEMESSAGE      "Для вывода справки нажмите клавишу F1"
    AFX_IDS_HELPMODEMESSAGE  "Выберите объект, по которому вы хотели бы
                              получить информацию"
END
STRINGTABLE

BEGIN
  ID_INDICATOR_EXT        "EXT"
  ID_INDICATOR_CAPS       "CAP"
  ID_INDICATOR_NUM        "NUM"
  ID_INDICATOR_SCRL       "SCRL"
  ID_INDICATOR_OVR        "OVR"
  ID_INDICATOR_REC        "REC"
END
STRINGTABLE

BEGIN
    ID_FILE_NEW             "Создание нового окна\nСоздать"
    ID_FILE_OPEN            "Открытие существующего документа\nОткрыть"
    ID_FILE_CLOSE           "Закрытие активного окна\nЗакрыть"
```

```
ID_FILE_SAVE            "Сохранение активного документа\nСохранить"
ID_FILE_SAVE_AS         "Сохранение активного документа под другим
                         именем\nСохранить как"
ID_FILE_PAGE_SETUP      "Изменение параметров страницы\nПараметры
                         страницы"
ID_FILE_PRINT_SETUP     "Смена принтера и измение его
                         параметров\nНастройка принтера"
ID_FILE_PRINT           "Печать активного документа\nПечать"
ID_FILE_PRINT_PREVIEW   "Просмотр печатаемого материала в полностраничном
                         режиме\nПредварительный просмотр"


ID_APP_ABOUT            "Вывод сведений о программе, ее версии
                         и авторских правах\nО программе"
ID_APP_EXIT             "Завершение работы с приложением; сохранение
                         документов\nВыход"
ID_CONTEXT_HELP         "Выводит информацию о кнопках, командах меню
                         и окнах\nСправка"
ID_HELP_INDEX           "Вывод содержимого файловой ситемы\nВывод
                         справки"
ID_HELP_FINDER          "Поиск по образцу\nВывод справки"
ID_HELP_USING           "Выводит руководство по пользованию справочной
                         системой\nСправка"
ID_HELP                 "Выводит справку по текущей задаче
                         или команде\nСправка"
ID_FILE_MRU_FILE1       "Открыть этот документ"
ID_FILE_MRU_FILE2       "Открыть этот документ"
ID_FILE_MRU_FILE3       "Открыть этот документ"
ID_FILE_MRU_FILE4       "Открыть этот документ"
ID_FILE_MRU_FILE5       "Открыть этот документ"
ID_FILE_MRU_FILE6       "Открыть этот документ"
ID_FILE_MRU_FILE7       "Открыть этот документ"
ID_FILE_MRU_FILE8       "Открыть этот документ"
ID_FILE_MRU_FILE9       "Открыть этот документ"
ID_FILE_MRU_FILE10      "Открыть этот документ"
ID_FILE_MRU_FILE11      "Открыть этот документ"
ID_FILE_MRU_FILE12      "Открыть этот документ"
ID_FILE_MRU_FILE13      "Открыть этот документ"
ID_FILE_MRU_FILE14      "Открыть этот документ"
ID_FILE_MRU_FILE15      "Открыть этот документ"
ID_FILE_MRU_FILE16      "Открыть этот документ"
ID_NEXT_PANE            "Переход к следующей панели окна\nСледующая
                         панель"
ID_PREV_PANE            "Переход к предыдущей панели окна\nПредыдущая
                         панель"
```

```
    ID_WINDOW_NEW           "Открыть другое окно\nНовое окно"
    ID_WINDOW_ARRANGE       "Упорядочить значки в нижней части
                             окна\nУпорядочить значки"
    ID_WINDOW_CASCADE       "Упорядочить окна таким образом, чтобы они
                             перекрывались\nКаскадировать окна"
    ID_WINDOW_TILE_HORZ     "Упорядочить окна по горизонтали без
                             перекрытия\nУпорядочить по горизонтали"
    ID_WINDOW_TILE_VERT     "Упорядочить окна по горизонтали без
                             перекрытия\nУпорядочить по горизонтали"
    ID_WINDOW_SPLIT         "Разбить активное окно на панели\nРазбить"


    ID_EDIT_CLEAR           "Удаление выделенного фрагмента\nУдалить"
    ID_EDIT_CLEAR_ALL       "Удалить все\nУдалить все"
    ID_EDIT_COPY            "Копирование выделенного фрагмента в буфер
                             обмена\nКопировать в буфер"
    ID_EDIT_CUT             "Удаление выделенного фрагмента в буфер
                             обмена\nУдалить в буфер"
    ID_EDIT_FIND            "Поиск заданного текста\nНайти"
    ID_EDIT_PASTE           "Вставка содержимого буфера обмена\nВставить"
    ID_EDIT_REPEAT          "Повторить последнее действие\nПовторить"
    ID_EDIT_REPLACE         "Замена одного текста другим\nЗаменить"
    ID_EDIT_SELECT_ALL      "Выделение всего документа целиком\nВыделить
                             все"
    ID_EDIT_UNDO            "Отмена последнего выполненого
                             действия\nОтменить"
    ID_EDIT_REDO            "Вернуть результат последнего отмененного
                             действия\nВернуть"
    ID_VIEW_TOOLBAR         "Вывод и скрытие панели интструмеентов\nПанель
                             инструментов"
    ID_VIEW_STATUS_BAR      "Вывод и скрытие строки состояния\nСтрока
                             состояния"
END


STRINGTABLE


BEGIN
    AFX_IDS_SCSIZE          "Изменение размеров окна"
    AFX_IDS_SCMOVE          "Изменение положения окна"
    AFX_IDS_SCMINIMIZE      "Минимизация окна"
    AFX_IDS_SCMAXIMIZE      "Максимизация окна"
    AFX_IDS_SCNEXTWINDOW    "Переход к следующему окну"
    AFX_IDS_SCPREVWINDOW    "Переход к предыдущему окну"
    AFX_IDS_SCCLOSE         "Закрытие активного окна и сохранение документа"
```

```
    AFX_IDS_SCRESTORE       "Восстановление размеров окна"
    AFX_IDS_SCTASKLIST      "Активизация списка задач"
    AFX_IDS_MDICHILD        "Активизация данного окна"
    AFX_IDS_PREVIEW_CLOSE   "Выйти из режима предварительного просмотра
                             печати\nПрекратить просмотр"
END

#endif

#ifndef APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////////////////////
// Создан из ресурса TEXTINCLUDE 3

#define _AFX_NO_SPLITTER_RESOURCES
#define _AFX_NO_OLE_RESOURCES
#define _AFX_NO_TRACKER_RESOURCES
#define _AFX_NO_PROPERTY_RESOURCES

#if !defined(AFX_RESOURCE_DLL)  |  |  defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_RUSSIAN, SUBLANG_DEFAULT
#pragma code_page(1251)
#endif //_WIN32
#include "res\\Help.rc2"  // Ресурсы, не созданные Microsoft Visual C++
#endif
#endif    // not APSTUDIO_INVOKED
```

7. Скопируйте все файлы ресурсов из подкаталога /res папки, в котором расположены файлы afxrec.rc и afxprint.rc, в подкаталог /res папки приложения Help.

Необходимость копирования файлов ресурсов связана с тем, что в Visual C++ очень странно трактуется включение файлов. По определению директива `#include` просто включает текст указанного в ней файла в ту позицию исходного файла, в которой она расположена. Поэтому, если ту же самую операцию проделает пользователь, ничего не должно измениться. Однако при использовании директивы `#include` относительные пути определяются от местоположения включаемого файла, что приводит к необходимости использовать при пользовательском включении абсолютные пути к файлам ресурсов или переместить их в свой каталог. В данном случае был выбран второй путь.

В исходный файл были внесены следующие изменения:

- все комментарии, кроме удаляемых, были переведены на русский язык;

- в разделе `TEXTINCLUDE` текст

```
"LANGUAGE 9, 1\r\n"
"#pragma code_page(1252)\r\n"
"#endif //_WIN32\r\n"
"#include ""res\\Help.rc2""  // non-Microsoft Visual C++ edited
                                resources\r\n"
"#include ""afxres.rc""    // Standard components\r\n"
"#include ""afxprint.rc""  // printing/print preview resources\r\n"
```

  был заменен текстом

```
"LANGUAGE LANG_RUSSIAN, SUBLANG_DEFAULT\r\n"
"#pragma code_page(1251)\r\n"
"#endif //_WIN32\r\n"
"#include ""res\\Help.rc2""  // non-Microsoft Visual C++ edited
                                resources\r\n"
```

- в разделе `Icon` текст

```
LANGUAGE 9, 1
#pragma code_page(1252)
```

  был заменен текстом

```
LANGUAGE LANG_RUSSIAN, SUBLANG_DEFAULT
#pragma code_page(1251);
```

- после раздела `Bitmap` был вставлен следующий текст из файла afxres.rc

```
/////////////////////////////////////////////////////////////////////////////
// CMiniFrameWnd Bitmap

AFX_IDB_MINIFRAME_MENU      BITMAP  DISCARDABLE     "res\\minifwnd.bmp"

/////////////////////////////////////////////////////////////////////////////
// CCheckListBox Bitmaps

AFX_IDB_CHECKLISTBOX_95     BITMAP  DISCARDABLE     "res\\95check.bmp"
/////////////////////////////////////////////////////////////////////////////
```

- был создан раздел `Cursors` и в него была помещена информация о курсорах из файлов afxres.rc и afxprint.rc;
- расположенный перед разделом `Menu` текст

```
LANGUAGE 9, 1
#pragma code_page(1252)
```

  был заменен текстом

```
LANGUAGE LANG_RUSSIAN, SUBLANG_DEFAULT
#pragma code_page(1251)
```

- в разделе `Menu` были переведены на русский язык все команды меню;
- в раздел `Accelerator` из файла afxprint.rc была перенесена таблица акселераторов предварительного просмотра печати;
- после раздела `Accelerator` был включен раздел `Cursors`, составленный из соответствующего раздела файла afxres.rc и ресурса курсора из файла afxprint.rc;
- в разделе `Dialog` было русифицировано окно **О программе** и добавлены русифицированное диалоговое окно **Создание документа** из файла afxres.rc и окна **Предварительный просмотр** и **Печать** из файла afxprint.rc;
- в начало раздела `String Table` были помещены таблица строковых ресурсов из файла afxres.rc, содержащая ресурсы для создания диалоговых окон **Создание документа** и **Сохранение документа**, а также таблицы строковых ресурсов из файла afxprint.rc, содержащие ресурсы для работы с окнами **Предварительный просмотр** и **Печать**;
- все строковые ресурсы в разделе `String Table` были переведены на русский язык;
- в разделе `Создан из ресурса TEXTINCLUDE 3` текст

```
LANGUAGE 9, 1
#pragma code_page(1252)
#endif //_WIN32
#include "res\\Help.rc2"  // non-Microsoft Visual C++ edited resources
#include "afxres.rc"      // Standard components
#include "afxprint.rc"    // printing/print preview resources
```

  был заменен текстом

```
LANGUAGE LANG_RUSSIAN, SUBLANG_DEFAULT
#pragma code_page(1251)
#endif //_WIN32
#include "res\\Help.rc2"  // non Microsoft Visual C++ edited resources
```

Если предполагается применение приложения в версии Windows, использующей язык, отличный от русского, для русификации сообщений об ошибках в таблицу строковых ресурсов следует поместить следующий текст:

```
STRINGTABLE
BEGIN
    AFX_IDS_SCSIZE          "Изменение размеров окна"
    AFX_IDS_SCMOVE          "Изменение положения окна"
    AFX_IDS_SCMINIMIZE      "Минимизация окна"
    AFX_IDS_SCMAXIMIZE      "Максимизация окна"
    AFX_IDS_SCNEXTWINDOW    "Переход к следующему окну"
    AFX_IDS_SCPREVWINDOW    "Переход к предыдущему окну"
```

```
    AFX_IDS_SCCLOSE         "Закрытие активного окна и сохранение документа"
    AFX_IDS_SCRESTORE       "Восстановление размеров окна"
    AFX_IDS_SCTASKLIST      "Активизация списка задач"
    AFX_IDS_MDICHILD        "Активизация данного окна"
    AFX_IDS_PREVIEW_CLOSE   "Выйти из режима предварительного просмотра
                             печати\nПрекратить просмотр"
END

STRINGTABLE DISCARDABLE
BEGIN
    AFX_IDS_MEMORY_EXCEPTION         "Недостаточно памяти."
    AFX_IDS_NOT_SUPPORTED_EXCEPTION  "Данная операция не поддерживается
                                      системой."
    AFX_IDS_RESOURCE_EXCEPTION       "Требуемый ресурс недоступен."
    AFX_IDS_USER_EXCEPTION           "Возникла неизвестная ошибка."

  // Основные сообщения об ошибках
#ifndef _AFX_NO_FILECMD_RESOURCES
    AFX_IDP_INVALID_FILENAME      "Недопустимое имя файла."
    AFX_IDP_FAILED_TO_OPEN_DOC    "Ошибка при открытии документа."
    AFX_IDP_FAILED_TO_SAVE_DOC    "Ошибка при закрытии документа."
    AFX_IDP_ASK_TO_SAVE           "Сохранить изменения в %1?"
    AFX_IDP_FAILED_TO_CREATE_DOC      "Ошибка при создании пустого
                                       документа."
    AFX_IDP_FILE_TOO_LARGE            "Невозможно открыть файл такого
                                       размера."
    AFX_IDP_FAILED_TO_START_PRINT     "Невозможно инициировать процесс
                                       печати."
#endif // !_AFX_NO_FILECMD_RESOURCES
    AFX_IDP_FAILED_TO_LAUNCH_HELP     "Ошибка при обращении к справочной
                                       информации."
    AFX_IDP_INTERNAL_FAILURE          "Внутренняя ошибка приложения."
    AFX_IDP_COMMAND_FAILURE           "Команда не выполнена."
    AFX_IDP_FAILED_MEMORY_ALLOC       "Недостаточно памяти для данной
                                       операции."
    AFX_IDP_GET_NOT_SUPPORTED,        "Чтение невозможно. Файл открыт
                                       только для записи."
    AFX_IDP_SET_NOT_SUPPORTED,        "Запись невозможна. Файл открыт
                                       только для чтения."
    AFX_IDP_UNREG_DONE                "Записи удалены из системного
                                       реестра, а файл INI (если он
                                       существовал) уничтожен."
```

```
    AFX_IDP_UNREG_FAILURE             "Не все записи удалены из
                                       системного реестра (или файла
                                       INI file)."

    AFX_IDP_DLL_LOAD_FAILED           "Для работы данной программы
                                       требуется использование файла %s,
                                       который отсутствует в данной
                                       системе."

    AFX_IDP_DLL_BAD_VERSION           "Данная программа связана с
                                       отсутствующим объектом %s в файле
                                       %s. На данном компьютере
                                       установлена несовместимая версия
                                       %s."


  // Сообщения об ошибках CFile/CArchive, используемые пользователем
#ifndef _AFX_NO_CFILE_RESOURCES
    AFX_IDP_FAILED_INVALID_FORMAT  "Неизвестный формат файла."

    AFX_IDP_FAILED_INVALID_PATH    "%1\nДанный файл
                                    отсутствует.\nПроверьте правильность
                                    задания пути и имени файла."

    AFX_IDP_FAILED_DISK_FULL       "На указанном диске отсутствует
                                    необходимое свободное пространство."

    AFX_IDP_FAILED_ACCESS_READ     "Невозможно чтение файла %1, он открыт
                                    другим приложением."

    AFX_IDP_FAILED_ACCESS_WRITE    "Невозможна запись в файл %1, он
                                    открыт только для чтения или открыт
                                    другим приложением."

    AFX_IDP_FAILED_IO_ERROR_READ   "Неизвестная ошибка при чтении файла
                                   %1."

    AFX_IDP_FAILED_IO_ERROR_WRITE  "Неизвестная ошибка при записи файла
                                    %1."
#endif // !_AFX_NO_CFILE_RESOURCES


  // Сообщения об ошибках ввода
    AFX_IDP_PARSE_INT           "Введите целое число."

    AFX_IDP_PARSE_REAL          "Введите число."

    AFX_IDP_PARSE_INT_RANGE     "Введите целое число в диапазоне от %1 до
                                 %2."

    AFX_IDP_PARSE_REAL_RANGE    "Введите число в диапазоне от %1 до %2."

    AFX_IDP_PARSE_STRING_SIZE   "Введите не более  %1 символов."

    AFX_IDP_PARSE_RADIO_BUTTON  "Выберите кнопку."

    AFX_IDP_PARSE_BYTE          "Введите целое число в диапазоне от 0 до
                                 255."

    AFX_IDP_PARSE_UINT          "Введите допустимое целое число."

    AFX_IDP_PARSE_DATETIME      "Введите дату и/или время."

    AFX_IDP_PARSE_CURRENCY      "Введите величину в денежном формате."

    AFX_IDP_PARSE_GUID,         "Введите GUID."
```

```
    AFX_IDP_PARSE_TIME,         "Введите время."
    AFX_IDP_PARSE_DATE,         "Введите дату."

#ifndef _AFX_NO_OLE_RESOURCES
  // Строки, используемые в серверах и контейнерах OLE
    AFX_IDS_PASTELINKEDTYPE      "Связано с %s"
    AFX_IDS_UNKNOWNTYPE          "Неизвестный тип"
    AFX_IDP_FAILED_TO_NOTIFY     "%1\nНевозможно зарегистрировать
                                  документ.\nДокумент, возможно, уже
                                  открыт."
  AFX_IDS_NOT_DOCOBJECT          "Файл не может быть обработан сервером
                                  документов."
#endif //!_AFX_NO_OLE_RESOURCES

    AFX_IDP_NO_ERROR_AVAILABLE    "Сообщения об ошибках отсутствуют."

#ifndef _AFX_NO_CFILE_RESOURCES
    AFX_IDP_FILE_NONE           "Ошибки отсутствуют."
    AFX_IDP_FILE_GENERIC        "Возникла неизвестная ошибка при доступе
                                 к %1."
    AFX_IDP_FILE_NOT_FOUND      "Файл %1 не найден."
    AFX_IDP_FILE_BAD_PATH       "Файл %1 имеет некорректный путь."
    AFX_IDP_FILE_TOO_MANY_OPEN  "Файл %1 не может быть открыт, поскольку
                                 в системе открыто слишком много файлов."
    AFX_IDP_FILE_ACCESS_DENIED  "Отказ в доступе к файлу %1."
    AFX_IDP_FILE_INVALID_FILE   "Файл %1 имеет недопустимый дескриптор."
    AFX_IDP_FILE_REMOVE_CURRENT "Файл %1 не может быть удален, поскольку
                                 является текущим каталогом."
    AFX_IDP_FILE_DIR_FULL       "Файл %1 не может быть создан, поскольку
                                 текущий каталог переполнен."
    AFX_IDP_FILE_BAD_SEEK       "На %1 файл отсутствует"
    AFX_IDP_FILE_HARD_IO        "При доступе к файлу %1 возникла
                                 аппаратная ошибка ввода/вывода."
    AFX_IDP_FILE_SHARING        "При доступе к файлу %1 возникла ошибка
                                 при разделении ресурсов."
    AFX_IDP_FILE_LOCKING        "При доступе к файлу %1 произошло
                                 нарушение защиты."
    AFX_IDP_FILE_DISKFULL       "При доступе к файлу %1 произошло
                                 переполнение диска."
    AFX_IDP_FILE_EOF            "Произошла попытка доступа к записи в
                                 файле %1, расположенной после его
                                 конца."
    AFX_IDS_UNNAMED_FILE        "Неименованный файл"
#endif // !_AFX_NO_CFILE_RESOURCES
```

```
    AFX_IDP_ARCH_NONE        "Ошибки отсутствуют."
    AFX_IDP_ARCH_GENERIC     "Возникла неизвестная ошибка при доступе к
                              %1."
    AFX_IDP_ARCH_READONLY    "Произошла попытка записи в файл %1,
                              открытый только для чтения."
    AFX_IDP_ARCH_ENDOFFILE   "Произошла попытка доступа к записи в файле
                              %1, расположенной после его конца."
    AFX_IDP_ARCH_WRITEONLY   "Произошла попытка чтения из файла %1,
                              открытого только для записи."
    AFX_IDP_ARCH_BADINDEX    "Файл %1 имеет недопустимый формат."
    AFX_IDP_ARCH_BADCLASS    "Файл %1 содержит недопустимый объект."
    AFX_IDP_ARCH_BADSCHEMA   "Файл %1 содержит неправильную схему."

#ifndef _AFX_NO_MAPI_RESOURCES
  // Строки и сообщения MAPI
    AFX_IDP_FAILED_MAPI_LOAD    "Невозможно связаться системой поддержки
                                 электронной почты."
    AFX_IDP_INVALID_MAPI_DLL    "Некорректная версия системного DLL
                                 электронной почты."
    AFX_IDP_FAILED_MAPI_SEND    "Приложение Send Mail не смогло отправить
                                 сообщение."
#endif //!_AFX_NO_MAPI_RESOURCES

#ifndef _AFX_NO_OCC_RESOURCES
  AFX_IDS_OCC_SCALEUNITS_PIXELS  "Элементы изображения"
#endif //!_AFX_NO_OCC_RESOURCES

#ifndef _AFX_NO_SCRIPT_RESOURCES
    AFX_IDP_SCRIPT_ERROR      "%1: %2\nПродолжать выполнение сценария?"
    AFX_IDP_SCRIPT_DISPATCH_EXCEPTION "Диспетчировать исключение: %1"
#endif //!_AFX_NO_SCRIPT_RESOURCES
    AFX_IDS_CHECKLISTBOX_UNCHECK  "Не отмечен"
    AFX_IDS_CHECKLISTBOX_CHECK    "Отмечен"
    AFX_IDS_CHECKLISTBOX_MIXED    "Смешанный"
END
```

При переводе английских наименований команд меню и соответствующих им сообщений учитывалась сложившаяся в большинстве стандартных приложений практика, однако перевод сообщений об ошибках в большинстве случаев не проверялся, поэтому приведенные выше варианты перевода могут корректироваться.

Для полной русификации приложения необходимо русифицировать также и строки, включаемые в поле **Тип файлов** диалоговых окон **Открытие документа** и

Сохранение документа. Эти строки располагаются в четвертом поле текстового ресурса, соответствующего идентификатору шаблона документа. Если приложение Help будет обрабатывать текстовые документы, то соответствующий строковый ресурс будет иметь следующий вид:

```
IDR_HELPTYPE  "\nHelp\nHelp\nТекстовые файлы
(.txt)\n.txt\nHelp.Document\nHelp Document"
```

Однако даже полная русификация строковых ресурсов не гарантирует полной русификации приложения, поскольку разработчики библиотеки MFC, по крайней мере, в ее отладочной версии, в некоторых случаях выводят диагностические сообщения на английском языке, не прибегая к использованию таблицы строковых ресурсов. В этом можно убедиться, введя некорректное имя файла в диалоговом окне **Open** (Открыть).

Создание системы командной справки

Среда программирования Visual C++ произвела практически все необходимые действия для создания командной справки и, в большинстве случаев, этого достаточно. Но если пользователю нужно добавить новую команду в меню **?**, то все необходимые для этого действия он должен осуществить сам.

Чтобы внести новую команду в меню **?** приложения Help:

1. Выберите команду **File | Open Solution** (Файл | Открыть решение) и в открывшемся диалоговом окне **Open Solution** (Открыть решение) раскройте папку **Help** (Помощь).
2. В окне списка выделите значок **Help** (Помощь) и нажмите кнопку **Open** (Открыть).
3. Раскройте окно **Resource View - Help** (Просмотр ресурсов) окна **Workspace** (Рабочая область). В нем будет указано, что оно не может быть использовано, поскольку файл Help.rc редактируется другим редактором, как это показано на рис. 13.18.
4. Закройте окно редактирования файла Help.rc и щелкните левой кнопкой мыши на знаке "минус" у папки **Help.rc**. Предупреждающая надпись исчезнет и окно **Resource View - Help** (Просмотр ресурсов) будет готово к работе.
5. Раскройте папку **Help.rc**, а в ней — папку **Menu** (Меню) и дважды щелкните левой кнопкой мыши на значке `IDR_HelpTYPE`. В окне редактирования ресурса появится заготовка соответствующего меню.
6. Раскройте меню **?**. Щелкните левой кнопкой мыши на прямоугольнике с надписью "Type Here", расположенном в нижней части данного меню и введите в появившееся на его месте текстовое поле команду "&Новая справка" и нажмите клавишу <Enter>.
7. Поместите указатель мыши на только что введенную команду меню и перетащите ее вверх, разместив между командой меню **Вызов справки** и разделителем.

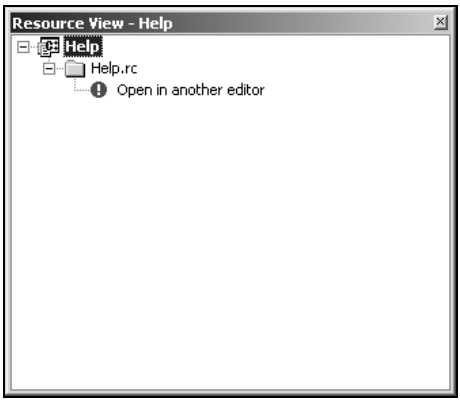


Рис. 13.18. Окно **Resource View - Help**

8. Отпустите левую кнопку мыши и щелкните по прямоугольнику правой кнопкой мыши.
9. В появившемся контекстном меню выберите команду **Properties** (Свойства). Откроется окно **Properties** (Свойства).
10. Введите в текстовое поле раскрывающегося списка **ID** (Идентификатор ресурса) идентификатор команды `ID_NEW_HELP` и нажмите кнопку **Save All** (Сохранить все) на панели инструментов **Standard** (Стандартная). Все изменения, внесенные в файл ресурса, будут сохранены.
11. В текстовое поле **Prompt** (Подсказка) введите текст "Новая справка\nНовая".
12. Откройте окно **Class View** (Просмотр класса), раскройте в нем папку **Help** (Помощь) и выделите в ней папку `CHelpView`.
13. В окне **Properties** (Свойства) нажмите кнопку **Events** (События). Раскроется список событий, обрабатываемых данным классом.
14. Раскройте в этом списке папку `ID_NEW_HELP`, выделите в ней строку `COMMAND` и введите в текстовое поле раскрывающегося списка имя функции обработки данного сообщения **OnNewHelp**.
15. В окне списка **Class View** (Просмотр классов) раскройте папку `CHelpView` и дважды щелкните левой кнопкой мыши на имени функции обработки сообщения `OnNewHelp`. Откроется окно редактирования файла HelpView.cpp и текстовый курсор установится в тексте функции `OnNewHelp`.
16. Измените функцию `OnNewHelp` в соответствии с текстом листинга 13.2.

Листинг 13.2. Функция `CHelpView::OnNewHelp`

```
// Функции обработки сообщений класса CHelpView

// Вызов новой справки
void CHelpView::OnNewHelp(void)
```

```
{
  WinHelp(HID_NEW_HELP);
}
```

17. Откройте окно редактирования файла HelpView.h и после оператора `#pragma once` вставьте строку

    ```
    #define  HID_NEW_HELP  0x01
    ```

18. Выберите команду **File | New | File.** (Файл | Создать | Файл). Раскроется диалоговое окно **New File** (Создать файл), изображенное на рис. 13.19.

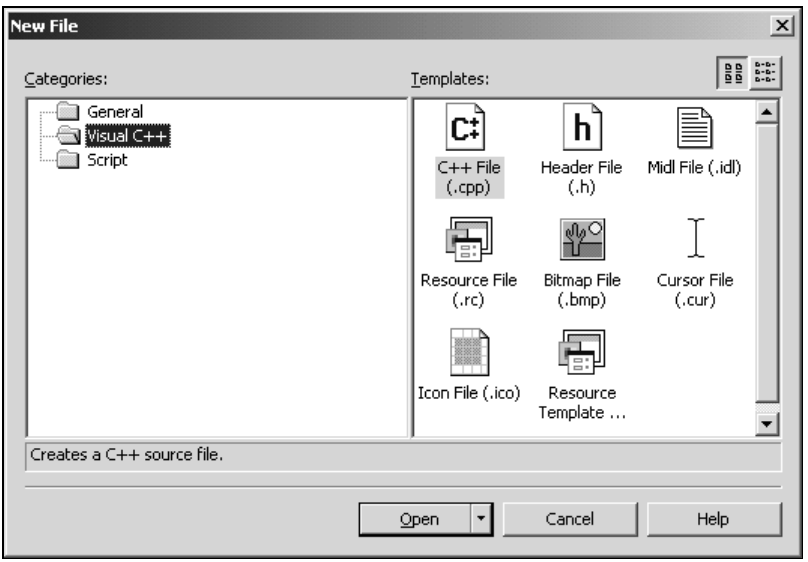


Рис. 13.19. Диалоговое окно **New File**

19. Раскройте папку **General** (Файлы общих типов), в окне списка **Templates** (Шаблоны) выделите значок **Text File** (Текстовый файл) и нажмите кнопку **Open** (Открыть). В окне редактирования появится новый чистый текстовый файл.

20. Введите в него строку

    ```
    HID_NEW_HELP    0x01
    ```

21. Закройте этот файл. Появится диалоговое окно **Microsoft Development Environment**, изображенное на рис. 13.20 и запрашивающее информацию о необходимости сохранить изменения в этом файле.

22. Нажмите кнопку **Yes** (Да). Появится диалоговое окно **Save File As** (Сохранить файл как), изображенное на рис. 13.21.

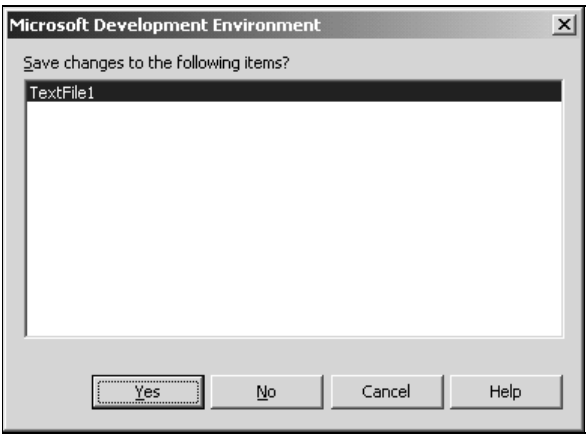


Рис. 13.20. Диалоговое окно **Microsoft Development Environment**

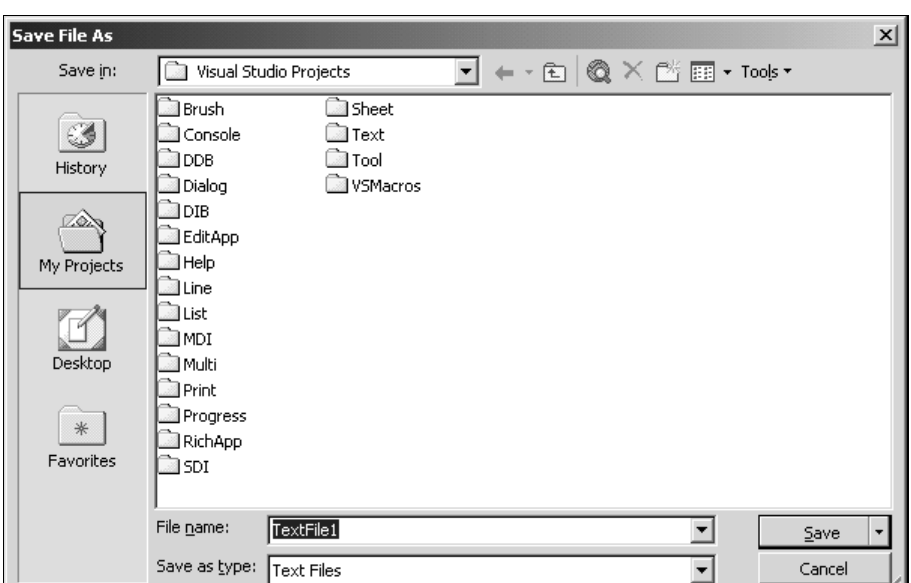


Рис. 13.21. Диалоговое окно **Save File As**

23. Раскройте папку **/Help**, а в ней — папку **/hlp**, в раскрывающемся списке **Save as type** (Тип сохраняемого файла) выделите строку **All Files** (Все файлы), в текстовое поле раскрывающегося списка **File name** (имя файла) введите имя файла **NewHelp.hm** и нажмите кнопку **Save** (Сохранить).

24. Откройте окно **Solution Explorer** (Проводник решения), раскройте в нем папку **Source Files** (Файлы реализации) и дважды щелкните левой кнопкой мыши на имени файла Help.hpj. Откроется окно редактирования этого файла.
25. В конец данного файла вставьте строку

```
#include <NewHelp.hm>
```

Операции по созданию команды меню и функции ее обработки ничем не отличаются от подобных операций для других команд меню. В функции обработки сообщения `ID_NEW_HELP` с использованием функции `CWinApp::WinHelp` вызывается справочная система по контекстному идентификатору `HID_NEW_HELP`.

Контекстный идентификатор используется для связи программы со справочной системой, поэтому его необходимо объявить одновременно в двух файлах: в файле заголовка программы и в файле адресации справок. Поэтому определение данного идентификатора включается в файлы HelpAppView.h и NewHelp.hm. Вообще-то определение контекстного идентификатора может быть помещено в уже существующий файл HelpApp.hm и файл NewHelp.hm создан исключительно для демонстрации возможности его создания. Поскольку файл NewHelp.hm создан, он должен быть зарегистрирован в файле HelpApp.hpj.

Произведенных выше действий не достаточно для создания командной справки, поскольку с указанным контекстным идентификатором не связан никакой раздел справки, однако этот вопрос будет рассмотрен позднее при описании процедуры редактирования исходного текстового файла справки.

Создание системы контекстной справки

Система контекстной справки, обычно, используется при работе с диалоговыми окнами и командами меню. Поэтому, прежде всего, создадим демонстрационное диалоговое окно и поместим в нем кнопку, для которой будет создана контекстная справка.

Чтобы внести в приложение Help изменения, необходимые для демонстрации процедуры создания контекстной и других видов справки:

1. Откройте приложение Help и раскройте окно **Resource View** (Просмотр ресурсов).
2. Раскройте папку **Help.rc**, а в ней — папку **Menu** (Меню).
3. Дважды щелкните левой кнопкой мыши по значку `IDR_HelpTYPE`. Откроется окно редактирования ресурса меню.
4. Щелкните левой кнопкой мыши на прямоугольнике с надписью "Type Here", расположенном в правой части панели меню, введите в появившееся на его месте текстовое поле заголовок раскрывающегося меню "&Демо" и нажмите клавишу <Enter>.
5. Поместите указатель мыши на заголовок меню **Демо**, нажмите левую кнопку мыши и переместите его между командами меню **Вид** и **Окно**.

6. Отпустите кнопку мыши и щелкните на прямоугольнике с надписью "Type Here", расположенном в первой строке раскрывшегося меню.
7. Введите в появившееся на его месте текстовое поле команду меню "&Вызов" и нажмите клавишу <Enter>.
8. Щелкните правой кнопкой мыши на команде меню **Вызов** и выберите в появившемся контекстном меню команду **Properties** (Свойства).
9. В открывшемся окне **Properties** (Свойства) введите в текстовое поле раскрывающегося списка **ID** (Идентификатор ресурса) идентификатор команды `ID_DEMO_DIALOG` и нажмите клавишу <Enter>.
10. В текстовое поле **Prompt** (Подсказка) введите текст "Вызов диалогового окна\nВызов".
11. В окне **Class View** (Просмотр классов) щелкните правой кнопкой мыши по папке **Dialog** (Диалог) и выберите в появившемся контекстном меню команду **Insert Dialog** (Создать новый ресурс диалогового окна).
12. В окне редактирования ресурсов появится заготовка нового диалогового окна.
13. Щелкните левой кнопкой мыши на заготовке диалогового окна.
14. В окне **Properties** (Свойства) введите в текстовое поле раскрывающегося списка **ID** (Идентификатор ресурса) идентификатор ресурса `IDD_DEMO`, а в текстовое поле **Caption** (Заголовок) введите заголовок "Контекстная справка".
15. В раскрывающемся списке **Context help** (Контекстная справка), расположенном в том же разделе, что и раскрывающийся список **ID** (Идентификатор ресурса), выделите строку **True** (Истина).
16. В окне **Toolbox** (Инструментарий) выделите элемент управления **Button** (Кнопка) и перетащите его в заготовку диалогового окна под кнопку **Cancel** (Отмена).
17. В окне **Properties** (Свойства) введите в текстовое поле раскрывающегося списка **ID** (Идентификатор ресурса) идентификатор `IDC_CONTEXT`, а в текстовое поле **Caption** (Заголовок) введите заголовок "&Демо".
18. Щелкните левой кнопкой мыши по кнопке **Cancel** (Отмена) заготовки диалогового окна и введите в текстовое поле **Caption** (Заголовок) окна **Properties** (Свойства) заголовок "&Отмена".
19. Щелкните правой кнопкой мыши на заготовке диалогового окна и выберите в появившемся контекстном меню команду **Add Class** (Добавить класс). Появится диалоговое окно **Add Class - Help** (Добавить класс), изображенное на рис. 13.22.
20. В окне списка **Templates** (Шаблоны) выделите значок **MFC Class** (Класс MFC) и нажмите кнопку **Open** (Открыть). Появится диалоговое окно **MFC Class Wizard - Help** (Мастер создания классов MFC), изображенное на рис. 13.23.
21. Введите в текстовое поле **Class name** (Имя класса) имя класса `CDemoDlg`, выделите в раскрывающемся списке **Base class** (Базовый класс) имя базового класса `CDialog`, выделите в раскрывающемся списке **Dialog ID** (Идентификатор ресурса диалогового окна) `IDD_DEMO` и нажмите кнопку **Finish** (Готово). Диалоговое окно **MFC Class Wizard - Help** (Мастер создания классов MFC) закроется.

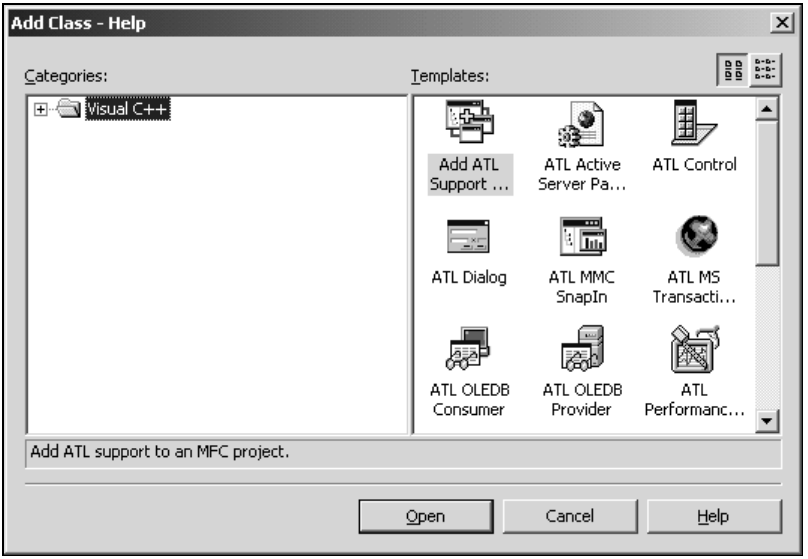


Рис. 13.22. Диалоговое окно **Add Class - Help**

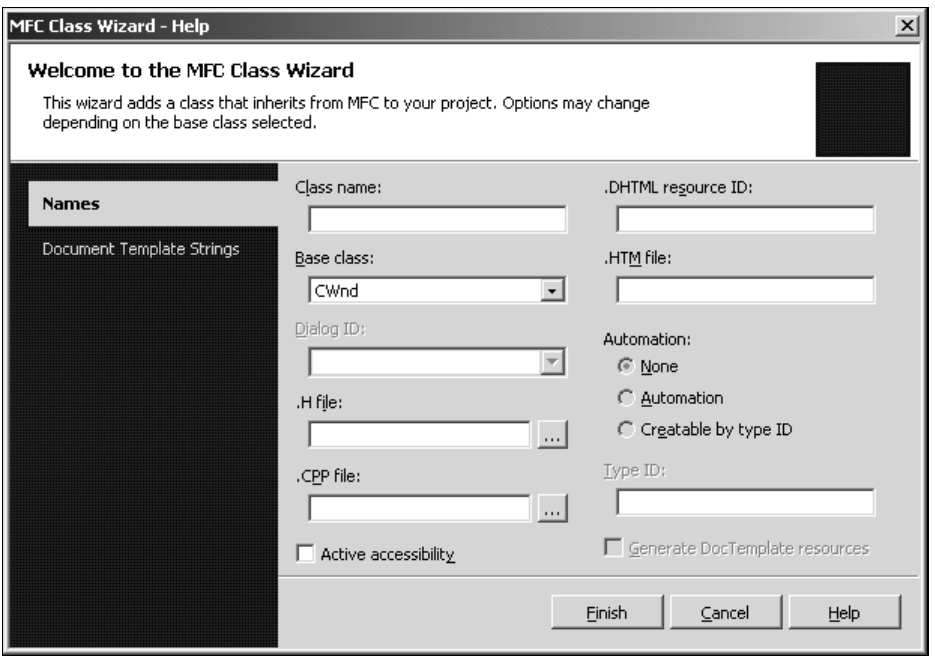


Рис. 13.23. Диалоговое окно **MFC Class Wizard - Help**

22. Откройте окно **Class View** (Просмотр класса), раскройте в нем папку **Help** (Помощь) и выделите папку `CHelpView`.
23. В окне **Properties** (Свойства) нажмите кнопку **Events** (События). Раскроется список событий, обрабатываемых данным классом.
24. Раскройте в этом списке папку `ID_DEMO_DIALOG`, выделите в ней строку `COMMAND` и добавьте в текстовое поле ее раскрывающегося списка функцию обработки данного сообщения. При этом откроется окно редактирования файла HelpView.cpp и текстовый курсор установится в тексте новой функции обработки сообщения .
25. Измените функцию `OnDemoDialog` в соответствии с текстом листинга 13.3.

Листинг 13.3. Функция `CHelpView::OnDemoDialog`

```
// Вывод демонстрационного диалогового окна
void CHelpView::OnDemoDialog(void)
{
  CDemoDlg  dlg(this);
  dlg.DoModal();
}
```

26. В начале файла после строки `#include "HelpView.h"` вставьте строку

```
#include "DemoDlg.h"
```

27. Откройте на редактирование файл DemoDlg.h и измените карту сообщений класса `CDemoDlg` в соответствии с приведенным ниже текстом.

```
afx_msg  BOOL OnHelpInfo(HELPINFO*);
afx_msg  void OnContextMenu(CWnd*, CPoint);
DECLARE_MESSAGE_MAP()
```

28. После строки `#pragma once` вставьте следующие строки:

```
#define HIDD_CONTEXT      2
#define HIDD_CONTEXT_OK    3
#define HIDD_CONTEXT_CANCEL    4
```

29. Откройте на редактирование файл DemoDlg.cpp и измените карту сообщений класса `CDemoDlg` в соответствии с приведенным ниже текстом.

```
BEGIN_MESSAGE_MAP(CDemoDlg, CDialog)
  ON_WM_HELPINFO()
  ON_WM_CONTEXTMENU()
END_MESSAGE_MAP()
```

30. После карты сообщений поместите следующий текст:

```
static  DWORD  aHelpIDs[] =
{
```

```
      IDC_CONTEXT, HIDD_CONTEXT,
      IDOK, HIDD_CONTEXT_OK,
      IDCANCEL, HIDD_CONTEXT_CANCEL,
      0, 0
    };
```

31. Вместо строки `// CDemoDlg message handlers` поместите текст листинга 13.4.

Листинг 13.4. Функция `CDemoDlg::OnHelpInfo`

```
// Функции обработки сообщений класса CDemoDlg

// Вывод контекстной справки
BOOL CDemoDlg::OnHelpInfo(HELPINFO* lpHelpInfo)
{
  if(lpHelpInfo->iContextType == HELPINFO_WINDOW)
    ::WinHelp((HWND) lpHelpInfo->hItemHandle,
    AfxGetApp()->m_pszHelpFilePath, HELP_WM_HELP, (DWORD) aHelpIDs);
  return TRUE;
}

void CDemoDlg::OnContextMenu(CWnd* pWnd, CPoint /*point*/)
{
  ::WinHelp((HWND) *pWnd, AfxGetApp()->m_pszHelpFilePath,
    HELP_CONTEXTMENU, (DWORD) aHelpIDs);
}
```

32. Откройте на редактирование файл NewHelp.hm и вставьте в его конец строки

```
    HIDD_CONTEXT    0x02
    HIDD_CONTEXT_OK    0x03
    HIDD_CONTEXT_CANCEL  0x04
```

При создании контекстной справки в карту сообщений добавляются объявления функций обработки сообщений контекстной справки.

В файл DemoDlg.cpp добавляется массив `aHelpIDs`, определяющий соответствие идентификаторов элементов управления контекстным идентификаторам. Используемые в нем контекстные идентификаторы должны быть объявлены в файлах заголовка и адресации справок. Для хранения этих идентификаторов используется созданный ранее файл адресации справок NewHelp.hm. Как и в предыдущем случае, вместо него мог бы быть использован файл HelpApp.hm.

При создании контекстной справки в окне **Properties** (Свойства) не устанавливался флажок **HelpID** (Идентификатор ресурса справки) (в соответствующем

раскрывающемся списке не выбирается значение **True** (Истина), позволяющий автоматически формировать контекстные идентификаторы, поскольку при установке данного флажка не производится автоматическое формирование массива соответствия идентификаторов элементов управления их контекстным идентификаторам, и не происходит автоматического включения их в файл заголовка соответствующего класса диалогового окна.

Введенные в класс диалогового окна функции обработки сообщений используются для вызова приложения WinHelp. Функция `CWnd::OnHelpInfo` вызывается приложением при нажатии пользователем клавиши <F1>. В ней производится проверка, что в данном случае справка вызывается для окна, а не для команды меню, и, в случае успешного завершения проверки, вызывается глобальная функция `WinHelp`. Необходимость вызова глобальной версии функции `WinHelp` обусловлена тем, что функция `CWinApp::WinHelp` не может принимать в качестве аргумента дескриптор окна.

Функция `CWnd::OnContextMenu` вызывается приложением при щелчке правой кнопки мыши в окне. В ней нет необходимости производить какие-либо проверки, поэтому она сразу вызывает глобальную функцию `WinHelp`.

Подготовка справочных текстов

Создатели языка Visual C++ понимали под подготовкой справочных текстов исключительно добавление или исключение разделов справки в исходных файлах справочной системы, имеющих расширение rtf, однако для русского пользователя встает еще и вопрос перевода уже содержащихся в них текстов на русский язык.

Русификация справочной системы приложения

Чтобы редактировать исходный файл справки:

1. Откройте приложение Microsoft Word, используя значок на Рабочем столе или кнопку **Start** (Пуск) на Панели задач.
2. Выберите команду меню **Файл | Открыть** (File | Open) или нажмите соответствующую кнопку на панели инструментов.
3. В появившемся диалоговом окне **Открытие документа** в раскрывающемся списке **Тип файлов** выберите **Текст в формате RTF**, раскройте папку hlp в каталоге вашего проекта, как это показано на рис. 13.24, и откройте нужный исходный файл справки.
4. Выберите команду **Вид | Сноски** (View | Footnotes). В окне редактирования текста появится панель сносок, как это показано на рис. 13.25.
5. Выберите команду **Сервис | Параметры** (Tools | Options), появится диалоговое окно **Параметры**.

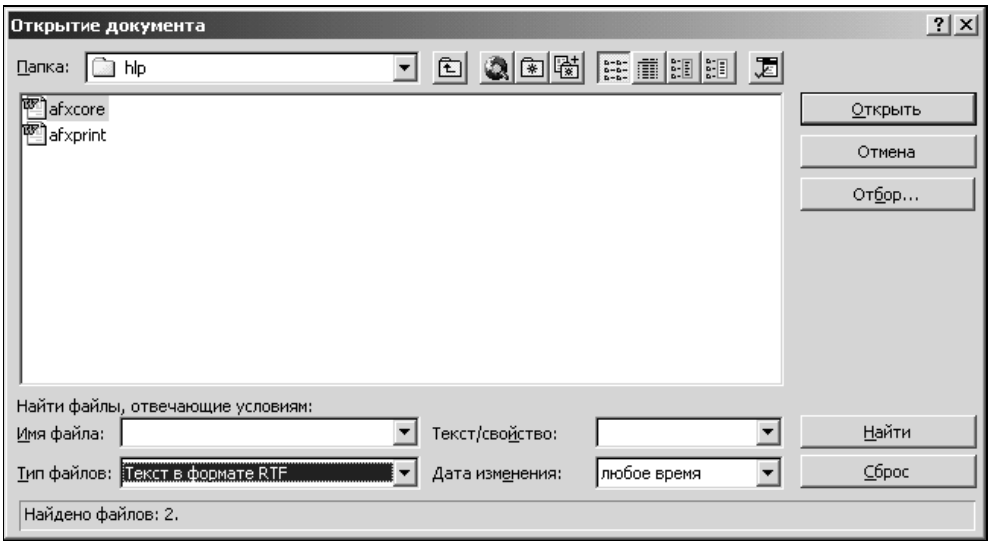


Рис. 13.24. Диалоговое окно **Открытие документа**

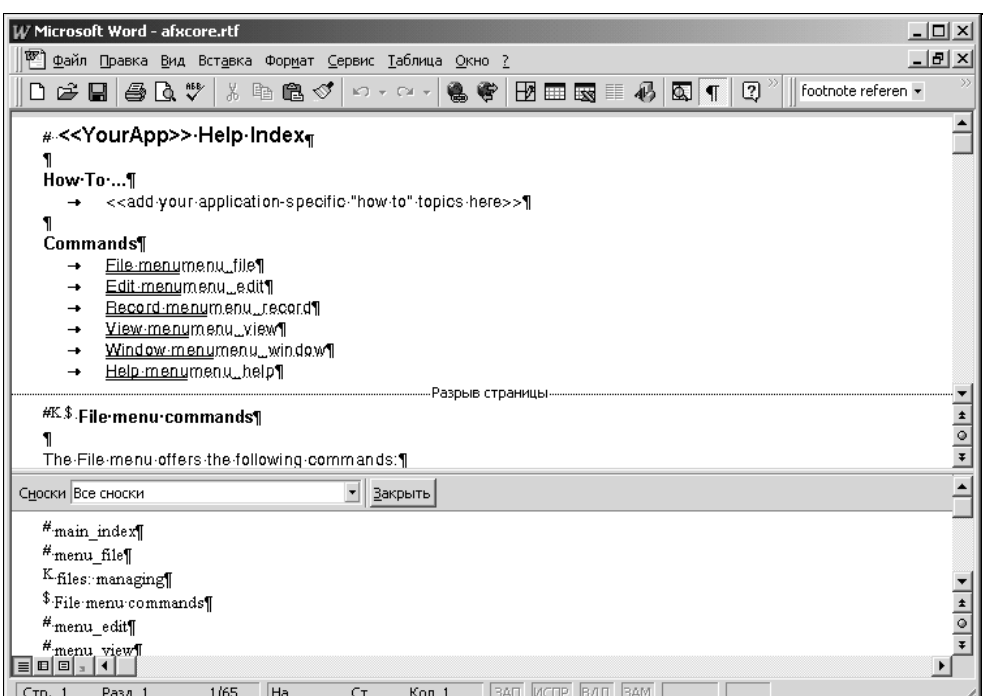


Рис. 13.25. Исходный файл справки с панелью сносок

6. Раскройте вкладку **Вид** (View) и установите флажок **скрытый текст** или флажок **все**, как это показано на рис. 13.26.

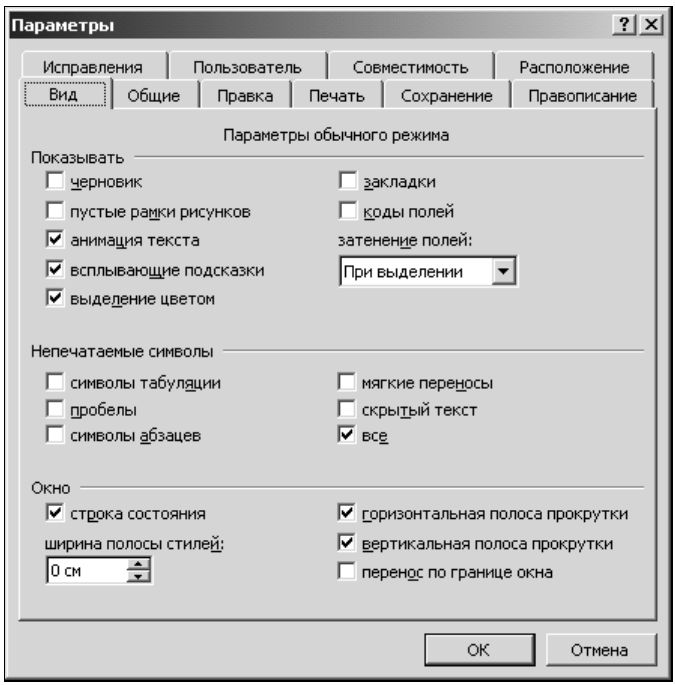


Рис. 13.26. Диалоговое окно **Параметры**

Скрытым текстом выводятся связи между разделами справочной системы. Каждый раздел начинается с новой страницы.

Справочная система использует восемь типов сносок, которые перечислены ниже:

- `#, идентификатор раздела справки` — используется глобальной функцией `WinHelp` при поиске данного раздела справки;
- `$, заголовок раздела` – выводится на экран при завершении поиска;
- `K, ключевые слова` — выводятся во вкладку **Index** (Предметный указатель) диалогового окна **Help Topics** (Справочная система);
- `A, ключевое слово типа A` — эти ключевые слова могут вызывать переход к следующей теме, но не выводятся во вкладку **Index** (Предметный указатель) диалогового окна **Help Topics** (Справочная система);
- `+, код просмотра` — определяет положение данного раздела в последовательности разделов;

- `!, вход макроса` — определяет макрос раздела, запускаемый при запросе данного раздела пользователем;
- `*, тег построения` — используется для включения определенных тегов в файл справочной системы специального формата;
- `>, тип окна` — определяет тип окна, используемого для вывода данного раздела.

Подчеркнутый двойной линией текст, за которым следует скрытый текст, указывает на возможность перехода к другому разделу справочной системы. Если пользователь щелкнет левой кнопкой мыши на данном тексте при его отображении в окне справки, то вместо текущей темы в данном окне будет отображаться новая тема, определяемая скрытым текстом.

Если текст перед скрытым текстом подчеркнут одной линией, то справка, определяемая скрытым текстом, появится во всплывающем окне поверх текущего окна справки. В обоих случаях скрытый текст содержит идентификатор раздела справочной системы, на которую будет сделан переход или которая будет отображаться во всплывающем окне справки.

Создаваемые мастером MFC Application Wizard исходные файлы справочной системы содержат достаточно большой объем информации, не относящейся к создаваемой заготовке приложения, но эти темы могут стать актуальными при превращении этой заготовки в рабочее приложение. Кроме того, в нем предусмотрены строки подсказок: какого рода информацию следует подставить вместо них.

Русификация приложения подразумевает, конечно, и русификацию справочной системы. Однако мастер MFC Application Wizard не позволяет создать исходный файл справки на русском языке. Поэтому ниже будет приведен вариант исходного файла справки на русском языке, созданный путем замены соответствующих текстов в исходном файле справки, созданном мастером MFC Application Wizard.

Поскольку исходный файл справки имеет достаточно сложный формат, который необходимо максимально точно передать при его воспроизведении в книге, в него были введены следующие изменения:

- начало новой страницы отмечается строкой `<<Новая страница>>`;
- комбинация заголовка раздела, подчеркнутого двойной линией, и идентификатора раздела, выведенного скрытым текстом, представляется в виде слитного написания заголовка и идентификатора раздела, в котором заголовок подчеркнут одной линией;
- местодержатель `<<YourApp>>`, используемый для обозначения имени приложения, остается без изменения;
- удалена ссылка на заголовок раздела **Record menu**, для которой отсутствует раздел справочной системы.

С использованием приведенных выше соглашений по изменению формата файл afxcore.rtf после его русификации имеет следующий вид:

```
# <<YourApp>> Предметный указатель


Что это такое ...
  <<добавьте сюда разделы контекстной справки вашего приложения>>


Команды меню
  Меню файлmenu_file
  Меню Правкаmenu_edit
  Меню Видmenu_view
  Меню Окномenu_window
  Меню Справкаmenu_help


<<Новая страница>>
#K $ Команды меню Файл


Меню Файл содержит следующие команды:


СоздатьHID_FILE_NEW            Создание нового документа.
ОткрытьHID_FILE_OPEN           Открытие существующего документа.
ЗакрытьHID_FILE_CLOSE          Закрывает текущий документ.

СохранитьHID_FILE_SAVE         Сохраняет текущий документ используя то же
                               имя файла.
Сохранить какHID_FILE_SAVE_AS  Сохраняет текущий документ в файле
                               с указанным именем.
ПечатьHID_FILE_PRINT           Печать текущего документа.
Предварительный просмотрHID_FILE_PRINT_PREVIEW    Выводит окно
                               предварительного просмотра печати.
Настройка принтераHID_FILE_PRINT_SETUP  Выбор принтера и настройка его
                               параметров.
Отправить...HID_FILE_SEND_MAIL  Посылает активный документ по электронной
                                почте.
ВыходHID_APP_EXIT               Завершает работу приложения <<YourApp>>.


<<Новая страница>>
#Команды меню Правка


Меню Правка содержит следующие команды:


ОтменитьHID_EDIT_UNDO                Отменяет предыдущую операцию
                                     редактирования.
```

ВырезатьHID_EDIT_CUT		Уничтожает выделенный фрагмент в документе и помещает его в буфер обмена.
КопироватьHID_EDIT_COPY		Копирует в буфер обмена выделенный фрагмент в документе.
ВставитьHID_EDIT_PASTE		Копирует данные из буфера обмена в активный документ.
Специальная вставкаHID_EDIT_PASTE_LINK		Копирует из буфера обмена связь с данными другого приложения.
Вставка объектаHID_OLE_INSERT_NEW		Вставляет в документ или связывает с ним объект, такой как диаграмма или формула.
СвязиHID_OLE_EDIT_LINKS		Выводит и позволяет редактировать связи с внедренными документами.

<<Новая страница>>

#**Команды меню Вид**

Меню Вид содержит следующие команды:

Панель инструментовHID_VIEW_TOOLBAR	Отображает или скрывает панель инструментов.
Панель состояния HID_VIEW_STATUS_BAR	Отображает или скрывает панель состояния.

<<Новая страница>>

#**Команды меню Окно**

Меню Окно содержит следующие команды, позволяющие пользователю организовывать несколько представлений нескольких документов в многооконном приложении:

НовоеHID_WINDOW_NEW	Создает новое окно для отображения текущего документа.
КаскадироватьHID_WINDOW_CASCADE	Выводит перекрывающиеся окна.
УпорядочитьHID_WINDOW_TILE	Выводит неперекрывающиеся окна.
Упорядочить значкиHID_WINDOW_ARRANGE	Упорядочивает значки минимизированных окон.
РазделитьHID_WINDOW_SPLIT	Разделяет активное окно на панели.
Окно 1, 2, ...HID_WINDOW_ALL	Осуществляет переход к указанному окну.

<<Новая страница>>

#**Команды меню Справка**

Меню Справка содержит следующие команды, позволяющие получить справочную информацию по данному приложению:

Вызов справки	HID_HELP_INDEX	Выводит предметный указатель по разделам справочной системы.
О программе	HID_APP_ABOUT	Выводит номер версии данного приложения.

<<Новая страница>>

#K $ **Команда Создать (Меню Файл)**

Данная команда меню используется для создания нового документа в приложении <<YourApp>>. Тип нового документа определяется в диалоговом окне Создание документа AFX_HIDD_NEWTYPEDLG. << Удалите предыдущее предложение, если ваше приложение работает только с одним типом документа. >>

Для открытия существующего документа может быть использована команда меню Открыть HID_FILE_OPEN.

Другие способы вызова

Панель инструментов: {bmc filenew.bmp}

Комбинация клавиш: CTRL+N

<<Новая страница>>

#**Диалоговое окно Создание документа**

<< Удалите этот раздел, если ваше приложение работает только с одним типом документа. >>

Определяет типы документов, создаваемые в приложении:

<< Перечислите типы документов, создаваемых в приложении >>

<<Новая страница>>

#K $ **Команда Открыть (Меню Файл)**

Данная команда меню используется для открытия существующего документа в новом окне. В приложении могут быть одновременно открыты несколько документов. Для перемещения между окнами документов используйте меню Окно. См. команду меню Окно 1, 2, ... HID_WINDOW_ALL.

Для создания нового документа используйте команду меню Создать HID_FILE_NEW.

Другие способы вызова

Панель инструментов: {bmc fileopen.bmp}

Комбинация клавиш: CTRL+O

<<Новая страница>>

#**Диалоговое окно Открытие документа**

Перечисленные ниже элементы управления диалогового окна Открытие документа позволяют задать открываемый файл:

Имя файла

Текстовое поле, в которое нужно ввести имя открываемого файла. Имя файла можно выделить в расположенном выше окне списка и открыть его двойным щелчком левой кнопки мыши.

Тип файлов

Определяет тип и расширение файлов, отображаемых в окне списка. В данном приложении могут быть открыты файлы следующих типов: << Перечислите типы файлов, используемых в приложении. >>

Папка

Раскрывающийся список, позволяющий открыть любую папку на любом диске компьютера.

<<Новая страница>>

#K $ **Команда Закрыть (Меню Файл)**

Данная команда меню используется для закрытия всех окон, содержащих текущий документ. Если данный документ не был сохранен перед данной операцией, приложение <<YourApp>> выводит окно сообщения о необходимости сохранить изменения, внесенные в данный документ. Если документ не имел имени, то автоматически открывается диалоговое окно Сохранение документаAFX_HIDD_FILESAVE, в котором предлагается выбрать имя файла сохраняемого документа.

<< Все описанные выше действия должны быть реализованы в данном приложении. >>

Документ может быть закрыт с использованием приведенного ниже значка Закрыть в окне документа:

{bml scmenu.bmp}

<<Новая страница>>

#K $ **Команда Сохранить (Меню Файл)**

Данная команда меню используется для сохранения активного документа с использованием его текущего имени и каталога. При сохранении документа без имени приложение <<YourApp>> выводит диалоговое окно Сохранение документаAFX_HIDD_FILESAVE, позволяющее выбрать имя файла сохраняемого документа.

Если необходимо изменить имя или каталог существующего документа перед его сохранением, выберите команду меню Сохранить какHID_FILE_SAVE_AS.

Другие способы вызова

Панель инструментов: {bmc filesave.bmp}

Комбинация клавиш: CTRL+S

<<Новая страница>>

#K $ **Команда Сохранить как (Меню Файл)**

Данная команда меню используется для сохранения активного документа в новом каталоге или под новым именем. Данная команда меню выводит диалоговое окно Сохранение документаAFX_HIDD_FILESAVE .

Чтобы сохранить документ под его текущим именем в своем каталоге выберите команду меню СохранитьHID_FILE_SAVE.

<<Новая страница>>

#**Команда Отправить (Меню Файл)**

Данная команда меню используется для пересылки активного документа по электронной почте. Эта команда выводит окно электронной почты, в котором содержится активный документ. Пользователь должен заполнить поля адреса, темы и другие поля, необходимые для пересылки сообщения, и нажать кнопку Отправить.

<<Новая страница>>

#**Диалоговое окно Сохранение документа**

Перечисленные ниже элементы управления диалогового окна Сохранение документа позволяют определить имя файла, в котором будет сохранен документ:

Имя файла

Текстовое поле, в которое нужно ввести имя файла, в котором будет сохранен документ. Если предполагается сохранить документ в одном из существующих файлов, выделите его имя в расположенном выше окне списка и дважды щелкните левой кнопкой мыши. После этого появится диалоговое окно с вопросом о необходимости замены существующего файла.

Тип файлов

Определяет тип и расширение файлов, отображаемых в окне списка.

В данном приложении могут сохраняться файлы следующих типов:

<< Перечислите типы файлов, используемых в приложении. >>

Папка

Раскрывающийся список, позволяющий открыть любую папку на любом диске компьютера.

<< Опишите другие элементы управления диалогового окна Сохранение документа, набор которых определяется флагами OFN_ в структуре OPENFILENAME, используемой в конструкторе класса CFileDialog. >>

<<Новая страница>>

K $ **Команды 1, 2, 3, 4 (Меню Файл)**

Номера и имена файлов, перечисленных в нижней части меню Файл, позволяют открыть файлы документов, которые были закрыты последними. Для вызова файла по номеру в списке одновременно нажмите клавишу Shift и соответствующий номер.

<<Новая страница>>

#K $ **Команда Выход (Меню Файл)**

Данная команда меню закрывает текущую копию приложения <<YourApp>>. При наличии в приложении <<YourApp>> несохраненных документов оно выводит сообщение, предупреждающее о необходимости их сохранить..

Другие способы вызова

Мышь:	Нажать кнопку Закрыть, расположенную в заголовке главного окна приложения. {bmc appexit.bmp}
Комбинация клавиш:	ALT+F4

<<Новая страница>>

Команда Отменить (Меню Правка)

<< Способ использования данной команды в приложении может отличаться от описанного ниже. Если это так, измените текст данного раздела. >>

Данная команда используется для отмены, если это возможно, последней операции редактирования.

Другие способы вызова

Панель инструментов:	{bmc editundo.bmp}
Комбинация клавиш:	CTRL+Z или ALT+BACKSPACE

<<Новая страница>>

Команда Повторить (Меню Правка)

<< Содержимое данного раздела зависит от приложения. >>

<<Новая страница>>

#**Команда Вырезать (Меню Правка)**

Данная команда меню уничтожает выделенный фрагмент в документе и помещает его в буфер обмена. Команда недоступна, если в приложении отсутствует выделенный фрагмент.

Помещенные в буфер обмена данные замещают прежнее его содержимое.

Другие способы вызова

Панель инструментов: {bmc editcut.bmp}

Комбинация клавиш: CTRL+X

<<Новая страница>>

#**Команда Копировать (Меню Правка)**

Данная команда меню копирует в буфер обмена выделенный фрагмент в документе. Команда недоступна, если в приложении отсутствует выделенный фрагмент.

Помещенные в буфер обмена данные замещают прежнее его содержимое.

Другие способы вызова

Панель инструментов: {bmc editcopy.bmp}

Комбинация клавиш: CTRL+C

<<Новая страница>>

#**Команда Вставить (Меню Правка)**

Данная команда меню копирует данные из буфера обмена в активный документ. Команда недоступна, если в буфере обмена не содержится информации.

Другие способы вызова

Панель инструментов: {bmc editpast.bmp}

Комбинация клавиш: CTRL+V

<<Новая страница>>

#K $ **Команда Панель инструментов (Меню Вид)**

Данная команда меню используется для вывода на экран и скрытия панели инструментов, содержащей кнопки наиболее часто используемых команд приложения <<YourApp>>, таких как Открыть. Для вывода на экран панели инструментов необходимо установить флажок у данного пункта меню.

См. справочную информацию по теме Панель инструментовAFX_HIDW_TOOLBAR.

<<Новая страница>>

#K $ **Панель инструментов**

{bml hlptbar.bmp}

Панель инструментов изначально выводится в верхней части главного окна приложения под панелью меню. Она может быть установлена вдоль любой из сторон главного окна приложения или выводиться в виде отдельного окна. Панель инструментов позволяет получить быстрый доступ к основным командам приложения <<YourApp>>.

Чтобы вывести или скрыть панель инструментов, выберите команду Вид, Панель инструментов.

<< Откорректируйте приведенный ниже список кнопок панели инструментов вашего приложения. >>

Кнопка	**Описание**
{bmc filenew.bmp}	Создает новый документ.
{bmc fileopen.bmp}	Открывает существующий документ. При этом приложение <<YourApp>> выводит диалоговое окно Открытие документа, позволяющее выбрать нужный файл.
{bmc filesave.bmp}	Сохраняет активный документ без изменения его имени. Если активный документ не имеет имени, приложение <<YourApp>> выводит диалоговое окно Сохранение документа.
{bmc fileprnt.bmp}	Печатает активный документ.
{bmc editcut.bmp}	Уничтожает выделенный фрагмент в документе и помещает его в буфер обмена.
{bmc editcopy.bmp}	Копирует в буфер обмена выделенный фрагмент в документе.
{bmc editpast.bmp}	Копирует данные из буфера обмена в активный документ.
{bmc editundo.bmp}	Отменяет последнюю операцию редактирования. Примечание: Некоторые операции не могут быть отменены.
{bmc recfirst.bmp}	Переход к первой записи в выделенном фрагменте.

{bmc recprev.bmp} Переход к предыдущей записи в выделенном фрагменте.

{bmc recnext.bmp} Переход к следующей записи в выделенном фрагменте.

{bmc reclast.bmp} Переход к последней записи в выделенном фрагменте.

<<Новая страница>>

#K $ **Команда Строка состояния (Меню Вид)**

Данная команда меню используется для вывода на экран или скрытия строки состояния, в которую выводится краткая справка по выделенной команде меню или кнопке панели инструментов, а также состояние некоторых управляющих клавиш. Для вывода на экран строки состояния необходимо установить флажок у данного пункта меню.

См. справочную информацию по теме Строка состоянияAFX_HIDW_STATUS_BAR.

<<Новая страница>>

#K $**Строка состояния**

{bml hlpsbar.bmp}

Строка состояния выводится в нижней части главного окна приложения <<YourApp>>. Чтобы вывести на экран строку состояния, выберите команду Вид, Строка состояния.

В левой части строки состояния выводится справочная информация по выделенным командам меню. Здесь же выводится справочная информация по кнопкам панели инструментов, на которых располагается указатель мыши.

Правая часть строки состояния содержит панели, в которых отображается состояние следующих индикаторов клавиатуры:

Панель	**Описание**
CAP	Горит индикатор Caps Lock.
NUM	Горит индикатор Num Lock.
SCRL	Горит индикатор Scroll Lock.

<<Новая страница>>

Команда Новое (Меню Окно)

Данная команда меню создает новое окно для вывода активного документа. Пользователь может открыть несколько окон для отображения различных фрагментов одного документа. При изменении содержимого одного окна все окна, в которых

отображается данный документ, отразят это изменение. При открытии нового окна оно становится активным и выводится поверх всех других открытых окон.

<<Новая страница>>

#**Команда Каскадировать (Меню Окно)**

Данная команда меню выводит перекрывающиеся окна.

<<Новая страница>>

#**Команда Упорядочить (Меню Окно)**

Данная команда меню выводит неперекрывающиеся окна.

<<Новая страница>>

#**Команда Расположить по горизонтали (Меню Окно)**

Данная команда меню выводит неперекрывающиеся окна, расположенные одно над другим (если количество окон не превышает трех).

<<Новая страница>>

#**Команда Расположить по вертикали (Меню Окно)**

Данная команда меню выводит неперекрывающиеся окна, расположенные одно рядом с другим (если количество окон не превышает трех).

<<Новая страница>>

#**Команда Упорядочить значки (Меню Окно)**

Данная команда упорядочивает значки минимизированных окон в нижней части главного окна приложения. Если в нижней части главного окна приложения располагается открытое окно, то некоторые или все значки могут быть невидны, поскольку будут располагаться под этим окном.

<<Новая страница>>

Команда Разделить (Меню Окно)

Данная команда меню позволяет разделить активное окно на панели. Для установки разделительной полосы между панелями можно использовать мышь или клавиши управления курсором. По завершении настройки щелкните кнопкой мыши или нажмите клавишу Enter. Нажатие клавиши Esc возвращает разделительную полосу в ее первоначальное положение.

<< В однооконном приложении эта команда расположена в меню Вид. >>

<<Новая страница>>

#**Команды 1,2,… (Меню Окно)**

Приложение <<YourApp>> выводит список открытых окон документов в нижней части меню Окно. Перед именем активного окна ставится флажок. Выбор документа из списка делает его окно активным.

<<Новая страница>>

#**Команда Вывод справки (Меню Справка)**

Данная команда меню выводит диалоговое окно Справочная система и раскрывает в нем вкладку Предметный указатель. Эта вкладка позволяет производить в справочной системе приложения поиск по ключевому слову.

Для возврата в приложение необходимо закрыть диалоговое окно Справочная система.

<<Новая страница>>

#**Команда Использование справки (Меню Справка)**

Данная команда меню позволяет получить информацию по пользованию справочной системой приложения.

<<Новая страница>>

#**Команда О программе (Меню Справка)**

Данная команда меню позволяет вывести информацию о правах собственности и версии приложения <<YourApp>>.

<<Новая страница>>

#$**Команда Справка**

{bml curhelp.bmp}

Команда Справка позволяет получить справочную информацию о различных элементах управления приложения <<YourApp>>. При нажатии кнопки Справка на панели инструментов указатель мыши приобретает вид стрелки и знака вопроса. При щелчке таким указателем на элементах управления приложения <<YourApp>> вызывает контекстную справку по данному элементу управления window.

Другие способы вызова

Комбинация клавиш: SHIFT+F1

<<Новая страница>>

#**Заголовок окна**

<< Поместите здесь заголовок вашего приложения. >>

Заголовок окна располагается в его верхней части. Он содержит имя приложения и документа.

Для перемещения окна следует перетащить его заголовок.

Примечание: Для перемещения диалоговых окон следует перетащить их заголовок.

Заголовок может содержать следующие элементы:

{bmc bullet.bmp} Системное меню приложения

{bmc bullet.bmp} Меню документа

{bmc bullet.bmp} Кнопка Развернуть окно

{bmc bullet.bmp} Кнопка Свернуть окно

{bmc bullet.bmp} Имя приложения

{bmc bullet.bmp} Имя документа

{bmc bullet.bmp} Кнопка Восстановить окно

<<Новая страница>>

Полосы прокрутки

Отображаются у правой и нижней границ окна документа. Бегунки, расположенные в полосах прокрутки, показывают вертикальное и горизонтальное положение окна в документе. Для прокрутки документа может быть использована мышь.

<< Опишите действия, которые можно производить с полосой прокрутки в вашем приложении. >>

<<Новая страница>>

Команда Размер (Меню документа)

Эта команда отображает на экране крестик, образованный из четырех стрелок, направленных наружу. Этот крестик может использоваться для изменения размеров активного окна с использованием клавиш управления курсором.

{bml curarw4.bmp}

После того как курсор принял форму крестика из стрелок:

1. Нажмите одну из клавиш управления курсором (влево, вправо, вверх или вниз) и переместите курсор на ту сторону рамки окна, которую необходимо переместить.
2. Используйте клавиши управления курсором для перемещения стороны рамки.
3. Нажмите клавишу ENTER, когда окно примет желаемый размер.

Примечание: Эта команда недоступна, если окно расширено до максимального размера.

Другие способы вызова

Мышь: Поместите указатель мыши на рамку окна, нажмите левую кнопку мыши и, не отпуская ее, измените размер окна.

<<Новая страница>>

Команда Переместить (Меню документа)

Эта команда отображает на экране крестик, образованный из четырех стрелок, направленных наружу. Этот крестик может использоваться для перемещения активного окна или диалогового окна с использованием клавиш управления курсором.

{bmc curarw4.bmp}

Примечание: Эта команда недоступна, если окно расширено до максимального размера.

Другие способы вызова

Комбинация клавиш: CTRL+F7

<<Новая страница>>

#$**Команда Свернуть (Системное меню)**

Данная команда меню позволяет свернуть приложение <<YourApp>> в значок.

Другие способы вызова

Мышь: Нажмите кнопку Свернуть окно {bmc scmin.bmp} в заголовке окна.

Комбинация клавиш: ALT+F9

<<Новая страница>>

#**Команда Развернуть (Меню документа)**

Данная команда меню позволяет расширить размеры окна таким образом, чтобы оно занимало все доступное пространство.

Другие способы вызова

Мышь:	Нажмите кнопку Развернуть {bmc scmax.bmp} в заголовке окна или дважды щелкните в заголовке окна.
Комбинация клавиш:	CTRL+F10 для окон документов.

<<Новая страница>>

#**Команда Следующее (Меню документа)**

Данная команда меню используется для перехода к следующему окну документа. Приложение <<YourApp>> определяет следующее окно исходя из порядка их открытия.

Другие способы вызова

Комбинация клавиш:	CTRL+F6

<<Новая страница>>

#**Команда Следующее (Меню документа)**

Данная команда меню используется для перехода к следующему окну документа. Приложение <<YourApp>> определяет следующее окно исходя из порядка их открытия.

Другие способы вызова

Комбинация клавиш:	SHIFT+CTRL+F6

<<Новая страница>>

#**Команда Закрыть (Меню документа)**

Данная команда меню используется для закрытия активного окна или диалогового окна.

Вместо выбора этой команды можно нажать кнопку Закрыть окно в заголовке активного окна.

{bml appexit.bmp}

Примечание: Если для одного документа открыто несколько окон, то для одновременного закрытия всех окон данного документа следует выбрать команду Файл | Закрыть.

Другие способы вызова

Комбинация клавиш: CTRL+F4 закрывает окно документа
ALT+F4 закрывает приложение <<YourType>>
или диалоговое окно

<<Новая страница>>

#**Команда Восстановить (Меню документа)**

Данная команда позволяет возвратить размер и позицию активного окна до вызова для него команд Развернуть или Свернуть.

<<Новая страница>>

#**Список задач (Меню приложения)**

Данная команда позволяет открыть меню Пуск операционной системы Windows.

Другие способы вызова

Комбинация клавиш: CTRL+ESC

<<Новая страница>>

#$**Команда Линейка (Меню Вид)**

<< Содержание данной темы зависит от приложения. >>

<<Новая страница>>

#$ **Диалоговое окно Шрифт**

<< Содержание данной темы зависит от приложения. >>

<<Новая страница>>

#$ **Диалоговое окно Установка цвета**

<< Содержание данной темы зависит от приложения. >>

<<Новая страница>>

#$**Команда Найти (Меню Правка)**

<< Содержание данной темы зависит от приложения. >>

<<Новая страница>>

#$**Диалоговое окно Найти**

<< Содержание данной темы зависит от приложения. >>

<<Новая страница>>

#$**Команда Заменить (Меню Правка)**

<< Содержание данной темы зависит от приложения. >>

<<Новая страница>>

#$ **Диалоговое окно Найти и заменить**

<< Содержание данной темы зависит от приложения. >>

<<Новая страница>>

#$ **Команда Повторить (Меню Правка)**

Данная команда позволяет повторить последнюю команду редактирования.

Другие способы вызова

Клавиша: F4

<<Новая страница>>

#$**Команда Очистить (Меню Правка)**

<< Содержание данной темы зависит от приложения. >>

<<Новая страница>>

#$**Команда Очистить все (Меню Правка)**

<< Содержание данной темы зависит от приложения. >>

<<Новая страница>>

#$**Следующая панель**

<< Содержание данной темы зависит от приложения. >>

```
<<Новая страница>>
#$Предыдущая панель

<< Содержание данной темы зависит от приложения. >>

<<Новая страница>>
# Внесение изменений в документ

<< Содержание данной темы зависит от того, какие типы документов используются в приложении и какие изменения в них можно вносить . >>

<<Новая страница>>
#Справочная информация отсутствует

По данной области окна справочная информация отсутствует.

<<Новая страница>>
# Справочная информация отсутствует

По данному окну сообщения справочная информация отсутствует.

<< Если необходимо создать справочную информацию, связанную с каждым окном сообщения, удалите значения AFX_HIDP_xxx из раздела [ALIAS] файла HPJ, и создайте раздел справки для каждого значения AFX_HIDP_xxx. Например, AFX_HIDP_INVALID_FILENAME представляет собой идентификатор справки для окна сообщения о недопустимом имени файла. >>

<<Новая страница>>
# Область фиксации панелей инструментов

Здесь могут фиксироваться панели инструментов приложения. Зафиксированная панель инструментов может быть превращена в плавающую перетаскиванием ее в рабочую область окна.
```

Для полной русификации данного файла необходимо изменить список его сносок в соответствии с приведенным ниже текстом:

```
# main_index    # AFX_HIDW_DOCKBAR_TOP
# menu_file
K Работа с файлами
$ Команды меню Файл
```

```
# menu_edit
# menu_view
# menu_window
# menu_help
# HID_FILE_NEW
K Работа с файлами
$ Команда меню Файл | Создать
# AFX_HIDD_NEWTYPEDLG
# HID_FILE_OPEN
K Работа с файлами
$ Команда меню Файл | Открыть
# AFX_HIDD_FILEOPEN
# HID_FILE_CLOSE
K Работа с файлами
$ Команда меню Файл |  Закрыть
# HID_FILE_SAVE
K Работа с файлами
$ Команда меню Файл | Сохранить
# HID_FILE_SAVE_AS
K Работа с файлами
$ Команда меню Файл | Сохранить как
# HID_FILE_SEND_MAIL
# AFX_HIDD_FILESAVE
# HID_FILE_MRU_FILE1
K Работа с файлами
$ Команда меню Файл, 1, 2, 3, 4
# HID_APP_EXIT
K Выход из приложения
$ Команда меню Файл | Выход
# HID_EDIT_UNDO
# HID_EDIT_REDO
# HID_EDIT_CUT
# HID_EDIT_COPY
# HID_EDIT_PASTE
# HID_VIEW_TOOLBAR
K Панель инструментов
$ Команда меню Вид | Панель инструментов
# AFX_HIDW_TOOLBAR
K Панель инструментов
$ Панель инструментов
```

```
# HID_VIEW_STATUS_BAR
K Строка состояния
$ Команда меню Вид | Строка состояния
# AFX_HIDW_STATUS_BAR
K Строка состояния
$ Строка состояния
# HID_WINDOW_NEW
# HID_WINDOW_CASCADE
# HID_WINDOW_TILE
# HID_WINDOW_TILE_HORZ
# HID_WINDOW_TILE_VERT
# HID_WINDOW_ARRANGE
# HID_WINDOW_SPLIT
# HID_WINDOW_ALL
# HID_HELP_INDEX
# HID_HELP_USING
# HID_APP_ABOUT
# HID_CONTEXT_HELP
$ Команда Справка
# HID_HT_CAPTION
# scrollbars
# HID_SC_SIZE
# HID_SC_MOVE
# HID_SC_MINIMIZE
$ Команда меню Свернуть
# HID_SC_MAXIMIZE
# HID_SC_NEXTWINDOW
# HID_SC_PREVWINDOW
# HID_SC_CLOSE
# HID_SC_RESTORE
# HID_SC_TASKLIST
# HID_VIEW_RULER
$ Команда меню Вид | Линейка
# AFX_HIDD_FONT
$ Диалоговое окно Шрифт
# AFX_HIDD_COLOR
$ Диалоговое окно Установка цвета
# HID_EDIT_FIND
$ Команда меню Правка | Найти
# AFX_HIDD_FIND
```

```
$ Диалоговое окно Найти
# HID_EDIT_REPLACE
$ Команда меню Правка | Заменить
# AFX_HIDD_REPLACE
$ Диалоговое окно Найти и заменить
# HID_EDIT_REPEAT
$ Команда меню Правка | Заменить
# HID_EDIT_CLEAR
$ Команда меню Правка | Очистить
# HID_EDIT_CLEAR_ALL
$ Команда меню Правка | Очистить все
# HID_NEXT_PANE
$ Команда Следующая панель
# HID_PREV_PANE
$ Команда Предыдущая панель
# HIDR_DOC1TYPE
# HID_HT_NOWHERE
# AFX_HIDP_default

# AFX_HIDW_DOCKBAR_TOP
```

Аналогичные изменения должны быть внесены и в текст файла afxprint.rtf. Его текст приведен ниже:

```
<<Новая страница>>
#K $ Команда меню Печать (Меню Файл)

Данная команда меню используется для печати документа. Эта команда вызывает
диалоговое окно ПечатьAFX_HIDD_PRINT, позволяющее пользователю задать диапа-
зон печатаемых страниц, число копий, принтер, на котором будет производиться
печать, и другие параметры печати.

Другие способы вызова
  Панель инструментов:  {bmc fileprnt.bmp}
  Комбинация клавиш:    CTRL+P

<<Новая страница>>
# Диалоговое окно Печать
```

Диалоговое окно Печать содержит следующие группы элементов управления, позволяющие задать параметры печати документа:

Принтер

Позволяет выбрать принтер и установить его параметры. Принтер выбирается из раскрывающегося списка Имя. Для изменения параметров принтера нажмите кнопку Свойства.

Кнопка Свойства

Раскрывает диалоговое окно Свойства AFX_HIDD_PRINTSETUP, позволяющее установить параметры принтера.

Печатать

Определяет диапазон печатаемых страниц в документе:

Все Печатает весь документ.

Выделенный фрагмент Печатает выделенный фрагмент текста.

Страницы Печатает страницы, определяемые текстовыми полями "с" и "по".

Копии

Определяет количество печатаемых копий страниц диапазона печати.

Разобрать

Установка этого флажка приводит к тому, что документ печатается в порядке возрастания номеров страниц, а не делает по несколько копий одной страницы.

<<Новая страница>>

#**Диалоговое окно Процесс печати**

Диалоговое окно Процесс печати выводится приложением <<YourApp>> в процессе подготовки документа для печати. В нем указывается число обработанных страниц, имя используемого принтера и порт, через который он подключен.

Для прекращения процесса печати нажмите кнопку Отмена (если успеете).

<<Новая страница>>

#K $ **Предварительный просмотр (Меню Файл)**

Данная команда меню используется для отображения активного документа в специальном окне, позволяющем оценить вид документа после его печати. При выборе данной команды активное окно замещается окном предварительного просмотра печати, в которое выводятся одна или две страницы документа в том виде, в котором они будут напечатаны. В верхней части окна располагается панель инструментов предварительного просмотра печатиAFX_HIDW_PREVIEW_BAR. Она содержит кнопки, позволяющие одновременно просматривать одну или две страницы документа, перемещаться по страницам документа, увеличивать или уменьшать масштаб отображения и инициировать процесс печати.

<<Новая страница>>

#K $ **Панель инструментов предварительного просмотра печати**

Панель инструментов предварительного просмотра печати содержит следующие кнопки:

Печать

Выводит диалоговое окно Печать для инициализации процесса печати.

Следующая

Предварительный просмотр следующей печатаемой страницы.

Предыдущая

Предварительный просмотр предыдущей печатаемой страницы.

Одна страница / Две страницы

Переход к одновременному просмотру указанного числа страниц.

Крупнее

Увеличивает масштаб отображения в окне.

Мельче

Уменьшает масштаб отображения в окне.

Закрыть

Закрывает окно предварительного просмотра печати и возвращается к отображению активного окна.

<<Новая страница>>

#K $ **Команда Настройка принтера (Меню Файл)**

Данная команда меню позволяет установить параметры используемого принтера. Она вызывает диалоговое окно Настройка принтераAFX_HIDD_PRINTSETUP.

<<Новая страница>>

Диалоговое окно Настройка принтера

Диалоговое окно Настройка принтера имеет следующие элементы управления:

Печать

Группа элементов управления Печать позволяет выбрать принтер, на котором будет распечатываться документ, и установить его параметры.

Ориентация

Переключатель Ориентация позволяет установить ориентацию листа при печати (Книжная или Альбомная).

Размер

Раскрывающийся список Размер группы Бумага позволяет выбрать формат листа, на который будет производиться печать.

Подача

Раскрывающийся список Подача группы Бумага позволяет установить способ подачи листов в принтере (Автоматическая подача, Ручная подача или подача с перфорацией).

Свойства

Кнопка Свойства группы Печать вызывает диалоговое окно Свойства, позволяющее установить параметры, специфичные для каждого принтера.

<<Новая страница>>

#$ **Команда Параметры страницы (Меню Файл)**

<< Содержимое данного раздела зависит от приложения. >>

Список сносок данного файла должен иметь следующий вид:

```
# HID_FILE_PRINT
K Печать и предварительный просмотр
$ Команда меню Файл | Печать
# AFX_HIDD_PRINT
# AFX_HIDD_PRINTDLG
# HID_FILE_PRINT_PREVIEW
K Печать и предварительный просмотр
$ Команда меню Файл | Предварительный просмотр
# AFX_HIDW_PREVIEW_BAR
K Печать и предварительный просмотр
$ Панель инструментов предварительного просмотра печати
# HID_FILE_PRINT_SETUP
K Печать и предварительный просмотр
$ Команда меню Файл | Настройка принтера
# AFX_HIDD_PRINTSETUP
# HID_FILE_PAGE_SETUP
$ Команда меню Файл | Параметры страницы
```

Для полной русификации справочной системы приложения остается внести изменения в файлы с расширениями cnt и hpj. В данном случае имеется только по одному файлу с каждым из расширений, имя которых совпадает с именем приложения. Файл Help.cnt, расположенный в подкаталоге hlp каталога приложения, должен содержать следующий текст:

```
:Base Help.hlp
1 Меню
2 Меню Файл=menu_file
```

```
2 Меню Правка=menu_edit
2 Меню Вид=menu_view
2 Меню Окно=menu_window
2 Меню Справка=menu_help
1 <<Вставьте сюда заголовки разделов справки вашего приложения>>
2 <<Вставьте сюда переходы к отдельным разделам справки>>=main_index
```

В файле Help.hpj, расположенном в том же каталоге, необходимо заменить строку `TITLE=Help Application Help` на строку

```
TITLE=Справка по приложению HELP
```

После внесения этих изменений справочная система приложения будет полностью русифицирована, однако она будет содержать большое количество ненужной информации и указаний на то, какую информацию в нее следует вставить или удалить.

Чтобы удалить раздел справочной системы, необходимо удалить всю страницу, содержащую его описание (вместе с одним из символов разрыва страницы, расположенным до или после нее). После редактирования текст справочной системы не должен содержать строк-местодержателей (строк, заключенных в угловые скобки (`<< >>`), и используемых для хранения указаний пользователю).

Кроме строк-местодержателей приложение содержит местодержатель `<<YourApp>>`, используемый в качестве имени текущего приложения. Этот местодержатель во всех исходных текстах справочной системы должен быть заменен именем приложения. Остается непонятным, почему эта операция не производится при создании приложения мастером MFC Application Wizard. Наверное, это очень сложная операция для специалистов из Microsoft. Поэтому она поручается создателю приложения. В нашем случае эта операция не производится, чтобы текст данной справки мог быть скопирован в любое приложение.

Добавление новых тем

При добавлении в проект нового ресурса необходимо также включить информацию о нем в файл формата RTF. Visual C++ автоматически создает идентификаторы контекстных ссылок, которые можно использовать в качестве ярлыков. Идентификаторы контекстных ссылок образуются из идентификаторов ресурсов добавлением буквы "H" в начало каждого идентификатора ресурса. Назначаемые им значения формируются как сумма значения идентификатора ресурса и некоторого базового значения. Базовые значения определены в файле afxpriv.h. Их описание содержится в разделе **TN028 Context-Sensitive Help Support** (Поддержка контекстной справки) справочной системы Visual C++. Значения идентификаторов контекстных ссылок содержатся в файле карты справочной системы (hm), используемом Visual C++. Если с момента предыдущей компиляции приложения в файл resource.h были внесены изменения, Visual C++ запускает приложение MakeHm. Приложение MakeHm читает файл resource.h и соз-

дает файл hm, который включается в раздел [MAP] файла hpj. Предположим, что файл resource.h содержит следующие идентификаторы ресурсов:

```
#define IDD_ABOUTBOX      100
#define IDR_MAINFRAME    128
#define IDR_DEMOAPP      129
#define IDM_FIRST      0X9000
#define IDM_SECOND      0X9001
#define IDM_THIRD      0X9002
```

В этом случае файл hm будет выглядеть следующим образом:

```
// Команды (ID_* and IDM_*)
HIDM_FIRST      0X19000
HIDM_SECOND      0X19001
HIDM_THIRD      0X19002

// Подсказки (IDP_*)

// Ресурсы (IDR_*)
HIDR_MAINFRAME    0x20080
HIDR_DEMOAPP      0x20081

// Диалоговые окна Dialogs (IDD_*)
HIDD_ABOUTBOX      0x20064

// Элементы управления рамки окна (IDW_*)
```

При добавлении новых тем не рекомендуется вносить их в файлы, подготовленные мастером AppWizard. Для этого лучше создать отдельные файлы. Каждый новый файл должен быть зарегистрирован в секции [FILES] файла проекта (с расширением hpj).

Чтобы включить в справочную систему приложения разделы, связанные со спецификой данного приложения:

1. Откройте на редактирование приложение Help.
2. Откройте окно **Solution Explorer** (Проводник решения), раскройте в нем папку **Help** (Помощь), а в ней папку **Source Files** (Файлы реализации).
3. Дважды щелкните левой кнопкой мыши на имени файла HelpApp.hpj. Откроется окно редактирования файла проекта справки.
4. В конец раздела [FILES] файла HelpApp.hpj добавьте строку

   ```
   help.rtf
   ```

5. Не закрывая среды программирования Visual C++ вызовите MS Word.

6. Откройте файл afxcore.rtf и сохраните его под именем help.rtf.
7. По методике, описанной в начале предыдущего раздела, установите режим отображения скрытого текста и раскройте панель сносок.
8. Используя форматирование разделов справки файла afxcore.rtf поместите в файл help.rtf следующий текст.

```
#K $ Команды меню Демо

Меню Демо содержит следующие команды:

ВызовHID_DEMO_DIALOG  Вызов диалогового окна

<<Новая страница>>
#K $ Команда Вызов (Меню Демо)

Данная команда меню используется для вызова диалогового окна,
демонстрирующего принципы программирования контекстной справки.

<<Новая страница>>
#Кнопка Демо (Диалоговое окно Контекстная справка)

Данная кнопка существует исключительно для того, чтобы можно было
вывести этот текст.

<<Новая страница>>
#Кнопка OK (Диалоговое окно Контекстная справка)

Данная кнопка служит для выхода из диалогового окна Контекстная
справка.

<<Новая страница>>
#Кнопка Отмена (Диалоговое окно Контекстная справка)

Данная кнопка служит для выхода из диалогового окна Контекстная
справка.

<<Новая страница>>
#K $ Дополнительные возможности приложения

Данное приложение является чисто демонстрационным. Данный раздел
демонстрирует возможности создания командной справки. Для демонстрации
```

```
возможности создания контекстной справки используется диалоговое окно
Контекстная справкаHIDD_DEMO.


<<Новая страница>>

#Диалоговое окно Контекстная справка


Данное диалоговое окно служит для демонстрации возможностей
программирования контекстной справки.
```

9. Измените список сносок файла help.rtf в соответствии со следующим текстом.

```
# menu_demo
K Демонстрационное меню
$ Команды меню Демо
# HID_DEMO_DIALOG
K Демонстрационное меню
$ Команда меню Демо | Вызов
# HIDD_CONTEXT
# HIDD_CONTEXT_OK
# HIDD_CONTEXT_CANCEL
# HID_NEW_HELP
K Дополнительно
$ Новая справка
# HIDD_DEMO
```

10. Нажмите клавишу <F5> и запустите на исполнение приложение Help.
11. Выберите команду **Демо | Вызов**. Появится диалоговое окно **Контекстная справка**.
12. Нажмите кнопку контекстной справки (кнопка со знаком вопроса, расположенная в заголовке диалогового окна). Курсор мыши примет вид курсора контекстной справки.
13. Щелкните левой кнопкой мыши по кнопке **Демо**. Появится контекстная справка, изображенная на рис. 13.27.
14. Закройте диалоговое окно **Контекстная справка** и выберите команду меню **? | Новая справка**. Появится окно **Справка по приложению HELP**, изображенное на рис. 13.28.
15. Закройте окно справки и закройте приложение.

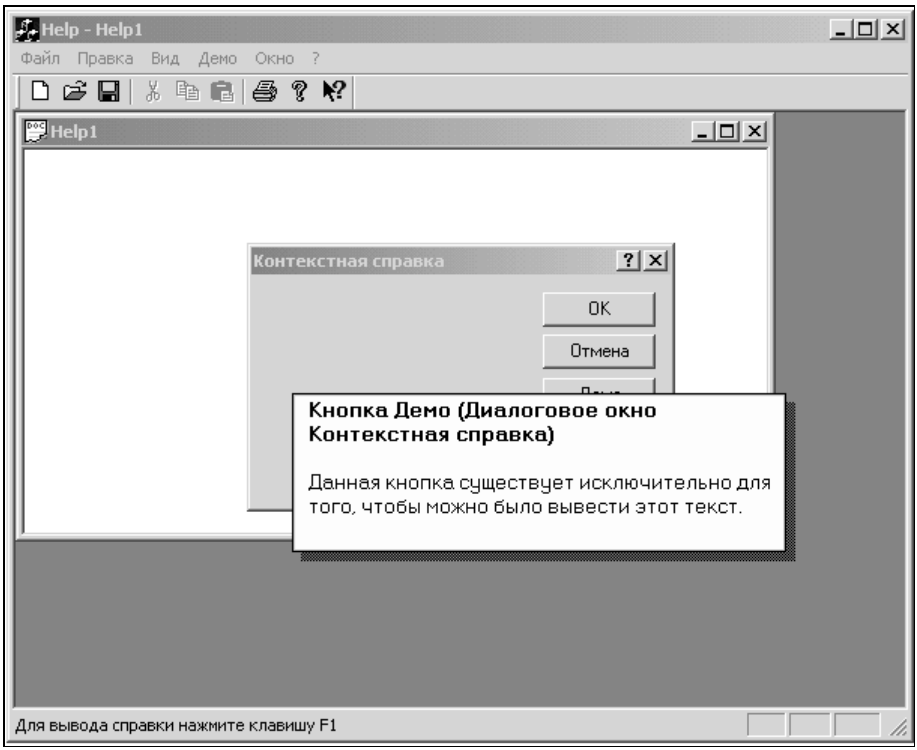


Рис. 13.27. Вывод контекстной справки

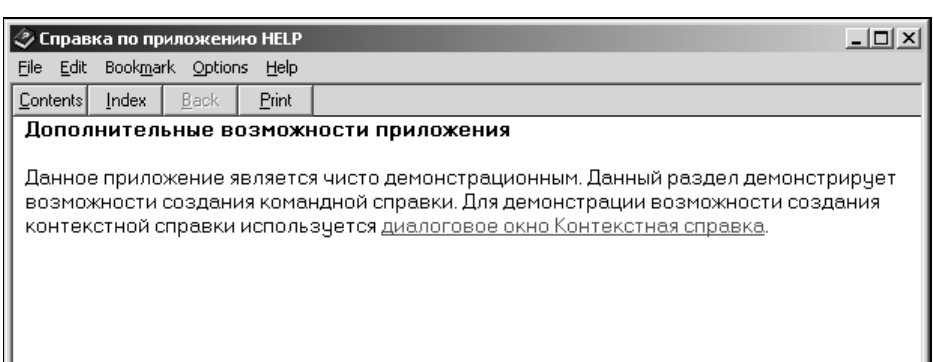


Рис. 13.28. Окно **Справка по приложению HELP**

Форматирование текстовых файлов справки

Обычно, для введения новых разделов в справочную систему достаточно воспользоваться копией уже имеющихся разделов. Однако, если у вас нет желания выис-

кивать соответствующим образом отформатированные фрагменты текста, вы можете самостоятельно произвести соответствующее форматирование в MS Word.

Чтобы вставить сноску:

1. Введите текст, к которому будет относиться сноска, и установите текстовый курсор на то место, где должна появиться сноска.
2. Выберите команду **Вставка | Сноска**.
3. В появившемся диалоговом окне **Сноски**, изображенном на рис. 13.29, введите в текстовое поле требуемый символ сноски (при этом переключатель **Нумерация** автоматически встанет в положение **другая**) и нажмите кнопку **OK**.

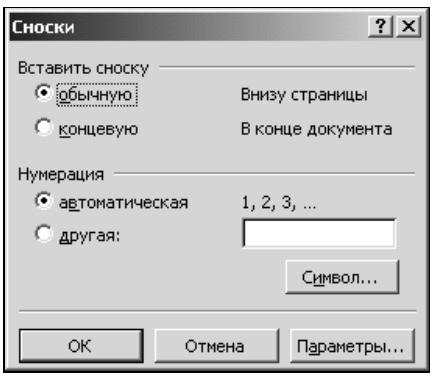


Рис. 13.29. Диалоговое окно **Сноски**

4. На панели **Сноски** введите текст данной сноски (зависит от типа сноски).
5. Выделите введенный символ сноски без стоящей за ним точки и выберите из раскрывающегося списка **Стиль** панели инструментов **Форматирование** стиль **footnote reference** (Знак ссылки).
6. Повторите п. 5 для знака ссылки в основном тексте файла справки.

Чтобы дважды подчеркнуть текст, необходимо выделить его и нажать комбинацию клавиш <Ctrl>+<Shift>+<D> или выполнить следующую последовательность действий:

1. Выберите команду **Формат | Шрифт**. Появится диалоговое окно **Шрифт**, изображенное на рис. 13.30.
2. Раскройте список **Подчеркивание** и выделите в нем строку **Двойное** и нажмите кнопку **OK**. Диалоговое окно **Шрифт** закроется.

Чтобы сделать текст скрытым, необходимо выделить его и нажать комбинацию клавиш <Ctrl>+<Shift>+<H> или выполнить следующую последовательность действий:

1. Выберите команду **Формат | Шрифт**. Появится диалоговое окно **Шрифт**.

2. Установите в этом диалоговом окне флажок **скрытый** и нажмите кнопку **OK**. Диалоговое окно **Шрифт** закроется.

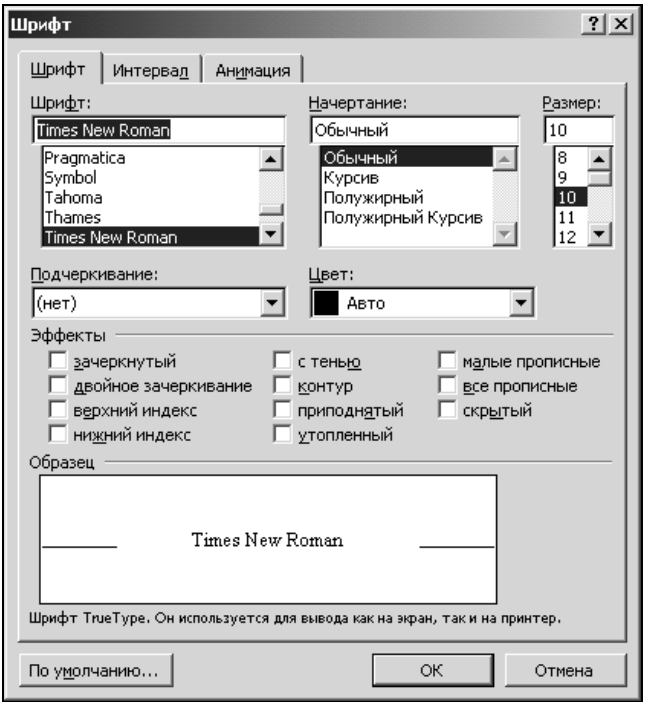


Рис. 13.30. Диалоговое окно **Шрифт**

Примечание

При помещении в текст справочной системы гипертекстовых ссылок убедитесь, что между подчеркнутым и скрытым текстом отсутствуют пробелы. Если скрытый текст расположен в конце абзаца, убедитесь, что символ возврата каретки и перевода строки не попал в скрытый текст.

Чтобы начать текст с новой страницы перед его вводом следует нажать комбинацию клавиш <Ctrl>+<Enter> или выполнить следующую последовательность действий:

1. Выберите команду меню **Вставка | Разрыв**. Появится диалоговое окно **Разрыв**, изображенное на рис. 13.31.
2. В группе **Начать** установите переключатель **новую страницу** и нажмите кнопку **OK**. Диалоговое окно **Разрыв** закроется.

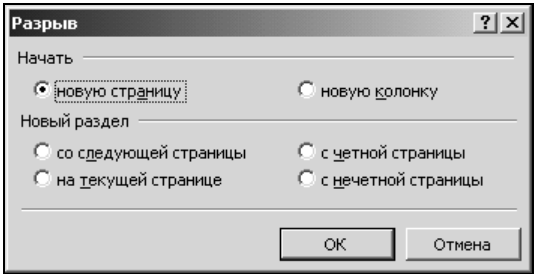


Рис. 13.31. Диалоговое окно **Разрыв**

Использование графики и гиперграфических ссылок

При добавлении в файл формата RTF графических или гиперграфических объектов их можно вставить непосредственно в файл или использовать для этого ссылку. Проще всего непосредственно вставить растровое изображение в файл справки, но этот подход имеет массу ограничений. Вставка ссылки на растровое изображение не намного сложнее, но обладает многими преимуществами. При вставке ссылки на изображение в файл добавляется специальный текст, помещенный в скобки. При компиляции файла справки компилятор включит изображение в текст файла hlp. Ниже приведен синтаксис ссылки на файл растрового изображения:

```
{Команда ИмяФайла}
```

`ИмяФайла` в этой строке определяет имя файла растрового изображения без пути к нему. Путь определяется в записи `BMROOT` раздела `[OPTION]` файла hpj. `Команда` задает компилятору способ вставки растрового изображения. Команды приведены в табл. 13.3.

Таблица 13.3. *Команды графических ссылок*

Команда	Описание
`bmc`	Выравнивать растровое изображение как текст
`bml`	Выравнивать растровое изображение по левому краю. Текст может обтекать правый край изображения
`bmr`	Выравнивать растровое изображение по правому краю. Текст может обтекать левый край изображения

В Visual C++ 6.0 для создания гиперграфических ссылок использовался специальный редактор (SHED.EXE). Этот редактор позволял работать с файлами bmp, dib и wmf, а также добавлять области срабатывания в существующие изображения. Результат мог быть сохранен в файле гиперграфики (SHG). В поставке Visual C++ 7.0 такой файл отсутствует.

После того как в изображении выделена область срабатывания, оно может быть включено по ссылке в файл формата RTF и скомпилировано в файл справки (hlp). В качестве области срабатывания можно указать любую прямоугольную область изображения. После выделения области срабатывания необходимо присвоить ей контекстный идентификатор и определить ее тип, которым может быть ссылка, всплывающее окно или макрос.

Использование макросов справочной системы

Макросы справочной системы представляют собой программы, встроенные в приложение WinHelp и позволяющие настраивать справочную систему. Для настройки справочной системы может быть использовано более 50 макросов WinHelp, работающих со справочными файлами. Эти макросы можно сгруппировать в четыре основные категории:

- работа с кнопками;
- работа с меню;
- гипертекстовые ссылки;
- вспомогательные макросы WinHelp.

Макрос может быть помещен в файл hpj, файл rtf или непосредственно в приложение. Если макрос помещается в раздел `[CONFIG]`, макрос будет запущен при первом открытии файла справки приложением WinHelp. Если макрос помещается в концевую сноску какой-либо темы, он запускается при вызове справки по этой теме. При помещении макроса в концевую сноску по теме обязательно используйте специальный символ концевой сноски — восклицательный знак (`!`). Макрос может быть связан не с меткой перехода темы, а с гипертекстовой или гиперграфической ссылкой. Ниже приведен пример вызова макроса в приложении:

```
char macroName[] = "HelpMacro()";

WinHelp(hWnd, "DemoApp.hlp", HELP_COMMAND,
reinterpret_cast<DWORD>(macroName));
```

Внесение изменений в оглавление справочной системы

Для внесения изменений в оглавление справочной системы лучше воспользоваться приложением Help Workshop. В Visual C++ 6.0 это приложение могло быть запущено из меню **Пуск**, расположенного на Панели задач. Конфигурация Visual Studio.NET такой возможности не предусматривает.

Чтобы внести изменения в оглавление справочной системы с использованием приложения Help Workshop:

1. Не выходя из среды программирования Visual C++ выберите в панели задач команду меню **Start | Run** (Пуск | Выполнить). Появится диалоговое окно **Run** (Выполнить), изображенное на рис. 13.32.

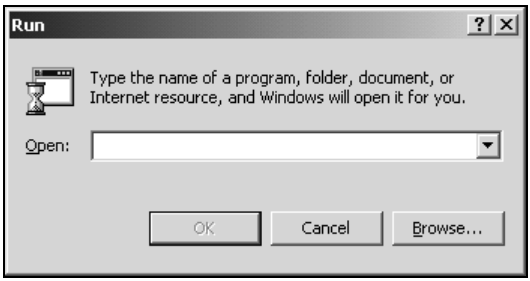


Рис. 13.32. Диалоговое окно **Run**

2. Нажмите кнопку **Browse** (Просмотр). Откроется диалоговое окно **Browse** (Просмотр), изображенное на рис. 13.33.

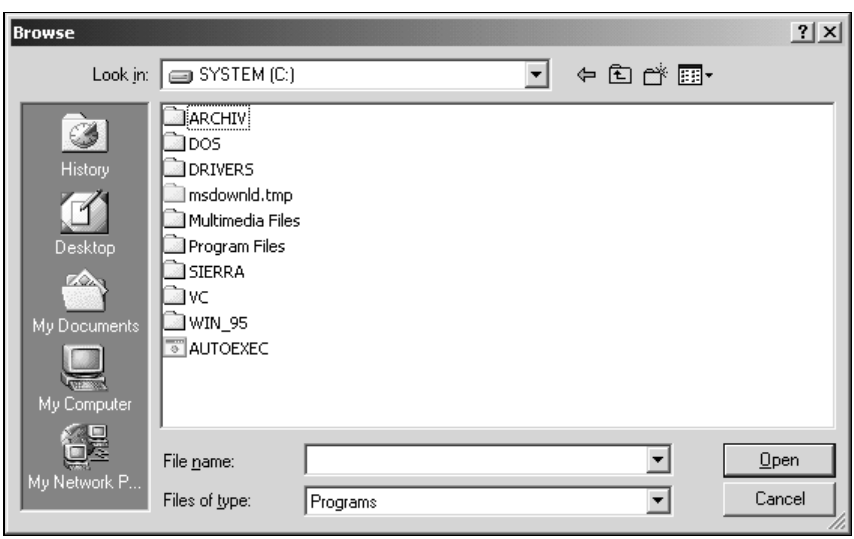


Рис. 13.33. Диалоговое окно **Browse**

3. Найдите исполняемый файл hcw.exe (при установке по умолчанию он располагается в папке C:\Program Files\Microsoft Visual Studio.NET\Common7\Tools), выделите его и нажмите кнопку **Open** (Открыть). Имя данного файла появится в текстовом поле **Open** (Открыть) диалогового окна **Run** (Выполнить).
4. Нажмите кнопку **OK**. Откроется приложение Help Workshop.
5. Выберите команду меню **File | Open** (Файл | Открыть) или нажмите кнопку **Open** (Открыть) на панели инструментов.
6. В появившемся диалоговом окне раскройте папку hlp в каталоге вашего приложения, в раскрывающемся списке **Тип файлов** выберите **Help Contents**

(*.cnt) (Справка предметного указателя) и дважды щелкните по имени файла HelpApp.

7. Откроется окно редактирования файла контекста, изображенное на рис. 13.34.

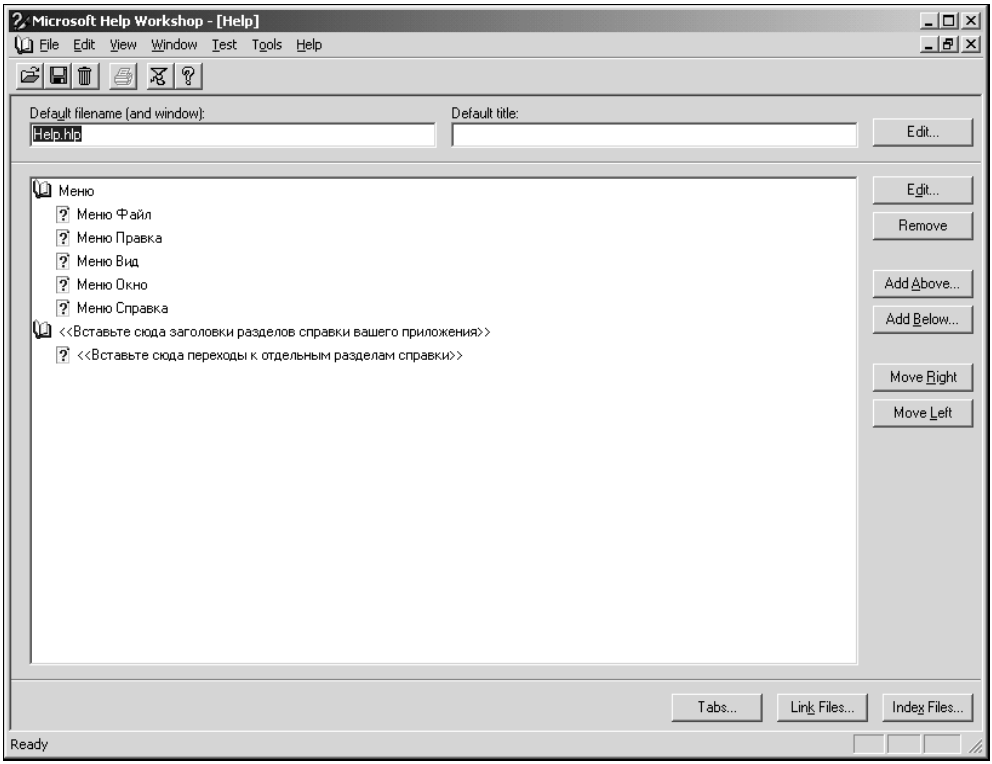


Рис. 13.34. Окно редактирования файла контекста

8. Выделите строку **Меню Окно** и нажмите кнопку **Add Above** (Вставить выше) (или выделите строку **Меню Вид** и нажмите кнопку **Add Below** (Вставить ниже)). Появится диалоговое окно **Edit Contents Tab Entry** (Добавление новых пунктов в оглавление справочной системы), изображенное на рис. 13.35.
9. Введите в текстовое поле **Title** (Заголовок) — `Меню Демо`, а в текстовое поле **Topic ID** (Идентификатор справки) — `menu_demo`. Нажмите кнопку **OK**.
10. Выделите строку **Меню Справка** и нажмите кнопку **Add Below** (Вставить ниже).
11. В диалоговом окне **Edit Contents Tab Entry** (Добавление новых пунктов в оглавление справочной системы) установите переключатель в положение **Heading** (Заголовок) и введите в текстовое поле **Title** (Заголовок) — "Дополнительные возможности" (остальные поля при таком положении переключателя становятся недоступными). Нажмите кнопку **OK**.

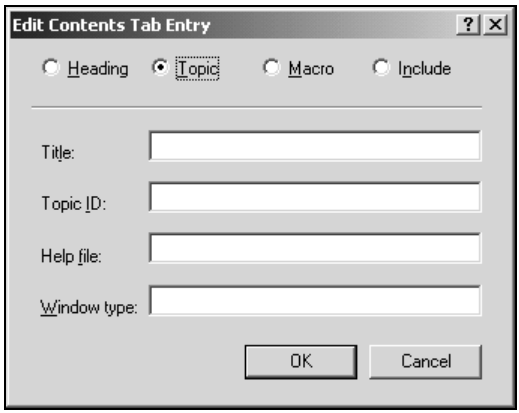


Рис. 13.35. Диалоговое окно **Edit Contents Tab Entry**

12. Нажмите кнопку **Add Below** (Вставить ниже) и в появившемся диалоговом окне **Edit Contents Tab Entry** (Добавление новых пунктов в оглавление справочной системы) введите в текстовое поле **Title** (Заголовок) — "Новая справка", а в текстовое поле **Topic ID** (Идентификатор справки) — `HID_NEW_HELP`. Нажмите кнопку **OK**.
13. Переместитесь на одну строчку вниз и нажмите кнопку **Remove** (Удалить). Заголовок со строкой-местодержателем исчезнет.
14. Повторите эту операцию еще раз.
15. Закройте окно редактирования файла контекста. Появится окно сообщения **Microsoft Help Workshop** (Мастер справки Microsoft) с вопросом о необходимости сохранения внесенных изменений.
16. Нажмите кнопку **Yes** (Да) в окне сообщения **Microsoft Help Workshop** (Мастер справки Microsoft) и выйдите из приложения Help Workshop.
17. Вернитесь в среду программирования Visual C++. Появится диалоговое окно **Microsoft Development Environment** (Среда разработки Microsoft) с сообщением о том, что в файл Help.cnt были внесены изменения, и с вопросом о том, следует ли его перегрузить, как это показано на рис. 13.36.

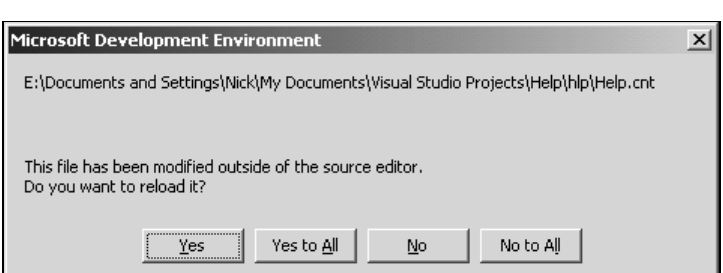


Рис. 13.36. Диалоговое окно **Microsoft Development Environment**

18. Нажмите кнопку **Yes** (Да) и запустите на исполнение приложение Help.
19. Убедитесь, что все внесенные изменения нашли отражение в справочной системе.

Использование справочной системы HTML

Использование справочной системы HTML позволяет расширить диапазон способов предоставления информации и повысить качество справочной системы приложения. Эта справочная система может работать с гиперссылками, элементами управления ActiveX, сценариями и динамическим языком HTML (DHTML). По оценкам Microsoft справочная система HTML представляет собой следующее поколение интерактивных систем разработки. Как и WinHelp, приложение HTML Help использует файлы проектов для объединения файлов тем, контекста, индексов и других файлов в одном скомпилированном файле справочной системы. Какие типы файлов будут включены в конкретный проект справочной системы, зависит от его структуры и предполагаемого способа распространения. После создания полноценной справочной системы HTML она может быть размещена на диске, в Internet или локальной сети. В табл. 13.4 приведено описание некоторых наиболее часто встречающихся файлов справочной системы HTML.

Таблица 13.4. *Наиболее часто встречающиеся файлы справочной системы HTML*

Имя	Тип	Описание
Проект	HPP	Используется для работы с файлами описания тем, оглавления, индексов и других исходных файлов. Также определяет внешний вид справочной системы
Описание разделов справочной системы	HTM, HTML	Содержит текст, появляющийся на экране и коды форматирования (теги), сообщающие браузеру, как выводить каждую страницу
Изображения	JPEG, GIF, PNG	Файлы изображений HTML
Оглавление	HHC	Содержит заголовки тем для вывода их в оглавлении
Индекс	HHK	Содержит ключевые слова индекса
Альтернативные имена	ALI	Устанавливает соответствие идентификаторов темам справки
Скомпилированный файл	CHM	Единственный файл, содержащий все элементы справочной системы. Этот файл поставляется с приложением и может быть просмотрен в Internet Explorer

Так же как и WinHelp, приложение HTML Help позволяет объединить различные компоненты справочной системы в файл проекта. Для создания файла про-

екта, а также для работы с большинством других файлов справочной системы, используется приложение HTML Help Workshop. Одной из полезных особенностей этого приложения является то, что оно может преобразовывать в свой формат существующие проекты формата WinHelp.

После преобразования проекта WinHelp или создания нового проекта HTML Help можно добавлять или удалять файлы разделов справочной системы HTML, а также определять положение файлов индексов, оглавления, изображений и мультимедийных файлов. При создании нового файла разделов справочной системы HTML приложение HTML Help Workshop создает структуру, в которую может быть вставлен текст, указатели, графические и анимационные объекты и звуковые файлы, а также добавлены связи и ссылки на разделы, расположенные в том же самом файле, в другом файле или на Web-странице. Каждый раздел должен быть помещен в свой файл htm. Программист может создать свое собственное оглавление, позволяющее пользователю открывать файлы разделов справочной системы HTML щелчком мыши на соответствующем разделе. Файл раздела справочной системы может являться частью локального файла chm или файла, расположенного на некоторой удаленной системе или Web-странице. Файлы индекса обрабатываются так же, как и файлы разделов справочной системы, но в них вместо разделов справочной системы используются ключевые слова. В зависимости от требований, предъявляемых приложением, может использоваться один из двух типов индексов. При работе с файлом chm используется двоичный индекс, а при работе с Web-страницей — индекс карты страницы.

Преобразование справочной системы приложения

Как было сказано выше, приложение HTML Help Workshop позволяет преобразовывать проект справочной системы формата WinHelp в проект своего формата. При этом следует обратить внимание на то, что преобразуется не тип справочной системы проекта, а проект справочной системы, что далеко не одно и то же.

Результатом преобразования проекта справочной системы формата WinHelp является проект справочной системы формата HTML, который может быть подключен к любому другому приложению.

Прежде, чем приступить к преобразованию проекта справочной системы, необходимо создать приложение, в котором эта справочная система будет использована. Дело в том, что в одном и том же приложении не могут одновременно использоваться оба типа справочной системы, поскольку при работе со справочной системой HTML в конструктор класса приложения необходимо вызвать функцию `CWinApp::EnableHtmlHelp`. Кроме того, эти приложения различаются содержимым пакетного файла компиляции справки (при создании обычной справки он имеет имя makehelp.bat, а при компиляции справки HTML — makehtmlhelp.bat).

Новое приложение будет называться HTML и его текст можно найти в одноименной папке на дискете, прилагаемой к данной книге. Чтобы самостоятельно создать данное приложение:

1. Выполните все операции, указанные в *главе 1* для создания многооконного приложения с именем HTML, но в диалоговом окне **MFC Application Wizard** (Мастер создания приложений MFC) не нажимайте кнопку **Finish** (Готово).
2. Вместо этого раскройте в нем вкладку **Advanced Features** (Дополнительные возможности), установите в ней флажок **Context-sensitive Help** (Контекстная справка), установите ставший при этом доступным переключатель в положение **HTML Help format** (Справка формата HTML) и только после этого нажмите кнопку **Finish** (Готово).
3. Внесите в текст файла HTML.rc изменения, аналогичные тем, которые были внесены в файл Help.rc для его русификации.

Примечание

Текст файлов будет практически идентичен, за исключением имен файлов и идентификаторов ресурсов.

4. Скопируйте файлы курсоров из подкаталога res каталога проекта Help в одноименный подкаталог каталога проекта HTML.
5. В редакторе ресурсов создайте те же команды меню и тот же ресурс диалогового окна, что и в приложении Help.
6. Создайте заготовки функций обработки сообщений о вызове новых команд меню, как это было сделано при создании приложения Help.
7. Скопируйте файлы DemoDlg.cpp и DemoDlg.h из каталога приложения Help в каталог данного приложения.
8. В окне **Solution Explorer** (Проводник решения) или **Class View** (Просмотр классов) выделите имя проекта и выберите команду меню **Project | Add Existing Item** (Добавить существующий элемент). Появится диалоговое окно **Add Existing Item - HTML** (Добавить существующий элемент), изображенное на рис. 13.37.
9. В окне списка выделите имена файлов DemoDlg.cpp и DemoDlg.h и нажмите кнопку **Open** (Открыть). Эти файлы будут добавлены в соответствующие папки окна **Solution Explorer** (Проводник приложения).

На этом первый этап преобразования проекта приложения HTML можно считать завершенным.

Чтобы преобразовать проект справочной системы приложения Help в формат HTML Help:

1. Выберите на Панели задач команду меню **Start | Run** (Пуск | Выполнить). Появится диалоговое окно **Run** (Выполнить).
2. Нажмите кнопку **Browse** (Просмотр). Откроется диалоговое окно **Browse** (Просмотр).

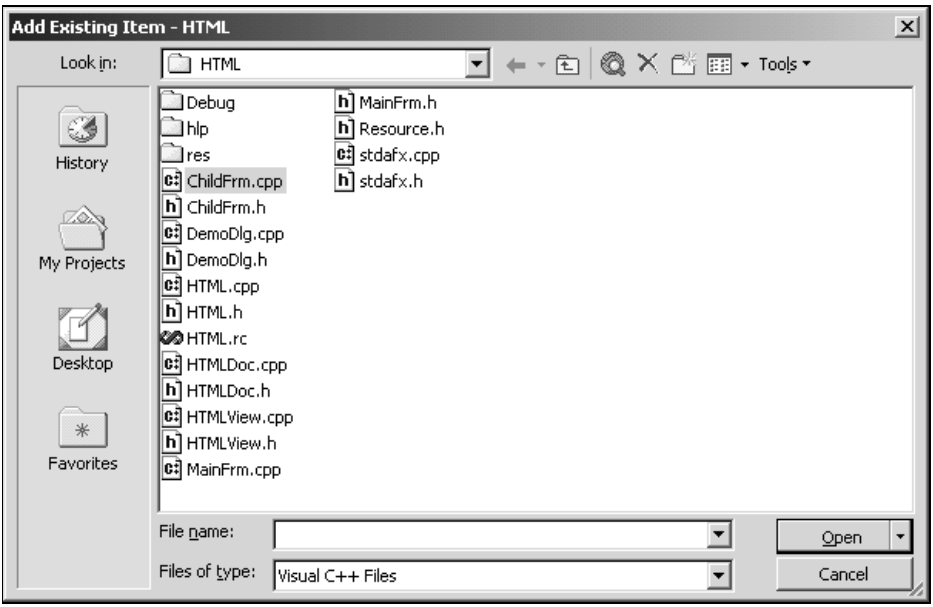


Рис. 13.37. Диалоговое окно **Add Existing Item - HTML**

3. Найдите исполняемый файл hhw.exe (при установке по умолчанию он располагается в папке C:\Program Files\Microsoft Visual Studio.NET\Visual Studio SDKs\HTML Help 1.3 SDK\workshop), выделите его и нажмите кнопку **Open** (Открыть). Имя данного файла появится в текстовом поле **Open** (Открыть) диалогового окна **Run** (Выполнить).
4. Нажмите кнопку **OK**. Откроется приложение HTML Help Workshop.
5. Выберите команду меню **File | New** (Файл | Создать) или нажмите кнопку **New** (Создать) на панели инструментов. Появится диалоговое окно **New** (Создать), изображенное на рис. 13.38.

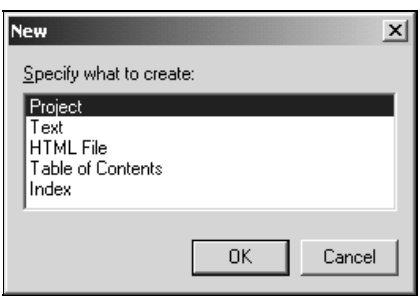


Рис. 13.38. Диалоговое окно **New**

6. В окне списка **Specify what to create** (Что нужно создать) оставьте выделенной строку **Project** (Проект) и нажмите кнопку **OK**. Появится диалоговое окно **New Project** (Новый проект) и приложение HTML Help Workshop примет вид, изображенный на рис. 13.39.

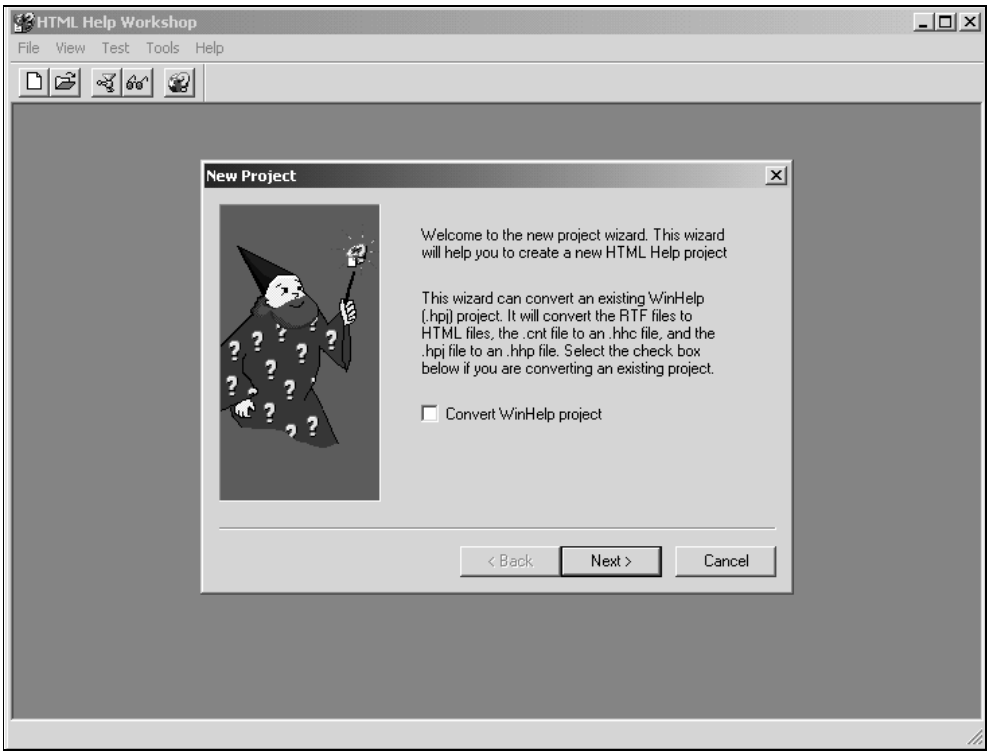


Рис. 13.39. Первая вкладка мастера New Project

7. Установите флажок **Convert WinHelp project** (Преобразовать проект WinHelp) и нажмите кнопку **Next** (Далее). Диалоговое окно **New Project** (Новый проект) примет вид, изображенный на рис. 13.40.
8. Нажмите кнопку **Browse** (Просмотр), расположенную под текстовым полем **Specify where your WinHelp project file is located** (Укажите путь к файлу проекта WinHelp). Появится диалоговое окно **Open** (Открыть), изображенное на рис. 13.41.
9. Раскройте в этом диалоговом окне папку hlp приложения Help, выделите в ней значок **Help** (Помощь) и нажмите кнопку **Open** (Открыть). Путь к этому файлу появится в соответствующем текстовом поле.
10. Нажмите кнопку **Browse** (Просмотр), расположенную под текстовым полем **Specify the name of the project (.hhp) file and where you would like to be created**

(Введите имя и путь к создаваемому файлу проекта). Появится диалоговое окно **Open** (Открыть).

Рис. 13.40. Вторая вкладка мастера New Project

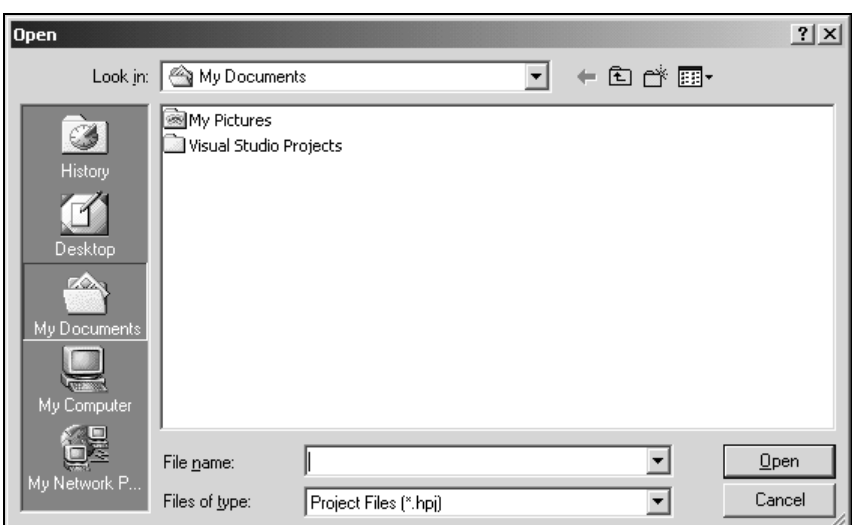


Рис. 13.41. Диалоговое окно **Open**

11. Откройте в нем папку hlp приложения HTML, выделите в ней значок **Help** (Помощь) и нажмите кнопку **Open** (Открыть). Путь к этому файлу появится в соответствующем текстовом поле.

12. Нажмите кнопку **Next** (Далее) в диалоговом окне **New Project** (Новый проект). Появится окно сообщения, предупреждающее о том, что этот файл уже существует и запрашивающее разрешение на его замену новым файлом.
13. Нажмите кнопку **Yes** (Да). Диалоговое окно **New Project** (Новый проект) примет вид, изображенный на рис. 13.42.

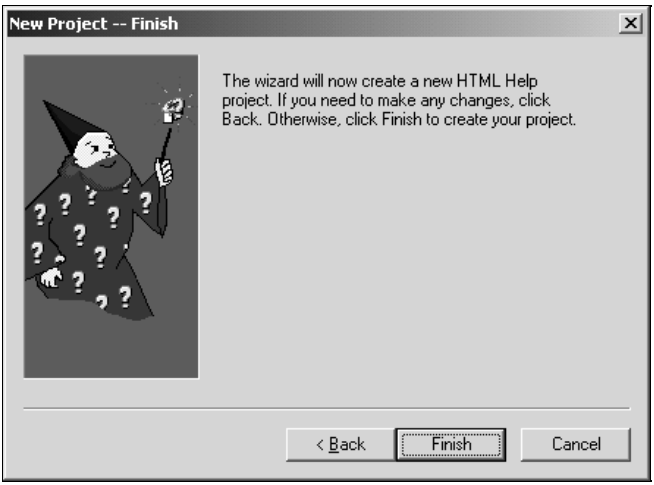


Рис. 13.42. Третья вкладка мастера New Project

14. Нажмите кнопку **Finish** (Готово). В приложении HTML будет создан новый проект справочной системы, а приложение HTML Help Workshop примет вид, изображенный на рис. 13.43.
15. Дважды щелкните левой кнопкой мыши на строке `html\afxc08.htm`. Откроется окно редактирования этого файла, как это показано на рис. 13.44.
16. Замените в нем текст "Windows-1252" текстом "Windows-1251".

Примечание

Возможно существуют более быстрые и корректные способы русификации текстов файлов HTML, возможно, они, даже будут включены в среду программирования Visual Studio.NET, однако пока их нет, следуйте этим инструкциям.

17. Повторите эту операцию для всех остальных файлов HTML. Если в строке `<Title>` вместо (`Untitled`) будут стоять вопросительные знаки, найдите эту тему в исходном файле rtf и восстановите ее заголовок.

Примечание

Восстанавливать ключевые слова нет необходимости, поскольку индекс проекта справки уже создан. Однако для очистки совести можно заменить знаки вопроса в строке `Context` текстом ключа соответствующей справки.

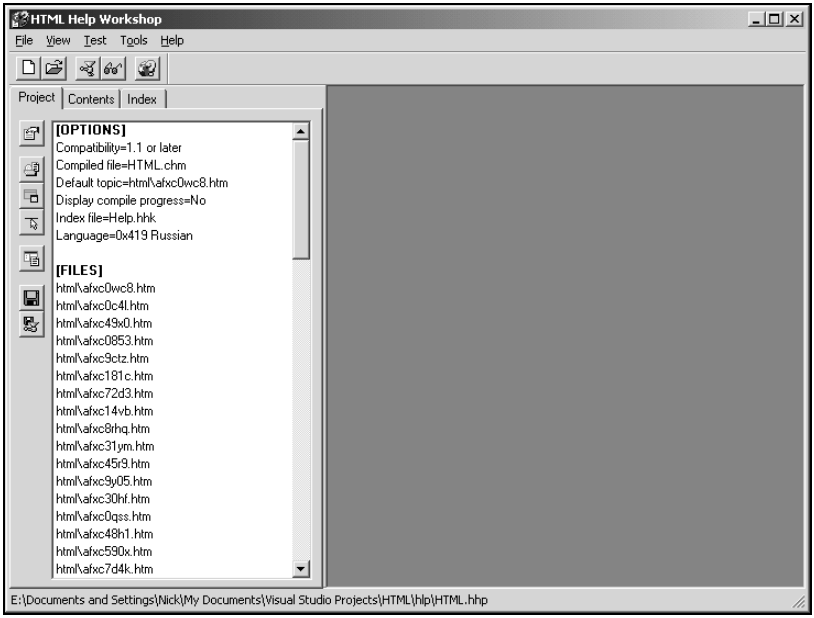


Рис. 13.43. Приложение HTML Help Workshop

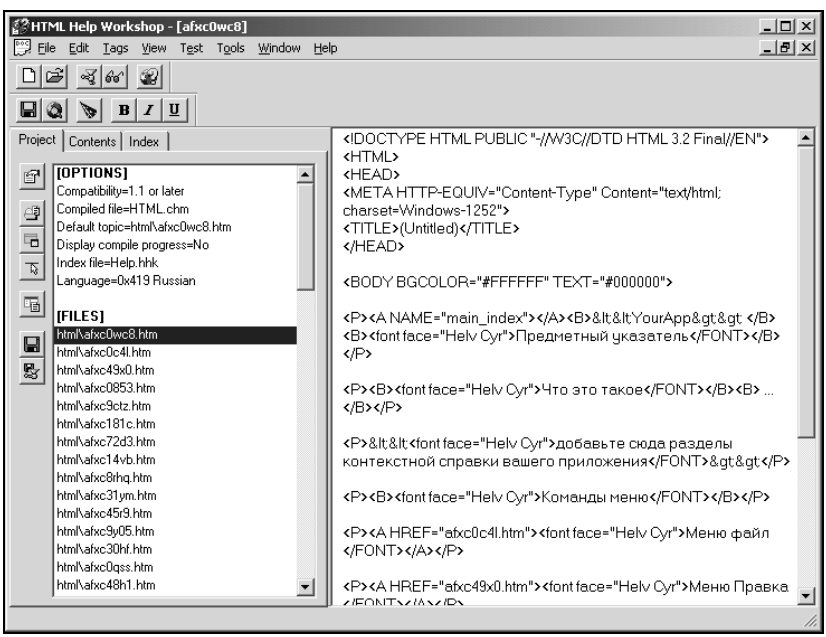


Рис. 13.44. Приложение HTML Help Workshop. Редактирование файла

Примечание

Настоятельно рекомендуется создать заголовки даже для тех тем, которые их не имели. Это существенно облегчит поиск этих тем при создании оглавления.

18. Нажмите кнопку **Change project options** (Изменение параметров проекта) на панели инструментов, расположенной слева от вкладки **Project** (Проект). Появится диалоговое окно **Options** (Опции), изображенное на рис. 13.45.

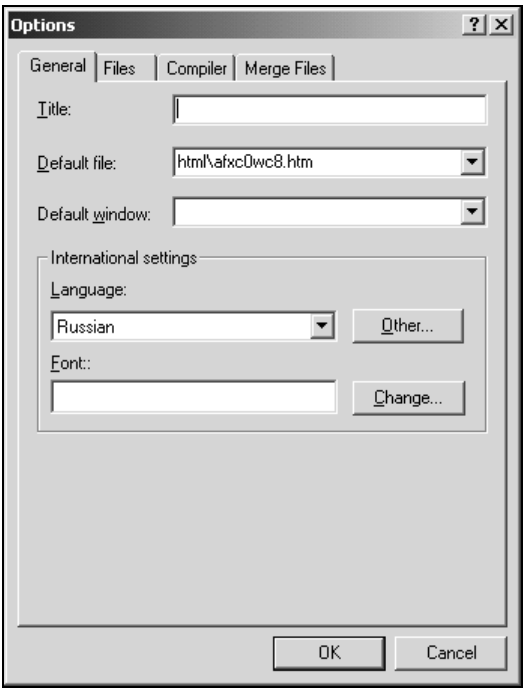


Рис. 13.45. Диалоговое окно **Options**

19. Раскройте вкладку **Compiler** (Компилятор), изображенную на рис. 13.46, установите в ней флажок **Compile full-text search information** (Компилировать поисковую информацию для всего текста) и нажмите кнопку **OK**. Диалоговое окно **Options** (Опции) закроется.
20. Попробуйте раскрыть вкладку **Contents** (Содержание). Появится диалоговое окно **Table of Contents Not Specified** (Отсутствует оглавление), изображенное на рис. 13.47 и предлагающее создать новый файл оглавления или воспользоваться одним из существующих.
21. Оставьте переключатель в его исходном положении и нажмите кнопку **OK**. Появится диалоговое окно **Save As** (Сохранить как), изображенное на рис. 13.48.

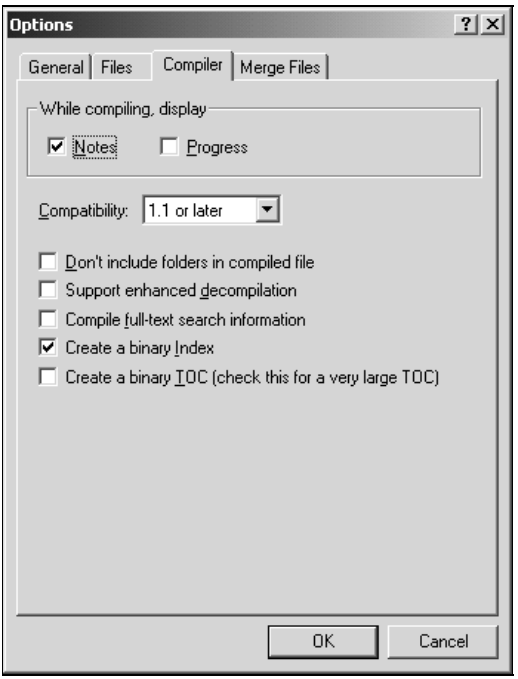


Рис. 13.46. Диалоговое окно **Options**, вкладка **Compiler**

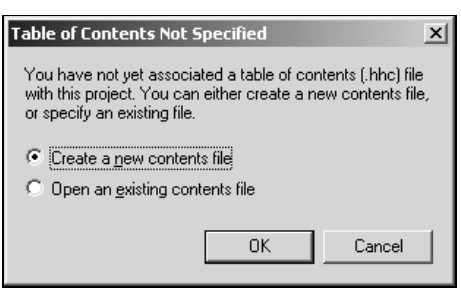


Рис. 13.47. Диалоговое окно **Table of Contents Not Specified**

22. Выделите значок HTML и нажмите кнопку **Save** (Сохранить). Появится окно сообщения о том, что данный файл уже существует. Нажмите кнопку **Yes** (Да). Раскроется пустая вкладка **Contents** (Содержание), как это показано на рис. 13.49.
23. Нажмите кнопку **Insert a page** (Вставка страницы) на панели инструментов, расположенной справа от вкладки (на этой кнопке изображен лист бумаги). Появится диалоговое окно **Table of Contents Entry** (Пункт оглавления), изображенное на рис. 13.50.

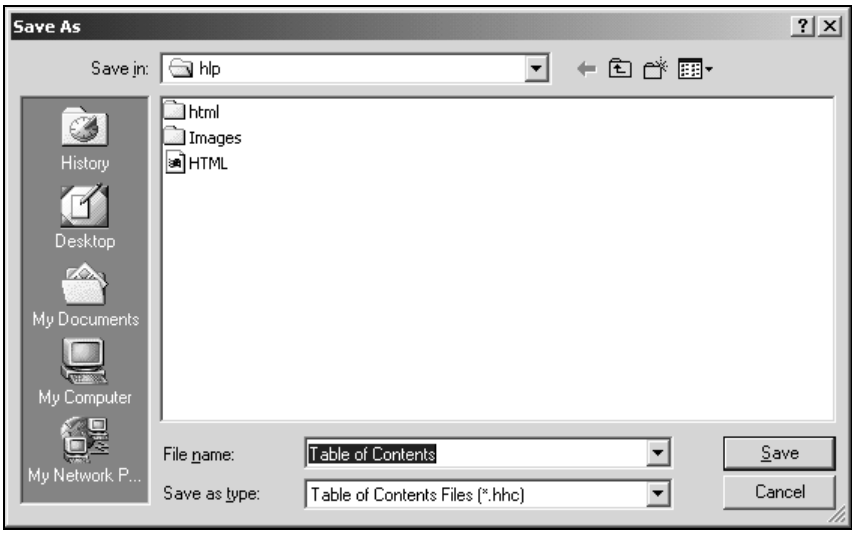


Рис. 13.48. Диалоговое окно **Save As**

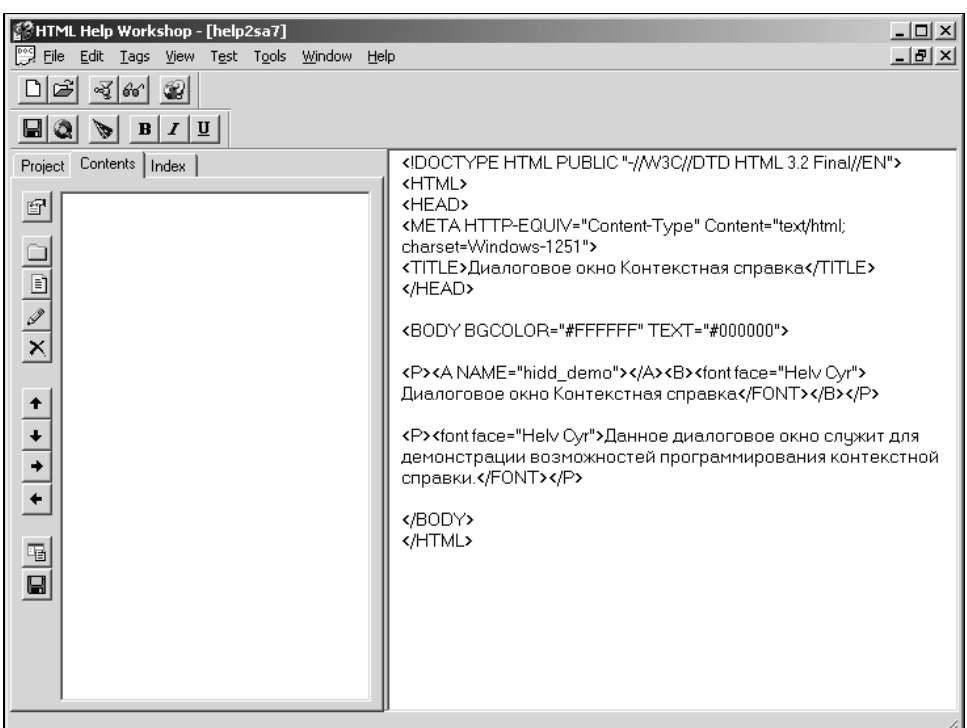


Рис. 13.49. Вкладка **Contents**

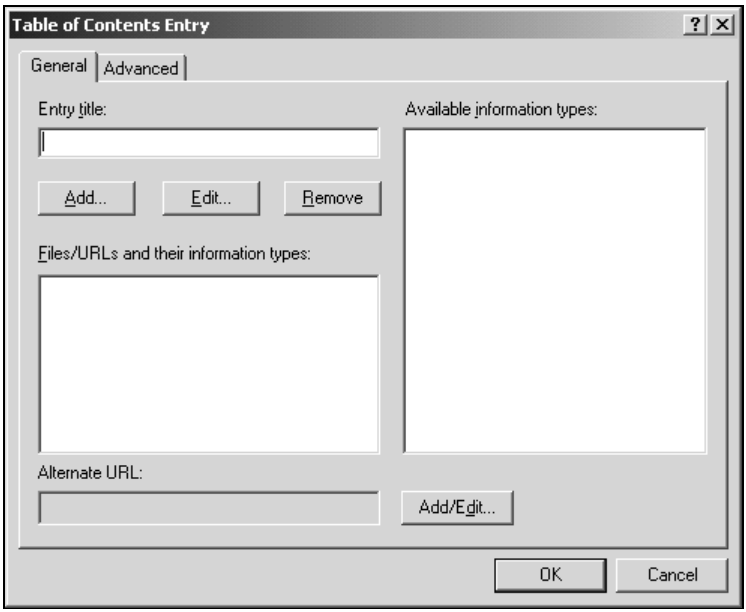


Рис. 13.50. Диалоговое окно **Table of Contents Entry**

Примечание

Для того чтобы внесенные в файлы HTML изменения были бы сохранены в проекте и в дальнейшем выводились бы их русифицированные заголовки, целесообразно на этом этапе закрыть редактируемый проект, а затем снова его открыть.

24. Введите в текстовое поле **Entry title** (Введите заголовок) текст пункта оглавления "<<Your App>> Предметный указатель" и нажмите кнопку **Add | Edit** (Добавить | Правка). Появится диалоговое окно **Path or URL** (Путь или URL), изображенное на рис. 13.51.
25. В окне списка **HTML titles** (Заголовки HTML) выделите заголовок **Предметный указатель** (или другой заголовок, назначенный вами данной теме) и нажмите кнопку **OK**. В окне списка **Files/URLs and their information types** (Файлы/URL и их типы) диалогового окна **Table of Contents Entry** (Пункт оглавления) появится путь к соответствующему файлу HTML.
26. Нажмите кнопку **OK**. В оглавление будет добавлен новый пункт.
27. На панели инструментов вкладки **Contents** (Содержание) нажмите кнопку **Insert a heading** (Вставка заголовка). На этой кнопке изображена папка. Появится диалоговое окно, запрашивающее о том, следует ли поместить этот заголовок в начало оглавления.
28. Нажмите кнопку **No** (Нет). Появится диалоговое окно **Table of Contents Entry** (Пункт оглавления).

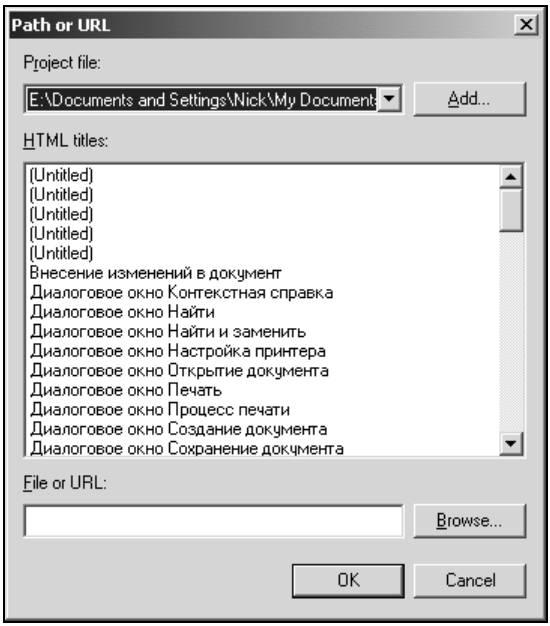


Рис. 13.51. Диалоговое окно **Path or URL**

29. Введите в текстовое поле **Entry title** (Введите заголовок) заголовок "Меню" и нажмите кнопку **OK**. Во вкладке **Contents** (Содержание) появится раскрытая папка с соответствующим именем.
30. Повторите п.п. 21—24 для включения в оглавление пункта меню "Меню Файл" (сопоставив ему соответствующий файл HTML).
31. Повторите п.п. 21—24 для включения в оглавление пункта меню "Меню Правка" (сопоставив ему соответствующий файл HTML).
32. Повторите п.п. 21—24 для включения в оглавление пункта меню "Меню Вид" (сопоставив ему соответствующий файл HTML).
33. Повторите п.п. 21—24 для включения в оглавление пункта меню "Меню Демо" (сопоставив ему соответствующий файл HTML).
34. Повторите п.п. 21—24 для включения в оглавление пункта меню "Меню Окно" (сопоставив ему соответствующий файл HTML).
35. Повторите п.п. 21—24 для включения в оглавление пункта меню "Меню Справка" (сопоставив ему соответствующий файл HTML).
36. Повторите п.п. 21—24 для включения в оглавление пункта меню "Строка состояния" (сопоставив ему соответствующий файл HTML).
37. На панели инструментов вкладки **Contents** (Содержание) нажмите кнопку **Move selection left** (Сдвинуть выделенный фрагмент влево). На этой кнопке

изображена стрелка влево. При этом выделенный пункт меню будет передвинут влево на верхний уровень иерархии.

38. Повторите п.п. 21—24 для включения в оглавление пункта меню "Панель инструментов" (сопоставив ему соответствующий файл HTML). В результате выполненных действий окно приложения HTML Help Workshop примет вид, изображенный на рис. 13.52.

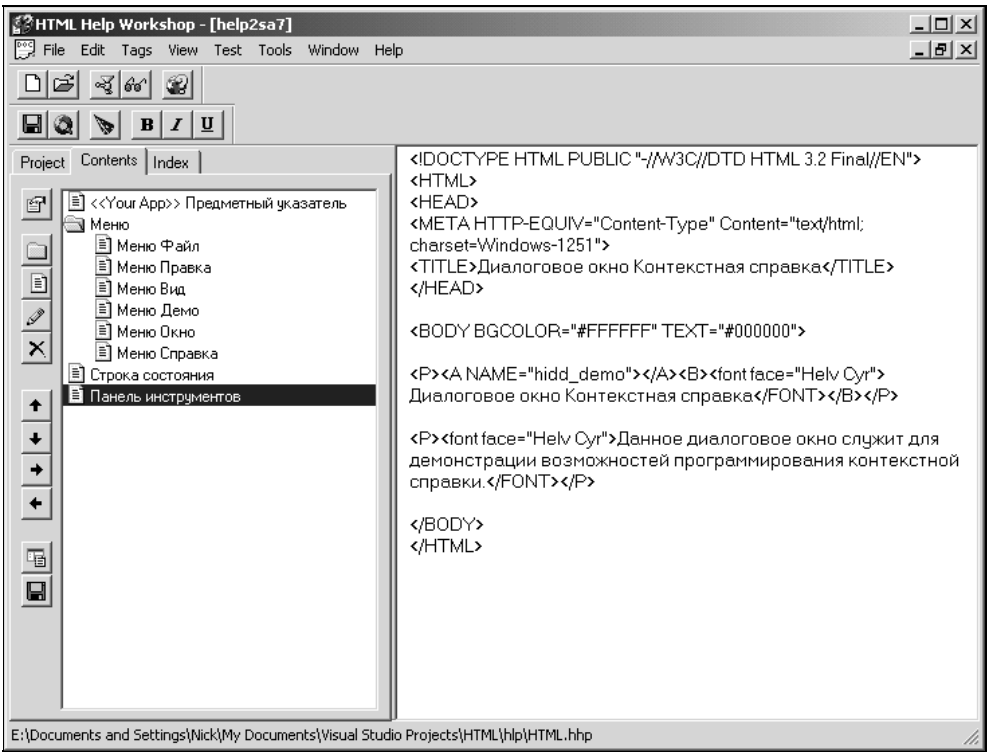


Рис. 13.52. Оглавление

39. Закройте приложение HTML Help Workshop, сохранив все измененные в нем файлы.

Теперь необходимо согласовать проект приложения HTML с преобразованным для него проектом справочной системы. Для этого:

1. Вернитесь в среду программирования Visual Studio.NET.
2. В окне списка **Class View** (Просмотр классов) раскройте папку **CHTMLView** и дважды щелкните левой кнопкой мыши на имени функции обработки сообщения `OnNewHelp`. Откроется окно редактирования файла HelpView.cpp и текстовый курсор установится в тексте функции `OnNewHelp`.

3. Измените функцию `OnNewHelp` в соответствии с листингом 13.5.

Листинг 13.5. Вызов справочной системы

```
// Функции обработки сообщений класса CHTMLView

// Вызов новой справки
void CHTMLView::OnNewHelp(void)
{
  HWND  hWnd = AfxGetApp()->m_pMainWnd->GetSafeHwnd();
  ::HtmlHelp(hWnd,
    "E:\\Documents and Settings\\Nick\\My Documents\\Visual Studio Proj-
ects\\HTML\\hlp\\HTML.chm::/html\\help7f5c.htm",
      HH_DISPLAY_TOPIC, NULL);
}
```

4. Измените функцию `OnDemoDialog` в соответствии с листингом 13.6.

Листинг 13.6. Вызов диалогового окна

```
// Вывод демонстрационного диалогового окна
void CHelpView::OnDemoDialog(void)
{
  CDemoDlg  dlg(this);
  dlg.DoModal();
}
```

5. В начале данного файла после строки `#include "HTMLView.h"` поместите строку

   ```
   #include "DemoDlg.h"
   ```

6. Откройте окно редактирования файла DemoDlg.h и удалите из него строки

   ```
   #define HIDD_CONTEXT    2
   #define HIDD_CONTEXT_OK  3
   #define HIDD_CONTEXT_CANCEL  4
   ```

7. Откройте окно редактирования файла DemoDlg.cpp и замените в нем строку `#include "Help.h"` строкой

   ```
   #include "HTML.h"
   ```

8. Измените текст функций обработки сообщений и массива соответствия идентификаторов ресурсов идентификаторам контекстной справки в соответствии с листингом 13.7.

Листинг 13.7. Вывод контекстной справки

```
// Функции обработки сообщений класса CDemoDlg

// Вывод контекстной справки
BOOL CDemoDlg::OnHelpInfo(HELPINFO* lpHelpInfo)
{
  if(lpHelpInfo->iContextType == HELPINFO_WINDOW)
  {
    return ::HtmlHelp((HWND)lpHelpInfo->hItemHandle,
      "E:\\Documents and Settings\\Nick\\My Documents\\Visual Studio Proj-
ects\\HTML\\hlp\\HTML.chm::/ContextHelp.txt",
        HH_TP_HELP_WM_HELP, (DWORD)(LPVOID)aHelpIDs) != NULL;
  }

  return TRUE;
}

void CDemoDlg::OnContextMenu(CWnd* pWnd, CPoint /*point*/)
{
  ::HtmlHelp(pWnd->GetSafeHwnd(),
    "E:\\Documents and Settings\\Nick\\My Documents\\Visual Studio Proj-
ects\\HTML\\hlp\\HTML.chm::/ContextHelp.txt",
      HH_TP_HELP_CONTEXTMENU, (DWORD)(LPVOID)aHelpIDs);
}
```

9. Выберите команду меню **File | New | File** (Файл | Создать | Файл) или нажмите комбинацию клавиш <Ctrl>+<N>, появится диалоговое окно **New File** (Создать файл), изображенное на рис. 13.53.
10. В окне списка **Templates** (Шаблоны) выделите значок **Text File** (Текстовый файл) и нажмите кнопку **Open** (Открыть). Откроется пустое окно редактирования текстового файла.
11. Поместите в этот файл приведенный ниже текст.

```
.topic 1
Данная кнопка служит исключительно для того, чтобы вывести этот текст
.topic 2
Данная кнопка служит для выхода из диалогового окна Контекстная справка
.topic 3
Данная кнопка служит для выхода из диалогового окна Контекстная справка
```

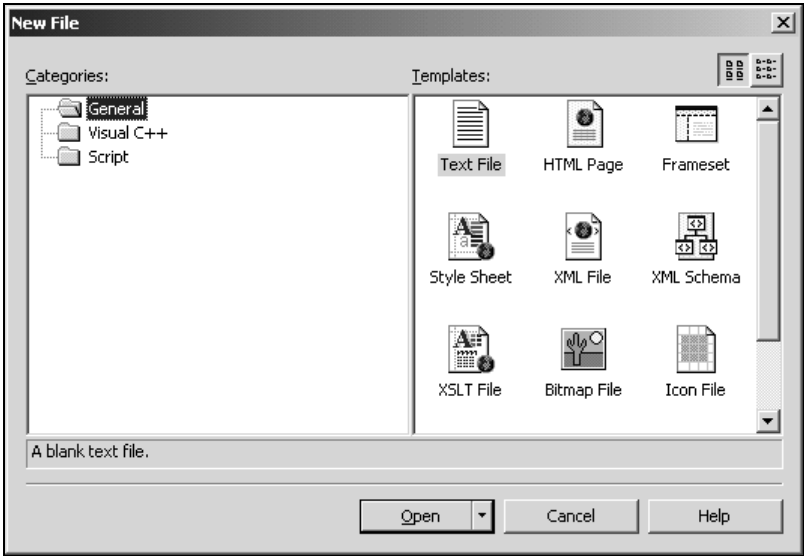


Рис. 13.53. Диалоговое окно **New File**

12. Выберите команду меню **File | Save TextFile1 As** (Файл | Сохранить файл TextFile1 как) и, с использованием появившегося диалогового окна **Save File As** (Сохранить файл как), сохраните его под именем ContextHelp.txt в каталоге hlp вашего приложения.
13. В окне **Solution Explorer** (Проводник решения) раскройте папку **HTML Help Files** (Файлы справки HTML) и дважды щелкните левой кнопкой мыши на значке **HTML.hhp**. Откроется окно редактирования соответствующего файла.
14. Добавьте в его конец текст, приведенный в листинге 13.8.

Листинг 13.8. Добавление в файл HTML.hhp

```
ContextHelp.txt

[ALIAS]
HIDR_MAINFRAME = html\afxc0wc8.htm
HIDR_HTML_newTYPE = html\afxc31id.htm
main_index        = html\afxc0wc8.htm
afx_hidd_color      = html\afxc465u.htm
afx_hidd_fileopen      = html\afxc31ym.htm
afx_hidd_filesave      = html\afxc48h1.htm
afx_hidd_find        = html\afxc2nac.htm
afx_hidd_font        = html\afxc369g.htm
```

```
afx_hidd_newtypedlg      = html\afxc14vb.htm
afx_hidd_replace      = html\afxc8asl.htm
AFX_HIDP_DEFAULT      = html\afxc9b3o.htm
afx_hidw_dockbar_top      = html\afxc78s0.htm
afx_hidw_status_bar      = html\afxc72ya.htm
afx_hidw_toolbar      = html\afxc9jci.htm
hid_app_about        = html\afxc9vp0.htm
hid_app_exit        = html\afxc7d4k.htm
hid_context_help      = html\afxc9ysw.htm
hid_edit_clear      = html\afxc214i.htm
hid_edit_clear_all   = html\afxc2fsc.htm
hid_edit_copy        = html\afxc7yt5.htm
hid_edit_cut        = html\afxc3ewk.htm
hid_edit_find        = html\afxc4ajo.htm
hid_edit_paste      = html\afxc67ad.htm
hid_edit_redo        = html\afxc1cof.htm
hid_edit_repeat      = html\afxc56lw.htm
hid_edit_replace      = html\afxc121x.htm
hid_edit_undo        = html\afxc8zxr.htm
hid_file_close      = html\afxc45r9.htm
hid_file_mru_file1      = html\afxc590x.htm
hid_file_new        = html\afxc72d3.htm
hid_file_open        = html\afxc8rhq.htm
hid_file_save        = html\afxc9y05.htm
hid_file_save_as      = html\afxc30hf.htm
hid_file_send_mail      = html\afxc0qss.htm
hid_help_index      = html\afxc85dk.htm
hid_help_using      = html\afxc1tk7.htm
hid_ht_caption      = html\afxc4r72.htm
hid_ht_nowhere      = html\afxc5zol.htm
hid_next_pane        = html\afxc0l7p.htm
hid_prev_pane        = html\afxc48o5.htm
hid_sc_close        = html\afxc0085.htm
hid_sc_maximize      = html\afxc6ztx.htm
hid_sc_minimize      = html\afxc1vqd.htm
hid_sc_move        = html\afxc57j9.htm
hid_sc_nextwindow      = html\afxc33uf.htm
hid_sc_prevwindow      = html\afxc6t2v.htm
hid_sc_restore      = html\afxc5tr9.htm
hid_sc_size        = html\afxc8eed.htm
```

```
hid_sc_tasklist      = html\afxc1sj8.htm
hid_view_ruler      = html\afxc71o2.htm
hid_view_status_bar      = html\afxc6tv6.htm
hid_view_toolbar      = html\afxc8ble.htm
hid_window_all      = html\afxc9sz0.htm
hid_window_arrange      = html\afxc4mn9.htm
hid_window_cascade      = html\afxc9u5h.htm
hid_window_new      = html\afxc0xpz.htm
hid_window_split      = html\afxc5ph0.htm
hid_window_tile      = html\afxc3zz9.htm
hid_window_tile_horz      = html\afxc3at6.htm
hid_window_tile_vert      = html\afxc1304.htm
hidr_doc1type        = html\afxc31id.htm
menu_edit        = html\afxc49x0.htm
menu_file        = html\afxc0c4l.htm
menu_help        = html\afxc181c.htm
menu_view        = html\afxc0853.htm
menu_window        = html\afxc9ctz.htm
scrollbars        = html\afxc1g4z.htm
afx_hidd_print      = html\afxp5hf8.htm
afx_hidd_printdlg      = html\afxp86av.htm
afx_hidd_printsetup      = html\afxp0vjk.htm
afx_hidw_preview_bar      = html\afxp3i9e.htm
hid_file_page_setup      = html\afxp42b4.htm
hid_file_print      = html\afxp225w.htm
hid_file_print_preview    = html\afxp7q2f.htm
hid_file_print_setup      = html\afxp0434.htm
HIDD_ABOUTBOX        = html\afxc9vp0.htm
HID_HT_SIZE        = html\afxc8eed.htm
HID_HT_HSCROLL      = html\afxc1g4z.htm
HID_HT_VSCROLL      = html\afxc1g4z.htm
HID_HT_MINBUTTON      = html\afxc1vqd.htm
HID_HT_MAXBUTTON      = html\afxc6ztx.htm
AFX_HIDP_INVALID_FILENAME    = html\afxc0wc8.htm
AFX_HIDP_FAILED_TO_OPEN_DOC    = html\afxc0wc8.htm
AFX_HIDP_FAILED_TO_SAVE_DOC    = html\afxc0wc8.htm
AFX_HIDP_ASK_TO_SAVE      = html\afxc0wc8.htm
AFX_HIDP_FAILED_TO_CREATE_DOC  = html\afxc0wc8.htm
AFX_HIDP_FILE_TOO_LARGE    = html\afxc0wc8.htm
AFX_HIDP_FAILED_TO_START_PRINT  = html\afxc0wc8.htm
```

```
AFX_HIDP_FAILED_TO_LAUNCH_HELP  = html\afxc0wc8.htm
AFX_HIDP_INTERNAL_FAILURE    = html\afxc0wc8.htm
AFX_HIDP_COMMAND_FAILURE    = html\afxc0wc8.htm
AFX_HIDP_PARSE_INT      = html\afxc0wc8.htm
AFX_HIDP_PARSE_REAL      = html\afxc0wc8.htm
AFX_HIDP_PARSE_INT_RANGE    = html\afxc0wc8.htm
AFX_HIDP_PARSE_REAL_RANGE    = html\afxc0wc8.htm
AFX_HIDP_PARSE_STRING_SIZE    = html\afxc0wc8.htm
AFX_HIDP_FAILED_INVALID_FORMAT  = html\afxc0wc8.htm
AFX_HIDP_FAILED_INVALID_PATH  = html\afxc0wc8.htm
AFX_HIDP_FAILED_DISK_FULL    = html\afxc0wc8.htm
AFX_HIDP_FAILED_ACCESS_READ    = html\afxc0wc8.htm
AFX_HIDP_FAILED_ACCESS_WRITE  = html\afxc0wc8.htm
AFX_HIDP_FAILED_IO_ERROR_READ  = html\afxc0wc8.htm
AFX_HIDP_FAILED_IO_ERROR_WRITE  = html\afxc0wc8.htm
AFX_HIDP_STATIC_OBJECT    = html\afxc0wc8.htm
AFX_HIDP_FAILED_TO_CONNECT    = html\afxc0wc8.htm
AFX_HIDP_SERVER_BUSY      = html\afxc0wc8.htm
AFX_HIDP_BAD_VERB      = html\afxc0wc8.htm
AFX_HIDP_FAILED_MEMORY_ALLOC  = html\afxc0wc8.htm
AFX_HIDP_FAILED_TO_NOTIFY    = html\afxc0wc8.htm
AFX_HIDP_FAILED_TO_LAUNCH    = html\afxc0wc8.htm
AFX_HIDP_ASK_TO_UPDATE    = html\afxc0wc8.htm
AFX_HIDP_FAILED_TO_UPDATE    = html\afxc0wc8.htm
AFX_HIDP_FAILED_TO_REGISTER    = html\afxc0wc8.htm
AFX_HIDP_FAILED_TO_AUTO_REGISTER  = html\afxc0wc8.htm
AFX_HIDW_DOCKBAR_BOTTOM    = html\afxc78s0.htm
AFX_HIDW_DOCKBAR_LEFT    = html\afxc78s0.htm
AFX_HIDW_DOCKBAR_RIGHT    = html\afxc78s0.htm

[MAP]
#include HTMLDefines.h

[INFOTYPES]
```

Примечание

Этот текст может быть получен путем копирования соответствующего фрагмента файла проекта справочной системы HTML для приложения, использующего английский язык. При этом имена виртуальных файлов, указанные в разделах файлов afxcore.htm и afxprint.htm, должны быть заменены именами соответствующих файлов, созданных приложением HTML Help Workshop.

15. Нажмите клавишу <F5> и запустите приложение на исполнение. Появится окно приложения, изображенное на рис. 13.54.

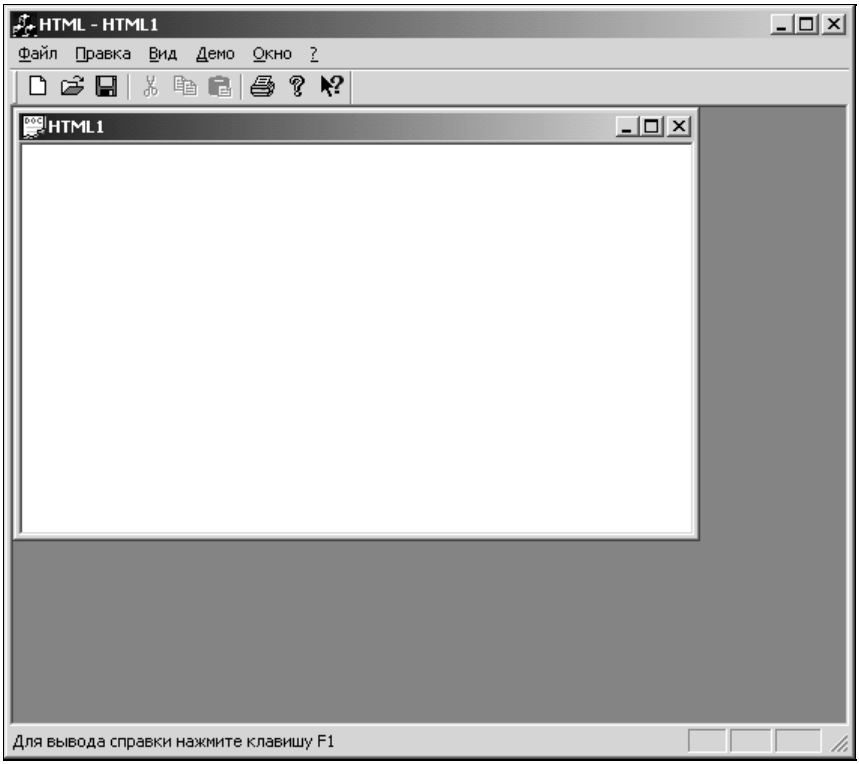


Рис. 13.54. Окно приложения

16. Выберите команду меню **? | Вызов справки**. Появится окно справочной системы приложения, приведенное на рис. 13.55. Принципы работы с этим окном были описаны в начале данной главы.
17. Убедитесь в работоспособности этого окна и закройте его.
18. Выберите команду меню **? | Новая справка**. Появится окно справочной системы приложения, содержащее справку по новым возможностям приложения, как это показано на рис. 13.56.
19. Закройте это окно и нажмите на панели инструментов кнопку **Справка**. Указатель мыши примет вид наклонной стрелки со знаком вопроса. Щелкните левой кнопкой мыши на заголовке окна документа. Появится окно справочной системы с контекстной справкой по данному вопросу, как это показано на рис. 13.57.

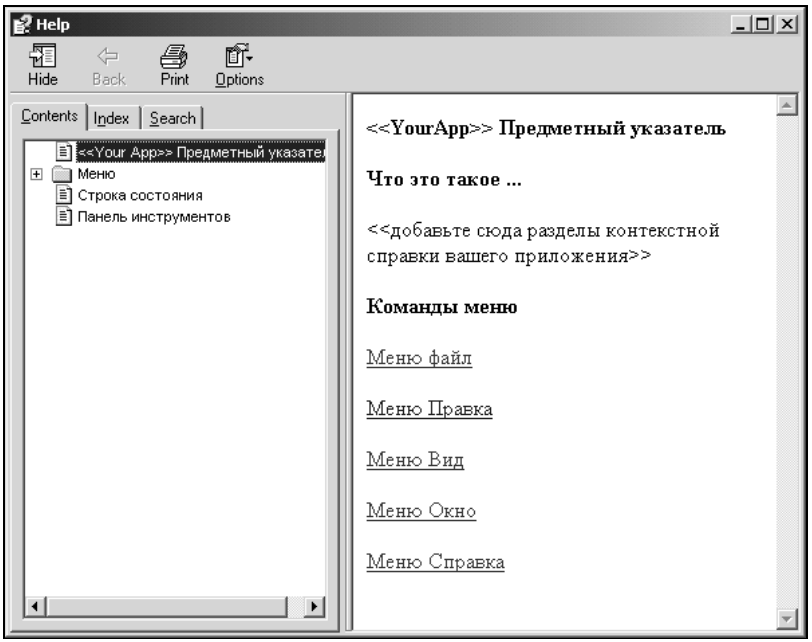


Рис. 13.55. Окно **Help**

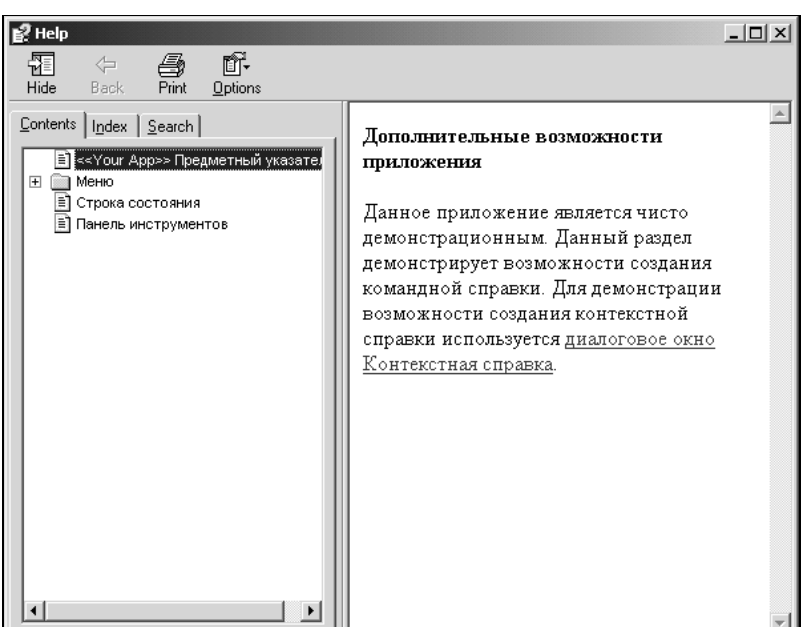


Рис. 13.56. Вывод справки по запросу

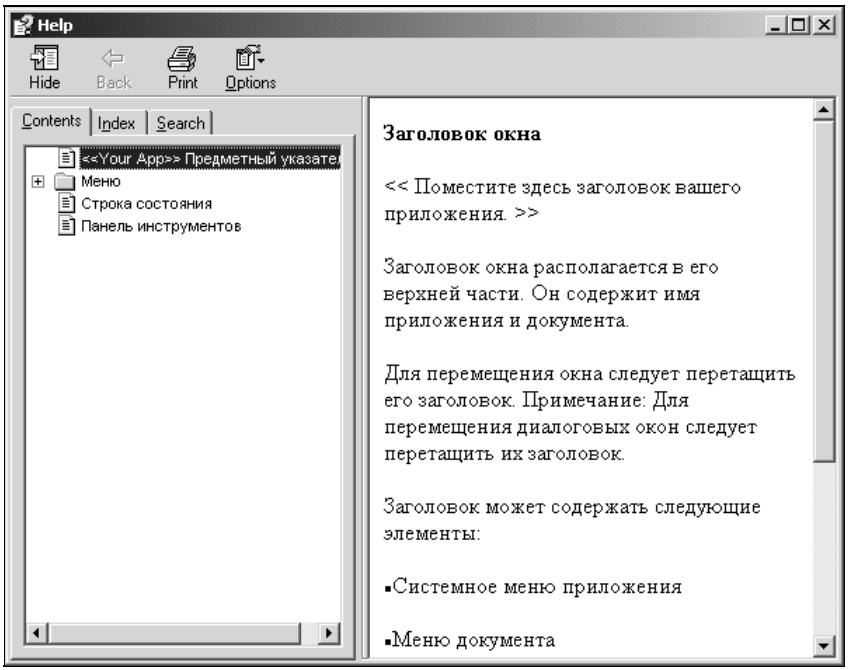


Рис. 13.57. Контекстная справка по стандартному запросу

20. Закройте это окно и выберите команду меню **Демо | Вызов**. Появится диалоговое окно **Контекстная справка**.
21. В заголовке диалогового окна нажмите кнопку **Help** (Справка). Указатель мыши примет вид наклонной стрелки со знаком вопроса. Щелкните левой кнопкой мыши на кнопке **Демо**. Для нее будет выведена контекстная справка, показанная на рис. 13.58.
22. Закройте диалоговое окно **Контекстная справка** и завершите работу с приложением.

Как следует из вышеизложенного, преобразование справочной системы приложения в формат HTML представляет собой достаточно сложную задачу. Во-первых, при этом преобразовании три текстовых файла (а в стандартном случае два текстовых файла) разбиваются на множество файлов с загадочными именами. Интересно, что при создании приложения со справочной системой HTML в нее включаются только два стандартных файла, содержащие ту же информацию (кроме нескольких пользовательских тем). Во-вторых, при преобразовании проекта справочной системы не создается оглавления и не включается информация, необходимая для создания контекстной справки по стандартным элементам приложения. В-третьих, преобразованный проект не включает в себя поисковую систему, которую приходится включать пользователю, хотя это и не сложная операция. И, наконец в-четвертых, при преобразовании не учитывается язык,

использованный для создания справочной системы, что приводит к существенным проблемам при создании поисковой системы.

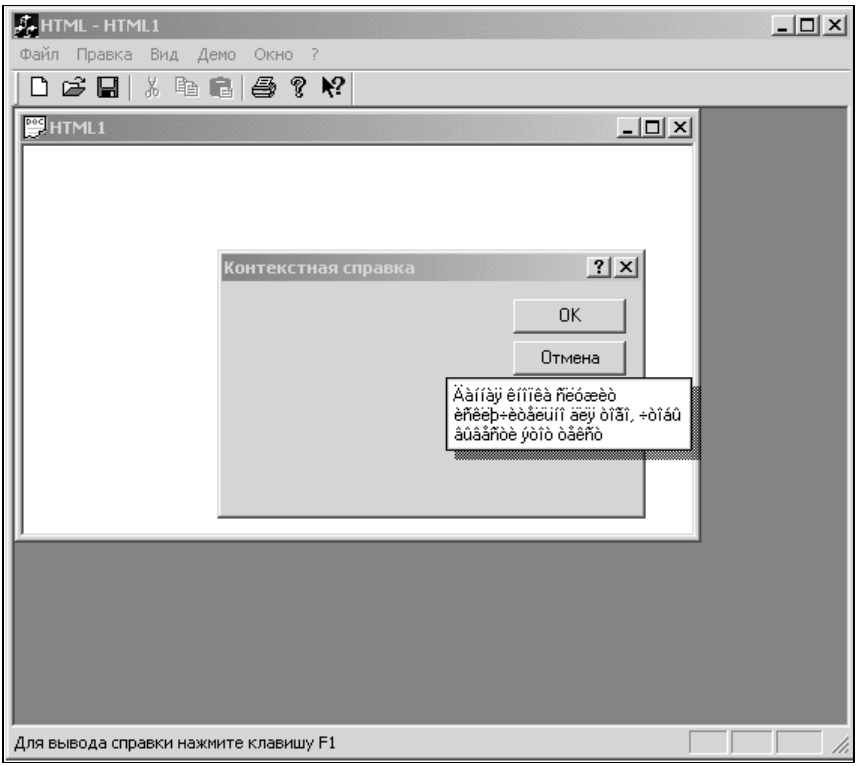


Рис. 13.58. Пользовательская контекстная справка

Исходя из вышеизложенного, представляется целесообразным не конвертировать русифицированные проекты, а русифицировать создаваемые мастером MFC Application Wizard проекты со справочной системой HTML.

Поскольку для создания демонстрационного проекта пришлось создавать множество промежуточных проектов, в нем были произведены минимально возможные изменения. При самостоятельной доработке конвертированного проекта, прежде всего, встанет вопрос о нахождении файлов, соответствующих конкретной теме. Это понадобится, например, при замене в разделе `[ALIAS]` файла проекта (hhp) виртуальных имен файлов именами преобразованных файлов.

Для этого полезно включить файлы HTML в приложение. Чтобы включить эти файлы в приложение:

1. В окне **Solution Explorer** (Проводник решения) щелкните правой кнопкой мыши на папке **HTML Help Topics** (Справочная система HTML) и в появившемся

контекстном меню выберите команду **Add | Add Existing Item** (Добавить | Добавить существующий элемент). Появится диалоговое окно **Add Existing Item - HTML** (Добавить существующий элемент), изображенное на рис. 13.59.

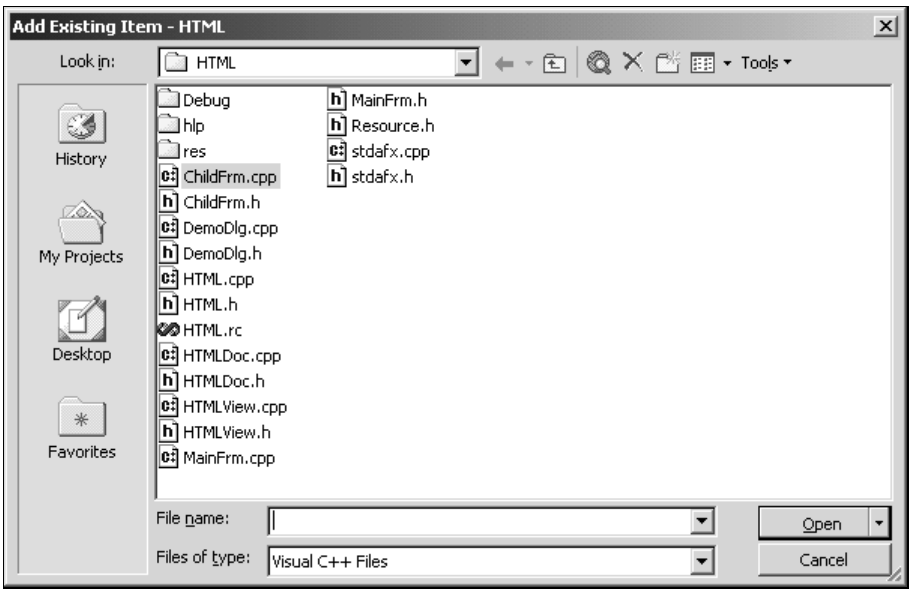


Рис. 13.59. Диалоговое окно **Add Existing Item - HTML**

2. В окне списка раскройте папку **\hlp**, а в ней — папку **\html**.
3. В раскрывающемся списке **Files of type** (Типы файлов) выделите строку **Native Web Files** (Файлы Web). В окне списка появятся файлы HTML.
4. Выделите все файлы в окне списка и нажмите кнопку **Open** (Открыть). Все выделенные файлы будут включены в приложение.

После этого содержимое этих файлов будет просматриваться при выборе команды меню **Edit | Find and Replace | Find in Files** (Правка | Поиск и замена | Поиск в файлах). Это позволит быстро установить соответствие характерных фрагментов текста (например, идентификаторов контекстной справки) в файлах afxcore.htm и afxprint.htm и в преобразованных файлах.

Рассмотрим теперь изменения, внесенные в само приложение HTML. Поскольку это приложение является копией приложения Help, и единственное различие между ними заключается в типе используемой справочной системы, перейдем сразу к описанию функций обработки сообщений, используемых при выводе справки.

Если при выводе обычной справки в приложении используется глобальная функция `WinHelp`, то для вывода справочной системы HTML используется глобальная функция `HtmlHelp`. Эти функции имеют между собой много общего.

Прежде всего, рассмотрим использование функции `::HtmlHelp` для вывода конкретной темы справки. Эта тема выводится при выборе команды меню **? | Новая справка**, сообщение о выборе которой обрабатывается функцией `CHTMLView::OnNewHelp`. Поскольку в аргументах функции обработки сообщения не передается дескриптор окна (эти функции, вообще, не имеют аргументов), а его значение требуется передать в первом аргументе функции `::HtmlHelp`, это значение получается с использованием функции `GetSafeHwnd`. Второй аргумент функции `::HtmlHelp` содержит полное имя скомпилированного файла справочной системы и имя файла HTML, содержащего требуемую справку. Для скомпилированного файла справочной системы указан полный путь (как это рекомендуется Microsoft). Учитывая длинные имена и разветвленную систему каталогов, используемую Microsoft, указание полного пути к файлу представляет собой далеко не тривиальную задачу. При распространении приложения это приведет к еще большим сложностям, если не поместить исполняемый файл приложения и файл справки в один каталог или не обеспечить возможность использования относительных путей. При указании имени файла HTML используется относительный путь из каталога справочной системы.

Третий аргумент функции `::HtmlHelp` определяет действие, выполняемое данной функцией. В нашем случае — это вывод раздела справочной системы. Четвертый аргумент этой функции здесь не используется.

Функция `::HtmlHelp` используется и для вывода контекстной справки. Поскольку контекстная справка может быть получена различными способами, для ее вывода в приложении перегружены две функции.

Функция `CWnd::OnContextMenu` вызывается при щелчке правой кнопкой мыши на элементе интерфейса. При этом перед выводом контекстной справки должно раскрыться контекстное меню с единственной командой **What's This?** (Что это такое?), но оно почему-то не появляется.

Единственным оператором перегруженной функции `OnContextMenu` является вызов функции `::HtmlHelp`. В данном случае значение первого аргумента функции `::HtmlHelp` получается из первого аргумента функции `OnContextMenu`. Имя скомпилированного файла справочной системы, указанное во втором аргументе данной функции, совпадает с именем, указанным в функции `OnNewHelp` (файл-то один), а имя файла справки отличается. Как ни странно, это обычный текстовый файл, что не позволяет указать в нем параметры используемых шрифтов и, самое главное, используемого в нем языка, что, как видно из рис. 13.58, привело к выводу псевдографики. Однако есть слабая надежда, что эта ошибка может быть исправлена в дальнейшем. Третий аргумент функции `::HtmlHelp` указывает на то, что будет выводиться контекстная справка. Четвертый аргумент данной функции содержит указатель на двумерный массив соответствия идентификаторов элементов управления идентификаторам контекстной справки. Поскольку эти идентификаторы используются только в указанном файле контекстной справки, Microsoft предлагает не определять специальных идентификаторов, а указывать непосредственные числовые значения. Если для данного элемента управления не нужно выводить контекстную справку, его идентификатору в этом массиве сопоставляется значение −1. Указанные числовые значения

используются в метке `.topic`, отмечающей в текстовом файле фрагмент, соответствующий данному элементу управления.

При всех других способах вызова контекстной справки вызывается функция `CWnd::OnHelpInfo`. В качестве аргумента данной функции передается указатель на объект структуры `HELPINFO`. В этом объекте структуры, прежде всего, проверяется значение переменной `iContextType`, чтобы удостовериться в том, что справка вызывается для окна, а не для команды меню. Если это так, вызывается функция `::HtmlHelp`, значение первого аргумента которой получается из переменной объекта структуры `HELPINFO`. Поскольку содержимое контекстной справки не зависит от способа ее вызова, второй аргумент данной функции совпадает с ее вторым аргументом при вызове функции `OnContextMenu`. Третий ее аргумент указывает на вывод контекстной справки, а четвертый — на массив соответствия идентификаторов.

Диалоговое окно *О программе*

При создании приложения с использованием мастера MFC Application Wizard во вкладке **User Interface Features** (Особенности интерфейса пользователя) по умолчанию установлен флажок **About box** (Диалоговое окно "О программе"), позволяющее включить в приложение возможность вывода диалогового окна, содержащего основные сведения о разработчике программы, ее версии и правах собственности на данную программу.

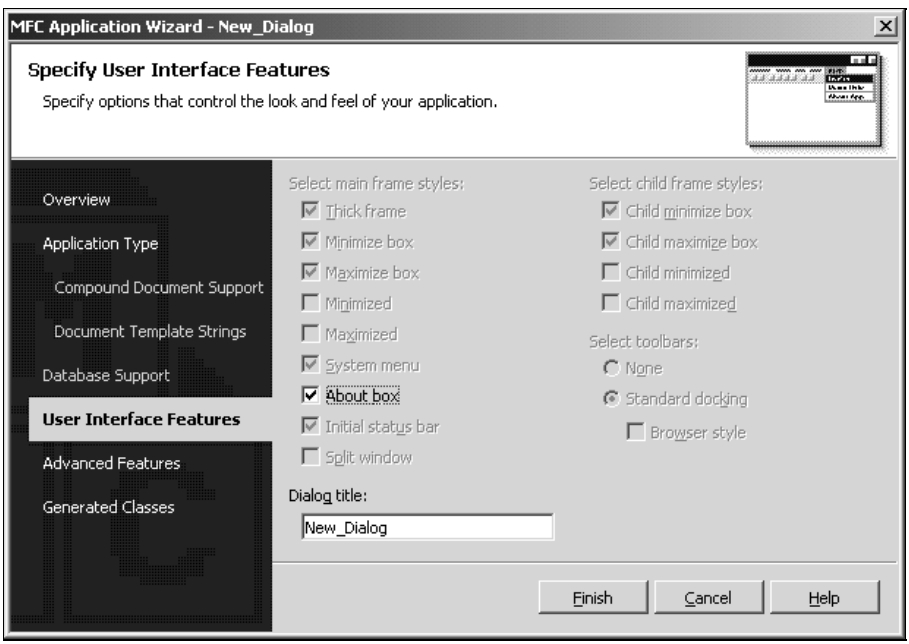


Рис. 13.60. Диалоговое окно **MFC Application Wizard - New_Dialog**

Окно мастера MFC Application Wizard с раскрытой вкладкой **User Interface Features** (Особенности интерфейса пользователя) приведено на рис. 13.60.

При создании многооконного или однооконного приложения вопрос о необходимости включения подобного окна в приложение даже не ставится: оно включается в приложение автоматически (этот флажок недоступен).

По умолчанию диалоговое окно **About MyApp** (О программе) имеет вид, приведенный на рис. 13.61.

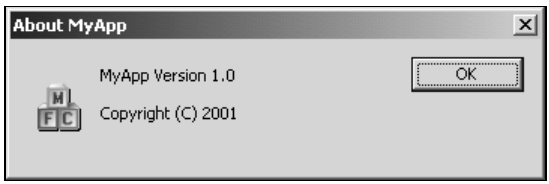


Рис. 13.61. Диалоговое окно **About MyApp**

После русификации файла ресурсов приложения HelpApp его диалоговое окно **О программе** приняло вид, приведенный на рис. 13.62.

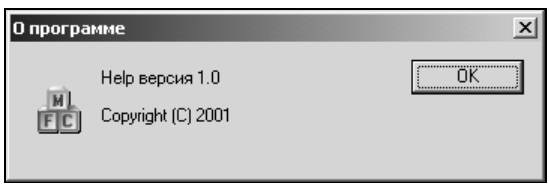
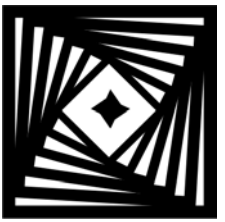


Рис. 13.62. Диалоговое окно **О программе**

Конечно, в его настоящем виде диалоговое окно **О программе** несет мало полезной информации, однако, как и всякое другое диалоговое окно, его можно отредактировать в соответствующем редакторе ресурса и, с использованием статического текста, внести в него всю информацию, которую создатель программы желает о себе сообщить. При этом в него могут быть помещены любые битовые образы вместо используемого по умолчанию значка библиотеки MFC.

Глава 14



Отладка приложения

Отладка приложения является существенной частью процесса программирования. Особенно это относится к программированию в среде Windows, в которой приложения управляются сообщениями и невозможно заранее определить, в какой последовательности будут выполняться его отдельные фрагменты.

Процесс отладки тесно связан с инструментарием, предоставляемым интерфейсом пользователя Visual C++, поэтому все компоненты интерфейса пользователя, связанные с процессом отладки, будут рассмотрены в данной главе. Остальные компоненты интерфейса пользователя рассматриваются в *приложении 2.*

Средства отладки, предоставляемые интерфейсом пользователя

Очевидно, что при отладке программ ключевым понятием является термин *точка останова.* Под точкой останова понимается место в программе, в котором требуется приостановить выполнение программы и дать пользователю возможность воспользоваться средствами отладки, предоставляемыми средой программирования Visual C++. После получения необходимой информации пользователь может:

- возобновить выполнение программы до следующей точки останова или до конца программы, если при ее дальнейшем выполнении не встретятся точки останова. Для этого он может нажать клавишу <F5> или выбрать команду меню **Debug | Continue** (Отладка | Продолжить). (Содержимое меню **Debug** (Отладка) изменяется после запуска приложения на исполнение. Вид этого меню в режиме отладки приведен на рис. 14.1);
- перезапустить исполняемое приложение. Для этого он может нажать комбинацию клавиш <Ctrl>+<Shift>+<F5> или выбрать команду меню **Debug | Restart** (Отладка | Перезапуск);.
- завершить работу с приложением. Для этого он может нажать комбинацию клавиш <Shift>+<F5> или выбрать команду меню **Debug | Stop Debugging** (Отладка | Остановить отладку);
- выполнить один оператор программы и снова остановиться. Для этого он может нажать клавишу <F10> или выбрать команду меню **Debug | Step Over** (Отладка | Обойти вызов функции);
- в том случае, если текущий оператор является функцией, доступной для отладки, пользователь может войти в эту функцию и остановиться на первом ее

операторе (заголовке). Для этого он может нажать клавишу <F11> или выбрать команду меню **Debug | Step Into** (Отладка | Войти в функцию). Если текущий оператор не является функцией, данная команда работает аналогично предыдущей команде. Если текст данной функции не содержится в каталогах, указанных пользователем при настройке системы, она выводит диалоговое окно, предлагая указать путь к данному файлу. Если данная функция содержится в системной библиотеке и недоступна для отладки, система выводит окно сообщения с соответствующим текстом;

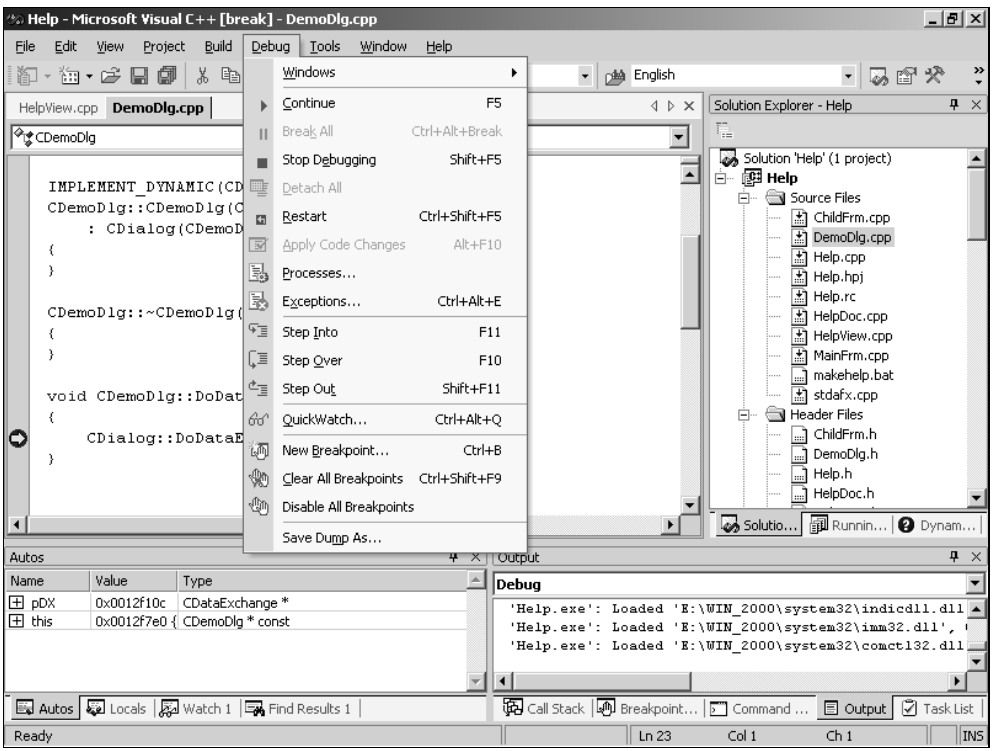


Рис. 14.1. Раскрытое меню **Debug**

- при нажатии комбинации клавиш <Shift>+<F11> или выборе команды меню **Debug | Step Out** (Отладка | Выйти из функции) приложение завершает работу данной функции и останавливается на следующем после вызова данной функции операторе;
- пользователь может продолжить выполнение программы и остановиться на операторе, на котором в данное время расположен текстовый курсор. Для этого он должен нажать комбинацию клавиш <Ctrl>+<F10>.

Точки останова

Как следует из вышесказанного, одним из способов установки точки останова является нажатие комбинации клавиш <Ctrl>+<F10>. Это наиболее простой и наиболее распространенный метод. Однако он имеет свои недостатки:

- таким образом может быть установлена только одна точка останова;
- данным методом невозможно воспользоваться, если приложение находится в состоянии ожидания команды пользователя, а это состояние очень удобно для установки точек останова в процессе выполнения программы;
- если данная команда используется для запуска приложения на исполнение, то возникновение ошибок в процессе компиляции отвлекает пользователя и он может после их устранения автоматически запустить приложение нормальным образом, не установив точку останова, и после этого долго удивляться, почему приложение не останавливается в данной точке.

Более надежным способом установки точек останова является использование клавиши <F9>. Чтобы установить или удалить точку останова, достаточно установить текстовый курсор на соответствующую строку текста программы и нажать клавишу <F9>. Точка останова индицируется красной точкой, расположенной слева от строки текста, перед выполнением которой приложение будет остановлено. Точка останова может быть также установлена или снята щелчком левой кнопки мыши в серой полосе, расположенной слева от текста редактируемого файла.

Для редактирования точек останова выберите команду **Debug | Windows | Breakpoints** (Отладка | Окна | Точки останова). Появится окно **Breakpoints** (Точки останова). Как правило, оно появляется в виде вкладки, но на рис. 14.2 оно показано как отдельное окно.

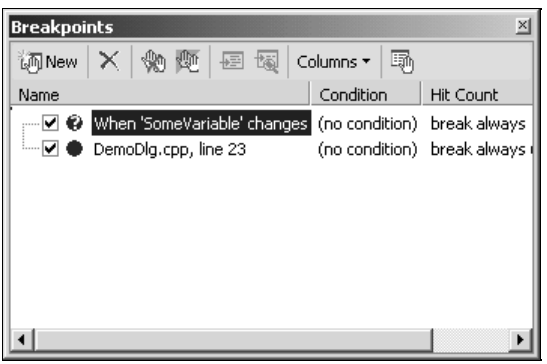


Рис. 14.2. Окно **Breakpoints**

Это окно позволяет создавать новую точку останова, но при этом от программиста требуется вручную ввести координаты этой точки, что в большинстве случа-

ев не совсем удобно. Проще установить точку останова клавишей <F9> или щелчком левой кнопкой мыши на серой полосе, а затем соответствующим образом отредактировать свойства этой точки останова. Чтобы отредактировать свойства точки останова, щелкните на ней правой кнопкой мыши в окне **Breakpoints** (Точки останова) и выберите в появившемся контекстном меню команду **Properties** (Свойства). Появится диалоговое окно **Breakpoint Properties** (Свойства точек останова), изображенное на рис. 14.3.

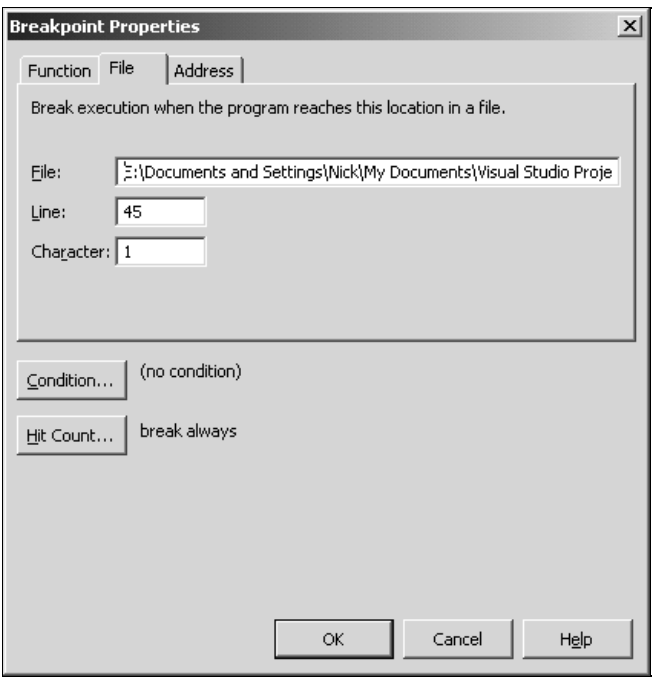


Рис. 14.3. Диалоговое окно **Breakpoint Properties**

При отладке программы часто приходится просматривать длинные циклы в ожидании возникновения события, вызвавшего отказ программы. Для упрощения этого процесса можно использовать кнопки **Condition** (Условие) и **Hit Count** (Счетчик повторов).

При нажатии кнопки **Condition** (Условие) появляется диалоговое окно **Breakpoint Condition** (Условия останова), изображенное на рис. 14.4. Это диалоговое окно содержит текстовое поле, в которое вводится условие, и переключатель. При установке переключателя в положение **is true** (Истина) значение выражения в текстовом поле преобразуется к логическому типу и программа будет останавливаться в этой точке только в том случае, если это значение будет истинно. При установке переключателя в положение **has changed** (Изменена) программа

будет останавливаться в данной точке только в том случае, если значение выражения в текстовом поле изменилось с момента предыдущего прохода программы через нее. Сброс флажка **Condition** (Условие) позволяет превратить эту точку останова в безусловную, сохранив при этом возможность быстро вернуться к установленным в ней условиям.

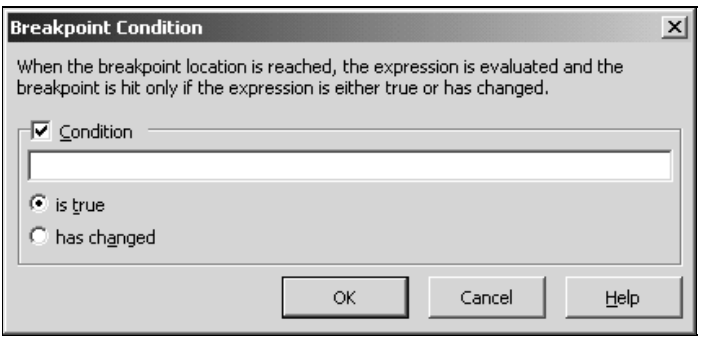


Рис. 14.4. Диалоговое окно **Breakpoint Condition**

При просмотре больших циклов бывает полезно увеличить шаг просмотра и производить остановку не на каждом шаге, а через определенное число шагов. Это позволит достаточно быстро локализовать значения переменной цикла, при которых происходит сбой, а затем вернуться к этим значениям по условию. Для этого используется кнопка **Hit Count** (Счетчик повторов), при нажатии которой на экран выводится диалоговое окно **Breakpoint Hit Count** (Счетчик повторов точки останова), изображенное на рис. 14.5.

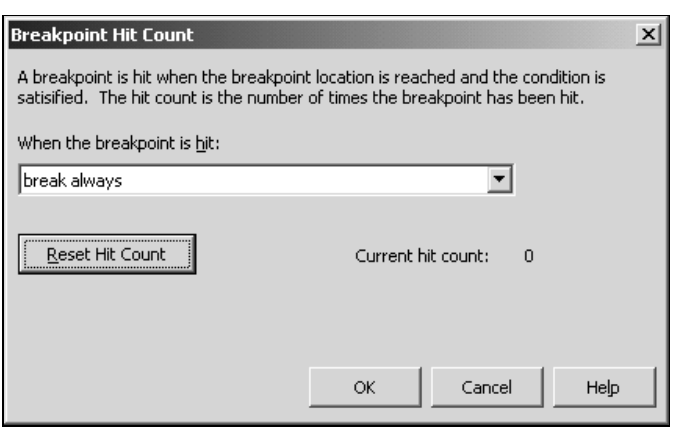


Рис. 14.5. Диалоговое окно **Breakpoint Hit Count**

В текстовое поле раскрывающегося списка **When the breakpoint is hit** (Условие остановки) выводится условие остановки в данной точке. Из списка могут быть выбраны следующие условия:

- **break always** — безусловная остановка (устанавливается по умолчанию);
- **break when the hit count is equal to** — остановка по равенству;
- **break when the hit count is multiple of** — остановка, если число проходов кратно указанному значению;
- **break when the hit count is greater then or equal to** — остановка по равенству или превышению.

В этом окне выводится текущее значение счетчика и имеется кнопка **Reset hit count** (Сброс счетчика повторов) для его сброса в ноль.

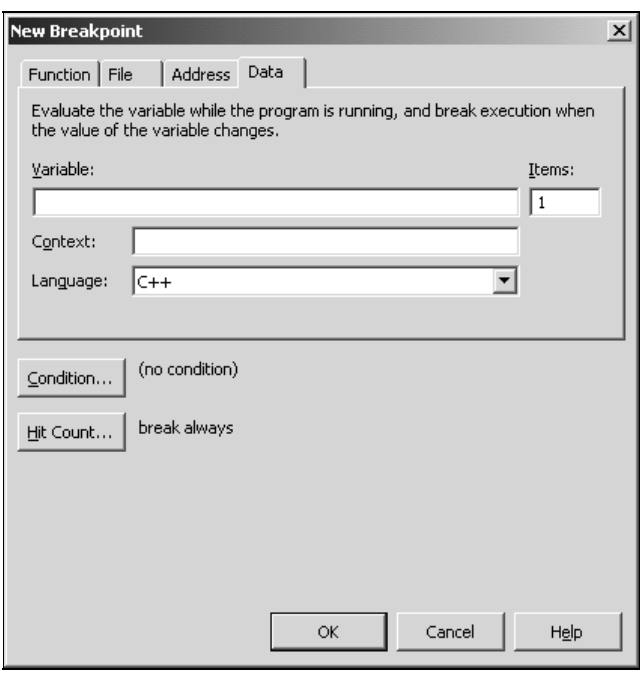


Рис. 14.6. Диалоговое окно **New Breakpoint**

Практически единственным случаем, когда целесообразно создавать новую точку останова из окна **Breakpoints** (Точки останова), является установка точки останова по изменению значения. Необходимость в установке подобной точки останова возникает в том случае, когда надо отследить момент изменения значения некоторой переменной, используемой во многих функциях. Чтобы установить такую точку останова, нажмите кнопку **New** (Создать) окна **Breakpoints** (Точки останова). Появится диалоговое окно **New Breakpoint** (Новая точка оста-

нова). Это окно практически неотличимо от окна **Breakpoint Properties** (Свойства точек останова) за исключением того, что в последнем заполнены все текстовые поля, определяющие положение данной точки останова, и в нем отсутствуют вкладки, неприменимые к данной точке останова. Раскройте в этом диалоговом окне вкладку **Data** (Данные). Диалоговое окно **New Breakpoint** (Новая точка останова) примет вид, изображенный на рис. 14.6.

Идентификатор глобальной переменной, при изменении значения которой следует прекратить выполнение программы, помещается в текстовое поле **Variable** (Переменная), а значение в текстовом поле **Items** (Элементы) определяет, за значениями скольких переменных данного типа следует вести наблюдение. Содержимое текстового поля **Items** (Элементы) используется в том случае, если в текстовом поле **Variable** (Переменная) указан идентификатор массива или указатель.

Если в процессе выполнения программы значение указанной переменной изменится, ее исполнение будет остановлено напротив строки, следующей за строкой, в которой произошло изменение, появится желтая стрелка и на экран будет выведено окно сообщения, содержащее сведения о причине остановки. Такое окно показано на рис. 14.7.

Рис. 14.7. Диалоговое окно **Microsoft Development Environment**

Использование окна списка **Breakpoints** (Точки останова) позволяет быстро перейти к любой точке останова в любом файле проекта. Для этого достаточно дважды щелкнуть левой кнопкой мыши на строке, соответствующей данной точке останова в окне списка.

Анализ исполнения программы

Обычно точки останова ставятся для обеспечения программисту возможности анализа процесса исполнения программы и, прежде всего, для анализа значений переменных в данной точке и последовательности функций, вызов которых привел программу в данную точку. Этот анализ позволяет оценить правильность выполнения программы и обнаружить источники ошибок, возникающих в процессе ее выполнения.

Среда программирования Visual C++ в процессе отладки приложения показана на рис. 14.8. Активная точка останова (на которой прервалось выполнение приложения) отмечается желтой стрелкой, расположенной поверх красной точки,

соответствующей точке останова. В нижней части экрана располагаются два окна. Левое окно обычно содержит вкладки **Autos** (Авто), **Locals** (Локальные) и **Watch** (Просмотр). Они имеют практически одинаковую структуру и различаются тем, что окно **Autos** (Авто) заполняет Visual C++ исходя из одному ему ведомого критерия актуальности значений отдельных переменных в данной точке программы, в окне **Locals** (Локальные) выводятся локальные переменные, а окно **Watch** (Просмотр) заполняет пользователь.

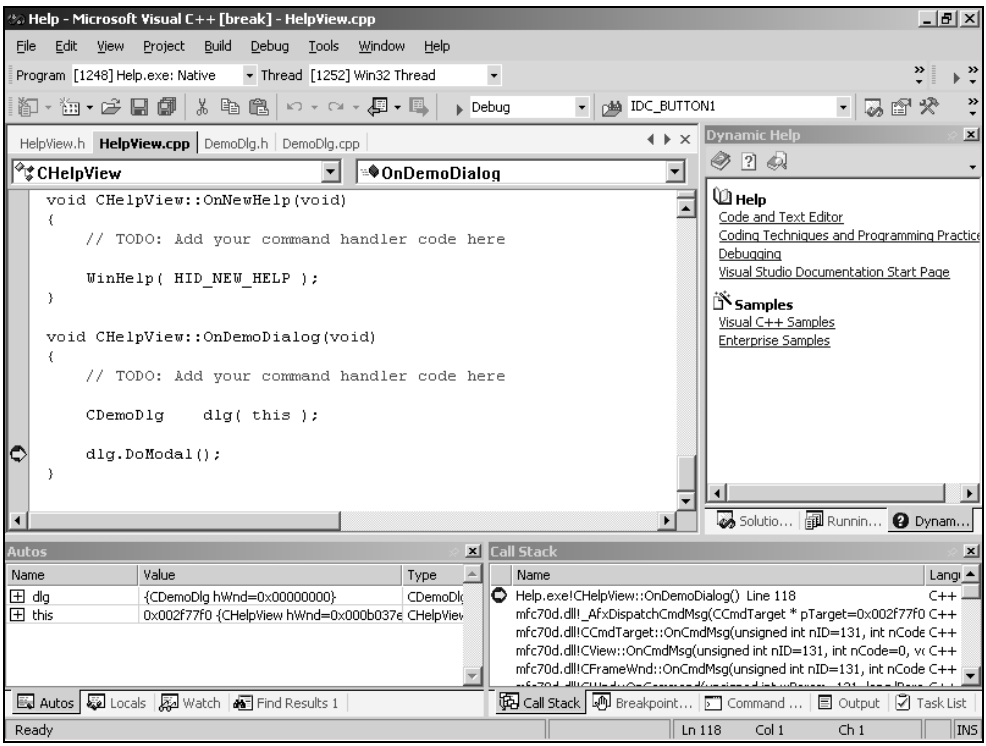


Рис. 14.8. Среда Visual C++ в процессе отладки приложения

Как правило, в окне **Autos** (Авто) отображаются значения переменных, используемых в текущем и предыдущем операторах программы. Если значение переменной было изменено с момента прошлого его отображения в данном окне, оно отображается красным цветом. Если около имени переменной стоит знак плюс, то эта переменная представляет собой объект структуры или класса. Чтобы просмотреть значения его переменных, достаточно щелкнуть левой кнопкой мыши по этому крестику.

Содержимое окна **Watch** (Просмотр) определяется пользователем и сохраняется даже после закрытия приложения. Чтобы поместить идентификатор переменной в поле **Name** (Имя) окна **Watch** (Просмотр), достаточно выделить его блоком,

поместить на этот блок указатель мыши, нажать левую кнопку мыши и, не отпуская ее, переместить указатель мыши в окно **Watch** (Просмотр). При этом в окне выделяется последняя занятая строка, после которой и вставляется имя выделенной переменной после отпускания левой кнопки мыши.

В окне **Watch** (Просмотр) могут просматриваться значения всех переменных и выражений, находящихся в текущей области видимости. Просматриваемое значение может быть соответствующим образом отформатировано. Синтаксис форматирования переменной предусматривает установку запятой после идентификатора переменной, а затем символа форматирования. Так, следующее выражение форматирует переменную с именем `result` как `HRESULT`:

```
result, hr
```

В этом случае значение `0x00000000L` выводится как `S_OK`, при указании формата флага сообщения Windows значение `0x000F` выводится как сообщение `WM_PAINT`.

Ниже приведен список наиболее распространенных кодов форматирования:

- `d`, `i` — десятичное целое число со знаком;
- `u` — десятичное целое число без знака;
- `o` — восьмеричное целое число без знака;
- `x`, `X` — шестнадцатеричное число;
- `l`, `h` — префикс `long` или `short` для типов `d`, `i`, `u`, `o`, `x`, `X`;
- `f` — число с плавающей точкой со знаком;
- `e` — число в научном написании со знаком;
- `g` — число с плавающей точкой или число в научном написании со знаком в зависимости от того, какое из них короче;
- `c` — отдельный символ;
- `s` — строка;
- `su` — строка Unicode;
- `st` — строка ANSI или строка Unicode в зависимости от значения `Unicode Strings` в файле autoexp.dat;
- `hr` — `HRESULT` или код ошибки Win32;
- `we` — флаг класса Windows;
- `wm` — номер сообщения Windows.

При необходимости внести изменения в идентификатор наблюдаемой переменной достаточно щелкнуть на нем левой кнопкой мыши. При этом соответствующая строка выделится. При вводе первого символа выделение убирается, но возможность редактирования при этом сохраняется. Щелчок по пустой строке позволяет ввести в нее идентификатор новой наблюдаемой переменной.

Другим способом помещения идентификатора переменной в окно **Watch** (Просмотр) является использование команды меню **Debug | QquickWatch** (Отладка | Быстрый просмотр), комбинации клавиш <Shift>+<Alt>+<Q> или нажатие

кнопки **QuickWatch** (Быстрый просмотр), расположенной на панели инструментов **Debug** (Отладка). Все эти действия приведут к появлению диалогового окна **QuickWatch** (Быстрый просмотр), изображенного на рис. 14.9. Нажатие кнопки **Add Watch** (Добавить просмотр) в данном окне добавит соответствующую переменную в окно **Watch** (Просмотр).

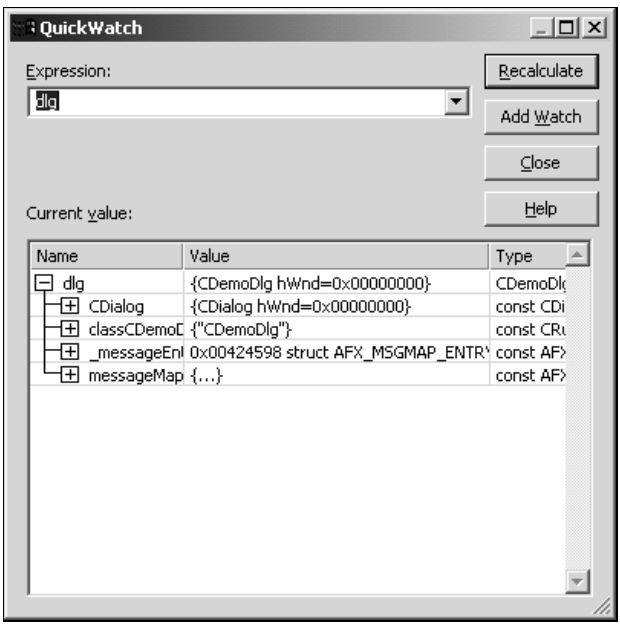


Рис. 14.9. Диалоговое окно **QuickWatch**

В окне, расположенном справа на рис. 14.8, раскрыта вкладка **Call Stack** (Вызвать стек), содержащая полный список функций, вызов которых привел в данную точку останова. В корне этого списка обычно находятся системные функции, не допускающие отладки. Функция, в которой находится точка останова, располагается в голове списка и маркируется так же, как и соответствующая строка программы. При необходимости просмотреть значения переменных в функциях данного списка достаточно дважды щелкнуть на них левой кнопкой мыши. Соответствующая функция выделяется стрелкой зеленого цвета и в окне редактирования файлов появляется ее текст.

Если в вашем приложении используется пользовательская функция для вывода информации обо всех ошибках приложения, то имеет смысл поместить в ней точку останова. В этом случае, при возникновении любой ошибки приложение будет останавливаться, оставаясь в рабочем состоянии. Последовательный просмотр функций в списке окна **Call Stack** (Вызвать стек), как правило, позволяет достаточно быстро установить конкретную причину возникновения ошибки.

Анализ конкретной причины возникновения ошибки позволит найти и ошибку в программе, ее вызвавшую.

Как уже говорилось, в окне **Autos** (Авто), как правило, отображаются только переменные, используемые в данном и в предыдущем операторах программы. Кроме того, это окно может иметь полосу прокрутки, которой бывает необходимо воспользоваться для просмотра значения интересующей вас переменной. Следует учесть, что в большинстве случаев после выполнения очередного оператора программы список прокручивается в начало и для просмотра значения переменной его нужно снова прокручивать. Поэтому во многих случаях для оценки значения переменной достаточно поместить на нее указатель мыши. При этом появляется окно **Data Tip** (Значение), в котором содержится ее значение. Если оцениваемая переменная является членом структуры или класса, то можно выделить эту переменную вместе с именем объекта ее структуры или класса и воспользоваться окном **Data Tip** (Значение). Однако этот метод срабатывает не во всех случаях.

Кратко рассмотрим другие окна, используемые при отладке.

- В окне **Registers** (Регистры) выводится содержимое основных регистров процессора. По умолчанию это окно не выводится при первом запуске приложения на отладку. Однако после своего вывода оно включается в установки проекта для сеанса отладки и будет выводиться при последующих запусках приложения на отладку. Пользователь может непосредственно изменять значения, содержащиеся в регистрах, изменяя их в окне. Кроме того, содержимое регистров можно перетаскивать в окна **Watch** (Наблюдение) и **Memory** (Память) для просмотра контекста.

  Для обхода некоторого фрагмента программы пользователь в процессе отладки может изменять содержимое регистра `EIP` (указатель команд).

  Пользователь также может заменить значение, возвращаемое функцией, если она помещает его в регистр `EAX` (так поступают многие функции).

- Окно **Memory** (Память) позволяет пользователю просмотреть содержимое большого связного фрагмента памяти. Это может понадобиться при просмотре содержимого большого текстового буфера, строки или массива. Это окно позволяет не только просматривать содержимое области памяти, но и редактировать его.

  Панель инструментов данного окна позволяет выбрать начальный адрес отображаемого фрагмента и число столбцов в окне, а его контекстное меню позволяет установить режим отображения информации как целого числа или числа с плавающей запятой, а также текстовый формат, в который параллельно преобразуется данная информация. Целые числа могут выводиться в шестнадцатеричном формате, а также в формате целого числа со знаком или без знака, а числа с плавающей запятой — в формате 32-разрядного или 64-разрядного числа. Содержимое буфера может вообще не преобразовываться в текстовый формат или выводиться в формате ASCII или Unicode.

- В окне **Disassembly** (Дизассемблер) выводится ассемблерный код исполняемого приложения. Содержимое этого окна может быть использовано для контроля результатов оптимизации кода программы, которая может в распространяемой версии привести к проблемам, отсутствующим в неоптимизированных отладочных проектах. Кроме того, это окно позволяет перемещаться между инструкциями, расположенными в одной строке исходного кода C/C++.
- В окне **Call Stack** (Стек вызова) выводятся функции, которые программа вызвала, но еще не завершила. Это окно позволяет определить путь, который привел программу в ее текущее положение. Если точка останова установлена в функции обработки ошибок приложения, окно **Call Stack** (Стек вызова) позволит перейти к функции, в которой возникла данная ошибка и разобраться с причиной ее возникновения.

Настройка уровня предупреждений транслятора

При отладке приложения часто приходится рассматривать возможность влияния на программу таких факторов, которые обычно не рассматриваются, и вывод которых в процессе трансляции считается излишним.

Чтобы повысить уровень выводимых предупреждений транслятора:

1. В окне **Solution Explorer** (Проводник решения) или в окне **Class View** (Просмотр классов) щелкните правой кнопкой мыши на папке с именем приложения и в появившемся контекстном меню выберите команду **Properties** (Свойства) (или же выделите эту папку и выберите команду меню **Vview | Property Pages** (Вид | Вкладки свойств) или нажмите комбинацию клавиш <Shift>+<F4>). Появится диалоговое окно **Help Property Pages** (Вкладки свойств), изображенное на рис. 14.10.
2. В раскрывающемся списке **Configuration** (Конфигурация) выделите строку **Debug** (Отладка) (если у вас установлена другая конфигурация).
3. Раскройте папку **C/C++** и выделите в ней строку **General** (Файлы общих типов) (если она еще не выделена).
4. В окне списка выделите строку **Warning level** (Уровень предупреждений транслятора). В соответствующем текстовом поле появится значок раскрывающегося списка.
5. Раскройте этот список и выделите в нем строку **Level 4**. Диалоговое окно **Property Pages** (Вкладки свойств) примет вид, изображенный на рис. 14.11.
6. Нажмите кнопку **OK**. Диалоговое окно **Help Property Pages** (Вкладки свойств) закроется.

При создании окончательной версии приложения будет использоваться установленный по умолчанию третий уровень предупреждений транслятора.

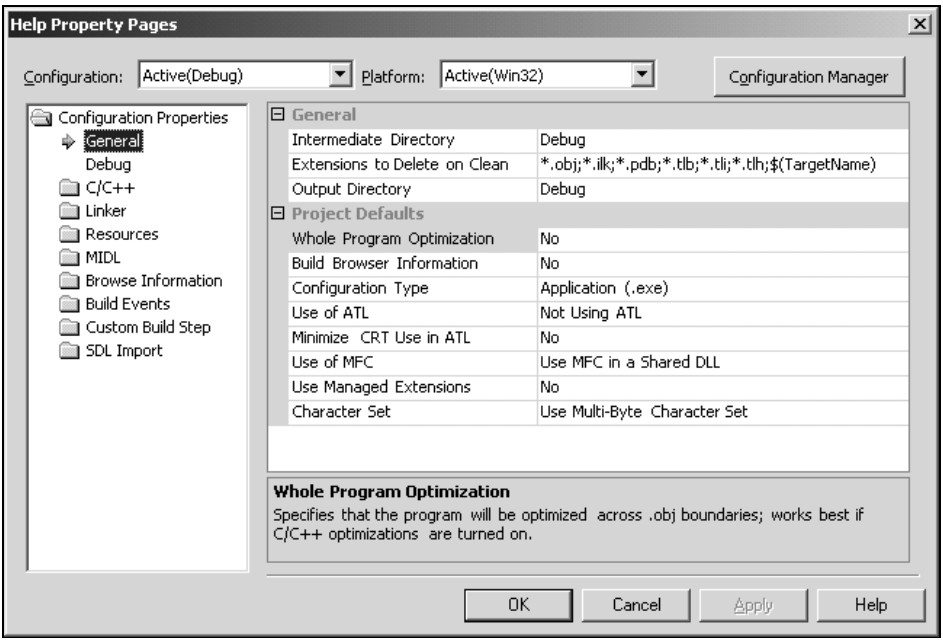


Рис. 14.10. Диалоговое окно **Help Property Pages**

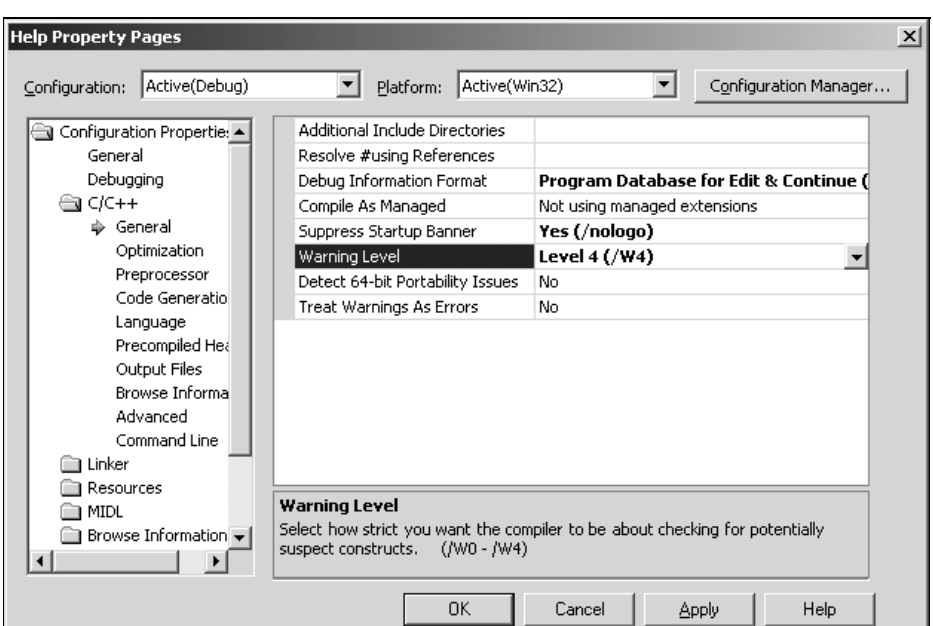


Рис. 14.11. Диалоговое окно **Help Property Pages**, папка **C/C++**

Программные средства отладки

При программировании приложения в него необходимо включить определенные операторы, используемые только в процессе его отладки. Включение данных операторов замедляет выполнение программы и чревато выдачей пользователю сообщений, понятных только программисту. Поэтому, при создании окончательной версии приложения эти операторы следует удалить. Однако для качественной отладки приложения количество отладочных операторов должно быть велико и они должны находиться в большинстве функций приложения.

Удаление этих операторов вручную представляет собой достаточно сложную задачу и не дает гарантию того, что все отладочные операторы будут удалены, и что вместе с ними не будут удалены и рабочие операторы. Кроме того, никто не может поручиться за то, что окончательную версию не придется отлаживать заново. Например, при внесении в нее изменений "по просьбам трудящихся". Поэтому в Visual C++ предусмотрено два режима трансляции приложения: **Debug** (Отладочный) и **Release** (Окончательный) и созданы специальные функции и макросы, работающие только в отладочном режиме и игнорируемые в окончательном приложении.

Ниже приведены отличительные особенности большинства отладочных версий программ:

- запрещена оптимизация при компиляции;
- с помощью директивы `#define` определяется символ `_DEBUG`;
- к программе подключаются отладочные версии библиотек поддержки и/или библиотек MFC;
- в проект включается символьная отладочная информация.

Определенный в отладочной версии приложения Visual C++ символ `_DEBUG` используется в директивах условной трансляции и в макросах отладки для исключения отладочной информации в окончательной версии программы.

Макросы *ASSERT* и *TRACE*

Макрос `ASSERT` (подтверждение) используется для проверки некоторых логических условий, которые должны выполняться в данной точке программы. Предположим, что в некоторую функцию приложения передается указатель на некоторый объект `lpObject`. Предполагается, что этот указатель имеет ненулевое значение, однако для предотвращения возможных ошибок это следует проверить. Для этого перед использованием данного указателя следует вызвать макрос `ASSERT`, как показано ниже:

```
ASSERT(lpObject)
```

Всякий раз, когда логическое выражение, переданное в качестве аргумента макросу `ASSERT`, будет принимать значение `FALSE`, выполнение приложения будет останавливаться и на экран будет выводиться окно сообщения, аналогичное

приведенному на рис. 14.12. В данном окне указывается имя исполняемого файла, имя исходного файла и номер строки, в которой произошла ошибка. Нажатие кнопки **Retry** (Повтор) позволяет перейти в текст программы для ее отладки. Причем текущая точка останова устанавливается на строку соответствующего макроса `ASSERT`.

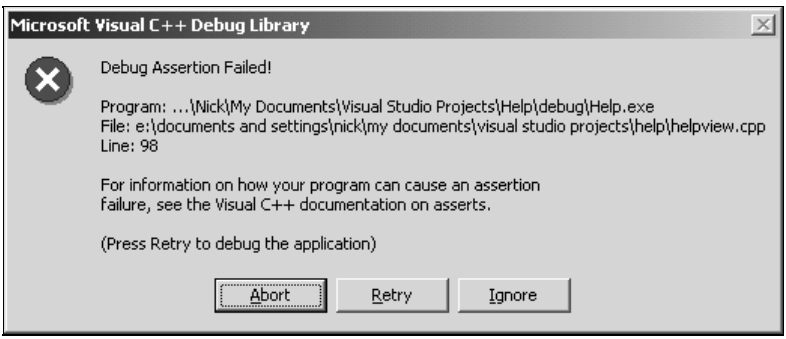


Рис. 14.12. Окно сообщения об ошибке

Макрос `TRACE` служит для вывода диагностических сообщений. При создании приложения программист обычно хорошо знает те места, в которых может возникнуть ошибка. Например, многие функции возвращают значение, говорящее о том, успешно ли завершилась их работа и, возможно, являющееся кодом возникшей ошибки. Это значение можно игнорировать, считая, что ничего плохого с этим приложением случиться не может, можно прекратить выполнение функции, в которой возникла ошибка, а можно и послать при этом сообщение пользователю.

Для посылки сообщения могут быть использованы и обычные функции вывода, однако они будут работать и в окончательной версии, что нежелательно в случае возникновения ошибок, позволяющих сравнительно безболезненно продолжить выполнение приложения, или простых информационных сообщений, используемых программистом в процессе отладки.

Для подобного вывода диагностической информации следует использовать макрос `TRACE`. Вообще-то под макросом `TRACE` понимается целое семейство макросов, включающее в себя макросы `TRACE`, `TRACE0`, `TRACE1`, `TRACE2` и `TRACE3`. Число в имени макроса указывает на количество параметрических аргументов в данном макросе. Под параметрическим аргументом понимается идентификатор переменной, значение которой должно быть преобразовано в текстовую строку в соответствии с указанным форматом. Синтаксис макроса `TRACE` аналогичен синтаксису функции `printf`. Все макросы `TRACE` посылают свои сообщения в поток `afxDump`.

Макрос `TRACE` позволяет выводить сообщения с любым числом параметрических аргументов. Макросы `TRACE0`, `TRACE1`, `TRACE2` и `TRACE3` созданы исключительно

с целью экономии места в сегменте данных. Ниже приведен пример использования макроса TRACE0 в функции CMainFrame::OnCreate.

```
if (!m_wndStatusBar.Create(this)  |  |
  !m_wndStatusBar.SetIndicators(indicators,
        sizeof(indicators)/sizeof(UINT)))
{
  TRACE0("Failed to create status bar\n");
  return -1;  // Ошибка создания строки состояния
}
```

Следующий пример демонстрирует вывод сообщения о возникновении ошибки в файле:

```
TRACE2("Ошибка номер %d в файле %s\n", nError, szFileName);
```

В окончательной версии приложения, в которой константа _DEBUG не определена, макросы ASSERT и TRACE не выполняют никаких действий. Это позволяет оставлять их в тексте программы не опасаясь, что они замедлят выполнение окончательной версии приложения или будут выдавать конечному пользователю непонятные сообщения.

Отладочные функции

Если для целей отладки недостаточно использования макросов, пользователь может создавать собственные отладочные функции, которые будут исключаться из окончательной версии приложения. Примером такой функции являются функции CObject::AssertValid и CObject::Dump, наследуемые практически всеми классами библиотеки MFC от класса CObject.

Функция AssertValid служит для проверки допустимости значений данного объекта. В случае возникновения ошибки она вызывает исключение. Функция Dump вызывается приложением при возникновении ошибки для передачи содержимого пользовательского объекта в объект класса CDumpContext. Данные функции имеют следующий формат:

```
virtual void AssertValid() const;
virtual void Dump(CDumpContext& dc) const;
```

Ключевое слово virtual говорит о том, что данный метод должен быть перегружен в производных классах, а ключевое слово const указывает на то, что данная функция не будет модифицировать состояние объекта.

Подобно отладочным макросам, функции AssertValid и Dump будут отсутствовать в окончательной версии приложения. Для этого объявление данных функций в файле заголовка заключается в операторы условной трансляции, как показано ниже.

```
#ifdef _DEBUG
  virtual void AssertValid() const;
```

```
  virtual void Dump(CDumpContext& dc) const;
#endif
```

По умолчанию данные функции просто вызывают методы базового класса. Однако в файл реализации каждого класса, производного от класса `CObject`, должна быть включена реализация этих функций, заключенная в операторы условной трансляции. Эту реализацию, как правило, включает в данный файл мастер MFC Application Wizard. Ниже приведена реализация этих функций, включенная мастером в файл реализации класса `CMainFrame`.

```
/////////////////////////////////////////////////////////////////////////////
// CMainFrame diagnostics

#ifdef _DEBUG
void CMainFrame::AssertValid() const
{
  CMDIFrameWnd::AssertValid();
}

void CMainFrame::Dump(CDumpContext& dc) const
{
  CMDIFrameWnd::Dump(dc);
}

#endif //_DEBUG
```

Примеры перегрузки этих функций приведены в их описаниях, содержащихся на дискете.

Глобальные диагностические функции

Помимо описанных выше отладочных функций и макросов, разработчик программного обеспечения может использовать глобальные диагностические функции. Ниже будут кратко рассмотрены некоторые из них.

- `AfxCheckError`. Эта функция имеет единственный аргумент, имеющий тип `SCODE`, и используется для проверки возвращаемых значений функций OLE и корректного вызова исключений MFC в случае необходимости.
- `AfxCheckMemory`. Эта функция проверяет существующую кучу, выводя сообщения о найденных несоответствиях в окно **Output** (Вывод). Эта функция не имеет аргументов и возвращает логическое значение статуса. При вызове данной функции задействовано много ресурсов, поэтому ее вызов существенно замедляет исполнение приложения.
- `AfxEnableMemoryTracking`. Данная функция используется для отмены проверки памяти в отладочной версии библиотеки MFC. Имеет единственный логический аргумент, определяющий необходимость проверки памяти.

- `AfxIsMemoryBlock`. Эта диагностическая функция проверяет соответствие предоставленных ей в качестве аргументов указателя и размера блока с теми, которые получаются при использовании отладочной версии оператора `new`.
- `AfxIsValidAddress`. Данная функция проверяет корректность указателя и размера блока в адресном пространстве вызывающей программы. Имеет необязательный третий аргумент, определяющий возможность записи информации в эту область памяти.
- `AfxIsValidString`. Эта функция проверяет, что переданный ей аргумент указывает на корректную строку. Второй ее аргумент, в котором передается длина строки, по умолчанию имеет значение −1, указывающее на то, что строка завершается нулем.
- `AfxSetAllocHook`. Используется для определения функции обратного вызова, вызываемой перед запуском процедуры выделения памяти. Она имеет единственный аргумент типа `AFX_ALLOC_HOOK`.

Отладка распространяемой версии приложения

Как уже говорилось выше, оптимизация кода, производимая при создании окончательной версии приложения, может привести к возникновению ошибок, отсутствующих в его отладочной версии. В этом случае разработчик оказывается в затруднительном положении: при переходе к отладочной версии ошибки исчезают, а в распространяемой версии приложения отсутствуют средства отладки.

Поэтому в Visual C++ 6.0 появилась функция `AfxDumpStack`, позволяющая пользователю без помощи отладчика извлекать из стека информацию обо всех функциях, выполнение которых не завершено на момент ее вызова. Другими словами, эта функция позволяет проследить путь приложения к данной точке программы. Ее прототип выглядит следующим образом:

```
void AFXAPI AfxDumpStack(DWORD dwTarget = AFX_STACK_DUMP_TARGET_DEFAULT);
```

В аргументе `dwTarget` может передаваться комбинация следующих значений, объединенная оператором побитового ИЛИ.

- `AFX_STACK_DUMP_TARGET_DEFAULT`. Это значение используется по умолчанию. Оно посылает отладочную информацию в контекст вывода, определенный по умолчанию. При работе с отладочной версией в качестве контекста вывода используется макрос `TRACE`, а в распространяемой версии — буфер обмена.
- `AFX_STACK_DUMP_TARGET_TRACE`. При использовании этого значения вся выходная информация как отладочной, так и распространяемой версии передается макросу `TRACE`. Таким образом, в распространяемой версии информация не выводится, поскольку макрос `TRACE` в ней игнорируется.
- `AFX_STACK_DUMP_TARGET_CLIPBOARD`. При использовании этого значения вся выходная информация как отладочной, так и распространяемой версии передается в буфер обмена. При записи информации используется формат `CF_TEXT`.

- `AFX_STACK_DUMP_TARGET_BOTH`. При использовании этого значения вся отладочная информация помещается в буфер обмена и одновременно передается макросу `TRACE`.
- `AFX_STACK_DUMP_TARGET_ODS`. Аббревиатура `ODS` указывает на использование функции Win32 `OutputDebugString`. Таким образом, это значение выводит отладочную информацию как для отладочной, так и для распространяемой версии приложения, каждая из которых может быть "подключена" к отладчику.

Для каждой записи стека, создаваемой данным методом, сохраняется следующая информация:

- адрес возврата последней вызванной функции;
- модуль и полный путь к файлу, содержащему данную функцию;
- прототип функции;
- смещение в байтах от прототипа функции до адреса возврата.

Формат отладочной информации, выводимой данной функцией, существенно отличается от формата окна **Call Stack** (Вызов стека), причем не в лучшую сторону, однако на безрыбье и рак рыба. Кроме того, следует помнить, что для своей корректной работы эта функция требует установки библиотеки imagehlp.dll. В противном случае при вызове функции появится сообщение об ошибке.

Устранение утечки памяти

Утечку памяти можно считать одной из наиболее распространенных и трудно локализуемых ошибок. Небольшие утечки памяти могут не проявлять себя до тех пор, пока приложение не будет запущено на машине с малым объемом оперативной памяти или параллельно с большим количеством других задач.

Поскольку программисты обычно работают на машинах с большим объемом оперативной памяти и при проверке функционирования своего приложения стараются не запускать других приложений (кроме среды программирования), то эти ошибки могут остаться незамеченными и перейти в окончательную версию. Последствия утечек памяти могут быть достаточно серьезными вплоть до краха операционной системы с необходимостью ее переинсталляции.

Основные причины возникновения утечек памяти

В принципе, все утечки памяти имеют одну причину: где-то в приложении под некоторый объект выделяется область оперативной памяти, которая не освобождается после использования данного объекта. Другой вопрос, что эта единая причина имеет несколько конкретных проявлений, на которые следует обратить внимание программисту.

Ниже приведены основные способы добиться утечки памяти в приложении, которыми часто пользуются программисты:

- создать указатель на объект с использованием оператора `new` и не освободить его с использованием оператора `delete`;
- создать указатель на объект с использованием оператора `new[]` и освободить его с использованием оператора `delete`. (Необходимо использовать оператор `delete[]`);
- присвоить переменной, в которой хранится указатель на некоторый объект, другое значение, не освободив этот объект;
- при вызове исключения не освобождать область памяти, выделенную из кучи переменным данной функции.

Эти способы имеют свои разновидности, связанные с работой с классами. Так, например, разновидностью второго метода является использование в классе переменной, являющейся указателем на объект при отсутствии в классе конструктора копирования, переопределенного оператора присваивания или деструктора. Ниже приведен листинг класса, в котором содержатся эти ошибки, и программа, вызывающая на их основе утечки памяти.

```
class SomeClass
{
  public:
    int*  p;
    SomeClass()  { p = new int[10];};
};
SomeClass SomeFunction()
{
  SomeClass  A;
  SomeClass  B;
  B = A;
  return  B;
}
void OtherFunction()
{
  SomeClass  A = SomeFunction();
}
```

При вызове функции `OtherFunction` вызывается функция `SomeFunction`. В этой функции создаются два объекта класса `SomeClass`, в каждом из них при создании переменной `p` ей присваивается значение указателя на некоторую область памяти для хранения десяти целых чисел.

Поскольку в данном классе отсутствует перегруженный оператор присваивания, то операция присваивания одного объекта данного класса другому трактуется как побитовое присваивание значений соответствующих переменных одного

объекта переменным другого объекта. При этом переменные `p` обоих классов будут указывать на одну и ту же область памяти, зарезервированную для объекта `A`. Указатель на другую область памяти будет потерян.

Оператор `return` использует конструктор копирования класса, который по умолчанию использует операцию побитового копирования переменных исходного объекта класса в создаваемый объект. Этот же конструктор используется и при создании объекта `A` в функции `OtherFunction`.

Включение в данный класс деструктора, уничтожающего указатель на целочисленный массив, только осложнит ситуацию и приведет к появлению явных ошибок, поскольку при закрытии функции `SomeFunction` два деструктора попытаются освободить одну и ту же область памяти, а если эта ошибка не приведет к остановке программы, при инициализации объекта `A` в функции `OtherFunction` будет использован некорректный указатель на область памяти.

Для устранения этих ошибок класс `SomeClass` должен быть переписан следующим образом:

```
class SomeClass
{
  public:
    int*  p;
    SomeClass()       {p = new int[10];};
    SomeClass(SomeClass&);
    ~SomeClass()      {if(p) delete p;};

    void  operator =(SomeClass&);
};

SomeClass:: SomeClass(SomeClass& Input)
{
  p = new int[10];

  for(int i=0; i < 10; i++)
    p[i] = Input.p[i];
}

void SomeClass:: operator =(SomeClass& Input)
{
  if(this == &Input) return;

  delete p;
  p = Input.p;
  Input.p = NULL;
}
```

В деструкторе данного класса использован оператор `delete`, хотя по теории необходимо использовать оператор `delete[]`. Различия в этих операторах при уничтожении массивов трудноуловимы и в некоторых случаях, действительно, необходимо использовать оператор `delete[]`, но в других случаях его использование приводит к возникновению ошибок. Правильный выбор оператора определяется "методом научного тыка".

Проверка значения указателя перед его уничтожением является простой формальностью, поскольку освобождение нулевого указателя не приводит ни к каким действиям, однако использование данного оператора поднимает программиста в собственных глазах.

Проверка значения указателя перед его уничтожением может потребоваться в том случае, если этот указатель содержит адрес объекта класса, имеющего непустой деструктор, который может быть создан, а может и не быть создан в процессе работы с исходным классом. Вызов деструктора для объекта класса, который не был создан, может привести к непредсказуемым последствиям. Поэтому в конструкторе пользовательского класса необходимо присвоить нулевое значение всем указателям на содержащиеся в нем объекты классов, а в деструкторе проверять значение указателя на объект перед его уничтожением. Если в процессе работы с пользовательским классом уничтожается указатель на один из содержащихся в нем объектов, то данному указателю следует присвоить нулевое значение.

В копирующем конструкторе данного класса создается указатель на новую область памяти, в которую копируется содержимое массива исходного объекта класса. Это позволяет впоследствии использовать оба объекта класса. Для разнообразия оператор присваивания выполнен по другому принципу. После него нельзя использовать исходный объект класса. Данный метод обычно используется совместно с методом, примененным в копирующем конструкторе, если в классе содержится индикатор временного объекта. Использование этого индикатора позволяет существенно сократить время выполнения промежуточных операций копирования в том случае, если в данном классе имеются перегруженные операторы арифметических действий, а сами объекты имеют большой размер.

Условный оператор, расположенный в начале функции оператора присваивания, имеет очень большое значение, особенно при используемом методе реализации данного оператора. Он обеспечивает возможность корректной обработки операции присваивания объекта самому себе.

Для предотвращения возможности возникновения утечек памяти:

- по возможности старайтесь не использовать память в куче (не пользоваться оператором `new`), если размещаемый в ней объект используется только в данной функции;
- если использование памяти из кучи неизбежно, создайте класс для работы с этой памятью, в который включите копирующий конструктор, деструктор,

освобождающий всю используемую данным классом память, и оператор присваивания для основных типов данных, которые будут использоваться с данным классом;

- если функция, запрашивающая выделение памяти, возвращает значение, позволяющее получить доступ к данной памяти, то это значение не должно быть ссылкой, поскольку освободить память по ссылке невозможно;
- никогда не изменяйте значения указателя на область памяти, пока эта область не будет освобождена;
- не используйте операцию приращения для указателя, полученного с использованием оператора `new`;
- используйте для хранения информации классы коллекций, описанные в *главе 11*.

Отладочные версии операторов *new* и *delete*

Как уже говорилось выше, утечка памяти представляет собой довольно коварную ошибку, которая может остаться невыявленной после тщательного тестирования приложения (не говоря уже о щадящем тестировании, которое устраивают многие программисты для своего детища). Поэтому так ценна возможность автоматического обнаружения утечек памяти, предоставляемая библиотекой MFC. В ней при создании отладочной версии приложения вместо операторов `new` и `delete` используются их специальные отладочные версии, отслеживающие имя файла и номер строки, в которых производится выделение данного фрагмента памяти.

По завершении работы приложения все операторы `delete` сопоставляются с сопряженными с ними операторами `new` и, в случае обнаружения несоответствия, в окно **Output** (Вывод) выводится сообщение, аналогичное показанному на рис. 14.13.

Как видно из диагностического сообщения, источник обеих ошибок указан один: конструктор класса. Однако даже эта информация значительно сужает круг поиска возможных причин возникновения утечки памяти.

При определении в программах C++ символа `_CRTDBG_MAP_ALLOC` вместе с функциями для работы с кучей выводятся их отладочные версии с именем исходного файла и номером строки. Отладочные версии этих функций выделяют память из так называемой *кучи отладки* (debug heap). Использование кучи отладки несколько снижает быстродействие и требует дополнительных ресурсов для проведения различных проверок. Кроме того, при этом выделяется больше памяти, чем запрашивалось. Компилятор "окружает" каждую выделенную область памяти зоной "недоступности", заполняя ее значениями `0xFD`. Операции, приводящие к выходу за границы выделенной памяти, попадают в эту маркированную область, вызывая сообщение об ошибке. Выделенная память заполняется значениями `0xCD`, а освобожденная память — значениями `0xDD`.

Получение информации об утечках памяти после завершения программы имеет свои существенные недостатки: программа может работать достаточно долго и для устранения утечек памяти хорошо бы определить, в какой момент она произошла и, кроме того, эта информация теряется в потоке другой информации и, если ее специально не искать, обнаружить ее практически невозможно. Поэтому в Visual C++ включены функции, позволяющие проверять корректность содержимого кучи отладки в процессе выполнения приложения. Среди них, прежде всего, следует упомянуть функцию `_CrtDumpMemoryLeaks`. Эта функция выводит ту же информацию об утечках памяти, что и библиотека MFC, но делает это в процессе выполнения программы, как это показано на рис. 14.14.

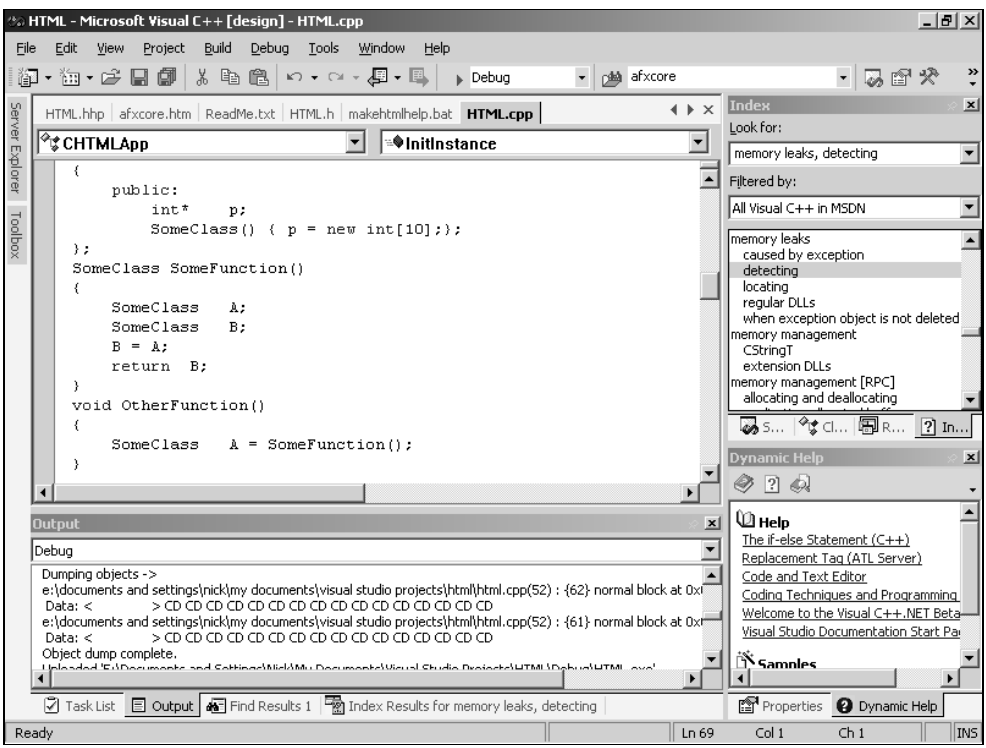


Рис. 14.13. Окно **Output** с сообщением об утечке памяти после завершения работы приложения

Если разработчику приложения необходимо просто убедиться в целостности кучи отладки, он может воспользоваться функцией `_CrtCheckMemory`. Кроме того, многие операции по проверке кучи могут быть реализованы функцией `_CrtSetDbgFlag`. Для вывода всех объектов, для которых в данный момент выделена память, используется функция `DumpAllObjectsSince`.

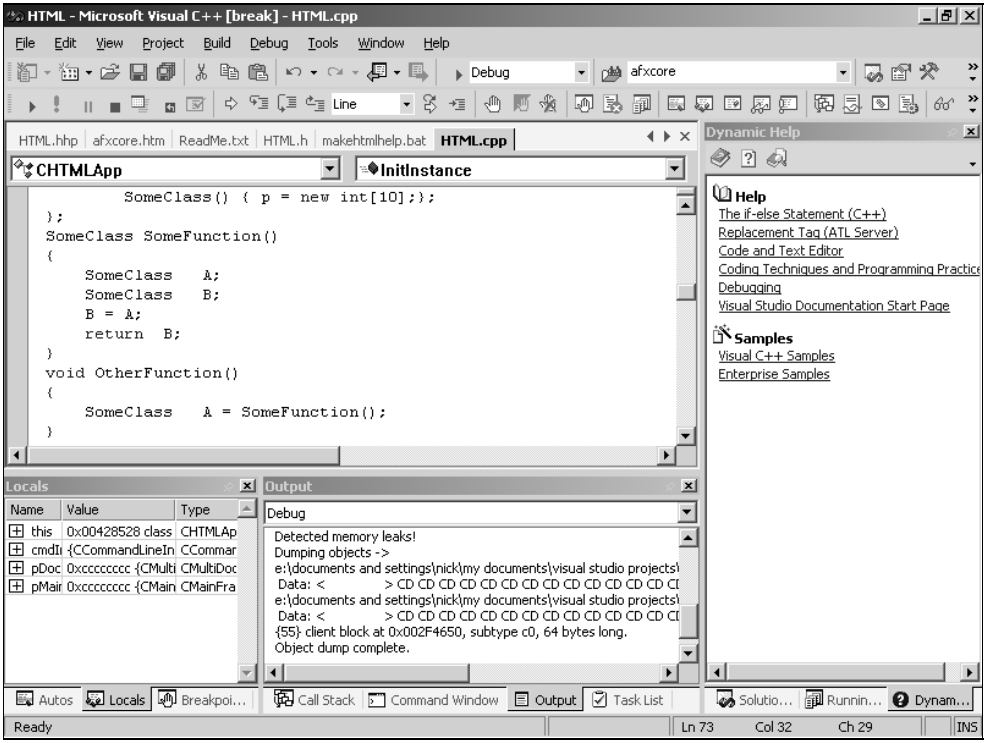
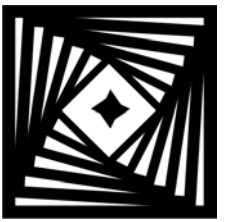


Рис. 14.14. Окно **Output** с сообщением об утечке памяти в процессе работы приложения

Если описанных выше средств недостаточно для выявления утечки памяти, разработчик может воспользоваться объектом класса `CMemoryState` и его функциями `Checkpoint` и `Difference`. Эти функции позволяют разработчику получить "моментальные снимки" памяти и идентифицировать последующие различия.

Глава 15



Создание пользовательской библиотеки

Любой программист, приступающий к написанию нового приложения, хотел бы максимально использовать в нем свои предыдущие наработки. Проще всего это желание реализуется в том случае, если он создал инструментарий для решения наиболее часто встречающихся задач и поместил его в свою библиотеку, которую можно подключать к любому приложению.

Пользователь может создать как статическую библиотеку, так и библиотеку динамической компоновки, однако различия между ними с точки зрения их программирования не столь существенны, чтобы рассматривать их раздельно. *Библиотеки динамической компоновки* (DLL), так же, как и статические библиотеки, представляют собой двоичные файлы, содержащие функции, а в некоторых случаях данные и ресурсы, которые могут быть использованы в приложении (или в другой библиотеке динамической компоновки). Существует несколько причин использования библиотек динамической компоновки, основными из которых являются:

- модульность приложения, облегчающая внесение изменений и повторное использование его составных частей;
- разделение библиотеки между несколькими одновременно работающими приложениями.

Прежде чем некоторый процесс сможет использовать библиотеку динамической компоновки, необходимо сначала зарузить в оперативную память файл образа библиотеки динамической компоновки и включить его в адресное пространство этого процесса. Это может быть сделано одним из двух способов: неявной компоновкой в процессе загрузки или явной компоновкой в процессе выполнения задачи.

В Visual C++ разделяют две основные категории библиотек динамической компоновки: те, которые используют библиотеку MFC, и те, которые ее не используют. Любая из этих библиотек может быть создана мастером Application Wizard. Поддержка, которую обеспечивают библиотекам динамической компоновки MFC и среда программирования Visual C++, существенно облегчает процесс их создания.

Использование библиотек динамической компоновки

Библиотека MFC скрывает большую часть функций, используемых для работы с библиотеками динамической компоновки. Для лучшего понимания их работы нам необходимо рассмотреть основные из этих функций и понять, каким образом Win32 интегрирует библиотеки динамической компоновки в процесс. Как уже говорилось выше, для того, чтобы процесс мог вызвать функцию из библиотеки динамической компоновки, эта библиотека должна быть размещена в виртуальном адресном пространстве процесса. После этого функции библиотеки динамической компоновки становятся доступными всем потокам данного процесса. Для потоков процесса функции библиотеки динамической компоновки ничем не отличаются от других функций, расположенных в адресном пространстве процесса. При вызове потоком функции из библиотеки динамической компоновки для передачи аргументов и локальных переменных используется стек потока. Кроме того, вызывающий поток или процесс становится собственником любого объекта, созданного функцией библиотеки динамической компоновки.

Главная функция библиотеки динамической компоновки

Библиотека динамической компоновки, как и любое другое приложение, обычно требует при своей загрузке некоторой инициализации. По умолчанию компоновщик назначает библиотеке динамической компоновки главную входную точку `_DllMainCRTStartup`. Эта функция вызывается Windows при загрузке библиотеки динамической компоновки для инициализации библиотек поддержки C/C++, вызова конструкторов C++ для статических и глобальных переменных, а также для выполнения других операций. Помимо инициализации библиотек поддержки C/C++ функция `_DllMainCRTStartup` вызывает функцию `DllMain`. Вид функции `DllMain` зависит от типа создаваемой библиотеки динамической компоновки. Ниже приведен пример фрагмента программы, созданного мастером Application Wizard:

```
BOOL APIENTRY DllMain( HANDLE hModule, DWORD  ul_reason_for_call, LPVOID
lpReserved )
{
  switch (ul_reason_for_call)
  {
    case DLL_PROCESS_ATTACH:
    case DLL_THREAD_ATTACH:
    case DLL_THREAD_DETACH:
    case DLL_PROCESS_DETACH:
    break;
  }
```

```
  return TRUE;
}
```

Первый аргумент, `hModule`, представляет собой дескриптор модуля экземпляра загружаемой библиотеки динамической компоновки. Поскольку библиотека динамической компоновки представляет собой отдельный модуль, она может иметь собственные ресурсы. В этом случае значение аргумента `hModule` должно сохраняться в оперативной памяти для дальнейшего использования.

Второй аргумент, `ul_reason_for_call`, указывает на причину вызова функции `DllMain`. Помимо того, что функция `DllMain` является исходной точкой загрузки, она вызывается также при завершении работы библиотеки динамической компоновки. При загрузке аргумент `ul_reason_for_call` имеет значение `DLL_PROCESS_ATTACH`, а при завершении работы — `DLL_PROCESS_DETACH`. Необходимо также помнить, что функция `DllMain` вызывается при создании или уничтожении потоком процесса вторичного потока. При создании потока аргумент `ul_reason_for_call` имеет значение `DLL_THREAD_ATTACH`, а при уничтожении потока — `DLL_THREAD_DETACH`.

Третий аргумент, `lpReserved`, предоставляет дополнительную информацию об инициализации и очистке.

Экспорт и импорт функций

Динамическая компоновка библиотеки предоставляет процессу возможность вызывать функцию, не являющуюся частью его исполняемого кода. Исполняемый код функции, расположенный в библиотеке динамической компоновки, может подключаться к процессу двумя способами: неявно, в процессе компоновки, или явно, в процессе выполнения. Динамическая компоновка производится посредством таблицы, содержащейся в библиотеке динамической компоновки. Эту таблицу обычно называют *экспортной таблицей* (exports table) или *таблицей имен* (name table). В ней содержатся символические имена или порядковые номера каждой функции и глобальной переменной, предоставляемой библиотекой динамической компоновки для использования другими процессами. Кроме того, в ней содержится адрес каждой предоставляемой функции. При первой загрузке процессом библиотеки динамической компоновки процесс не знает адресов вызываемых функций, но знает их символьные имена или порядковые адреса. В ходе динамической компоновки запросы приемника ставятся в соответствие адресам функций библиотеки динамической компоновки.

Экспортную таблицу в библиотеку динамической компоновки помещает компоновщик, но для этого программист должен предоставить ему информацию о том, какой символ (функцию или переменную) следует в нее поместить. Существует два способа предоставления этой информации компоновщику. Первый состоит в том, что список экспортируемых символов помещается в файл определения модуля (DEF). Файл определения модуля представляет собой текстовый файл, содержащий выражения для определения файлов EXE и DLL. Описание выражения определения модуля, приведенное в интерактивной справочной сис-

теме *Visual C++ Programmer Guide*, позволяет получить для каждого из выражений эквивалентную ему командную строку. Вторым способом сообщить компоновщику о том, какие символы следует включать в экспортную таблицу, является прямое указание на это в исходном тексте библиотеки динамической компоновки. Ниже приведен пример такого указания:

```
__declspec(dllexport) int Some_Func(int i);
```

В программе приемника необходимо в явном виде указать, какие функции следует импортировать. Ниже приведен пример такого указания:

```
__declspec(dllimport) int Some_Func(int i);
```

По соглашению для упрощения процедуры включения описаний библиотеки динамической компоновки используются файл заголовка и макрос препроцессора. Ниже приведен пример их использования:

```
// DemoDLL.H


#pragma once


#if !defined(_DEMODLL_)
#define DEMODLL_LIB    __declspec(dllimport)
#else
#define DEMODLL_LIB    __declspec(dllimport)
#endif // !defined(_DEMODLL_)
DEMODLL_LIB  int  Some_Func(int i);
```

Решение о том, следует ли экспортировать или импортировать функцию `Some_Func`, препроцессор принимает на основании информации об определенности символа `_DEMODLL_`. Это позволяет подключать один и тот же файл заголовка при создании и использовании библиотеки динамической компоновки. Единственное, что должен сделать разработчик библиотеки динамической компоновки, — это включить определение переменой `_DEMODLL_` в файл реализации перед инструкцией `include`, включающей соответствующий файл заголовка, как это показано ниже.

```
// DemoDLL.CPP
#define  _DEMODLL_
#include  "DemoDLL.h"
```

Примечание

Несмотря на простоту представленного выше подхода, многие производители программного обеспечения считают, что без использования файла DEF, обеспечивающего централизованное определение интерфейса, теряется контроль над интерфейсом. Файлы DEF обеспечивают краткость и понятность интерфейса, а также позволяют следить за взаимодействием его составляющих.

При компиляции библиотеки динамической компоновки компоновщик создает сопутствующий *файл импорта* (LIB), содержащий символические имена и порядковые значения всех экспортируемых символов. Файл LIB является суррогатом библиотеки динамической компоновки и должен включаться в проекты программ приемника для обеспечения возможности неявной компоновки.

Неявная компоновка

Как уже говорилось выше, существуют два способа подключения библиотек динамической компоновки: явный и неявный. Чаще всего используется неявная компоновка библиотеки динамической компоновки в пользовательское приложение. Это происходит по той простой причине, что при этом от пользователя практически ничего не требуется делать. Разработчик приложения просто включает в него требуемые файлы заголовков, вызывает необходимые функции и подключает к приложению статический файл LIB библиотеки динамической компоновки. При компоновке приложения импортируемые символы (или порядковые номера) сопоставляются с экспортируемыми символами в файле LIB, и эти символы включаются в файл EXE.

Имея эту информацию при загрузке приложения, Windows находит и загружает библиотеку динамической компоновки, а затем динамически связывает ее с символами или порядковыми номерами. При загрузке библиотеки динамической компоновки Windows просматривает папки в следующей последовательности.

1. Папка с исполняемым файлом приложения.
2. Текущая папка процесса.
3. Папка Windows.
4. Папка, указанная в системной переменной `PATH`.

Как правило, неявная компоновка библиотеки подходит для большинства случаев. Однако она имеет два недостатка, о которых следует постоянно помнить:

- при увеличении числа библиотек динамической компоновки увеличивается время загрузки приложения;
- библиотека динамической компоновки загружается даже в том случае, если ни одна из ее функций не будет вызвана.

Примечание

Чтобы при неявной компоновке использовать расширение файла, отличное от dll, необходимо перегрузить заданное по умолчанию имя файла, передаваемое компоновщику (`/OUT`). Этот ключ также позволяет изменить расположение и имя библиотеки динамической компоновки.

Явная компоновка

Явная компоновка предполагает непосредственное указание Windows, какой файл и когда следует загрузить. Чтобы выполнить явную компоновку, от разра-

ботчика приложения требуется выполнение дополнительных операций. Чтобы включить в приложение функции, используя явную компоновку:

1. Вызовите в приемнике функцию `::LoadLibrary`, передавая ей имя загружаемой библиотеки динамической компоновки. Если работа функции завершилась успешно, указанная библиотека помещается в адресное пространство вызывающего процесса и ее дескриптор возвращается приемнику. Функция `LoadLibrary` пытается найти библиотеку динамической компоновки в тех же папках и в той же последовательности, что и при неявной компоновке. При аварийном завершении работы функции она возвращает нулевое значение.
2. Вызовите в приемнике функцию `::GetProcAddress`, передавая ей дескриптор, возвращенный функцией `::LoadLibrary`, а также символическое имя или порядковый номер требуемой функции. В случае успешного завершения работы она возвращает адрес экспортируемой функции библиотеки динамической компоновки. При аварийном завершении работы функция `::GetProcAddress` возвращает нулевое значение. Для получения более подробной информации об ошибке можно вызвать функцию `::GetLastError`.
3. Повторите п. 2 для остальных функций, которые предполагается использовать в приемнике.
4. После завершения работы приемника с библиотекой динамической компоновки вызовите функцию `::FreeLibrary` для выгрузки библиотеки.

Описанная выше последовательность действий утомительна и может вызвать ошибки, но она позволяет разработчику приложения избежать недостатков, присущих неявной компоновке.

Использование для работы с библиотеками динамической компоновки

При создании библиотек динамической компоновки в MFC необходимо, прежде всего, определить, будет ли это регулярная библиотека динамической компоновки или библиотека динамической компоновки расширения MFC. При принятии решения придется ответить на следующие вопросы:

- Будет ли библиотека динамической компоновки использовать MFC, и содержит ли она классы, производные от классов MFC?
- Позволит ли библиотека динамической компоновки использовать в приложении ссылки на ее классы, производные от классов MFC?
- Будет ли библиотека динамической компоновки вызываться из других сред разработки, таких как Visual Basic или PowerBuilder?
- Какой тип компоновки (статическая или динамическая) будет использоваться библиотекой динамической компоновки при подключении ее к библиотеке MFC?

Регулярные библиотеки динамической компоновки MFC

Регулярные библиотеки динамической компоновки MFC (Regular MFC DLL) представляют собой библиотеки динамической компоновки, скрывающие наличие библиотеки MFC и обеспечивающие экспорт функций с использованием стиля C. Это означает, что следует тщательно выбирать имена функций и должным образом употреблять инструкцию `extern "C"`. Регулярные библиотеки динамической компоновки можно использовать в Visual Basic, Delphi и других языках программирования, не поддерживающих функций в стиле C.

При создании регулярной библиотеки динамической компоновки MFC необходимо определить, как она будет подключаться к MFC — статически или динамически. Метод подключения определяет необходимость поставки вместе с приложением библиотек динамической компоновки MFC.

Простейшим путем создания регулярной библиотеки динамической компоновки MFC является использование мастера MFC DLL Wizard. Для этого после вызова мастера MFC DLL Wizard (как его вызвать, будет описано позже) необходимо раскрыть вкладку **Application Settings** (Настройки приложения) диалогового окна **MFC DLL Wizard - MyDll** (Мастер создания библиотеки динамической компоновки MFC), изображенную на рис. 15.1, и установить переключатель **DLL type** (Тип библиотеки динамической компоновки) в соответствующее положение в зависимости от типа ее подключения к библиотеке MFC.

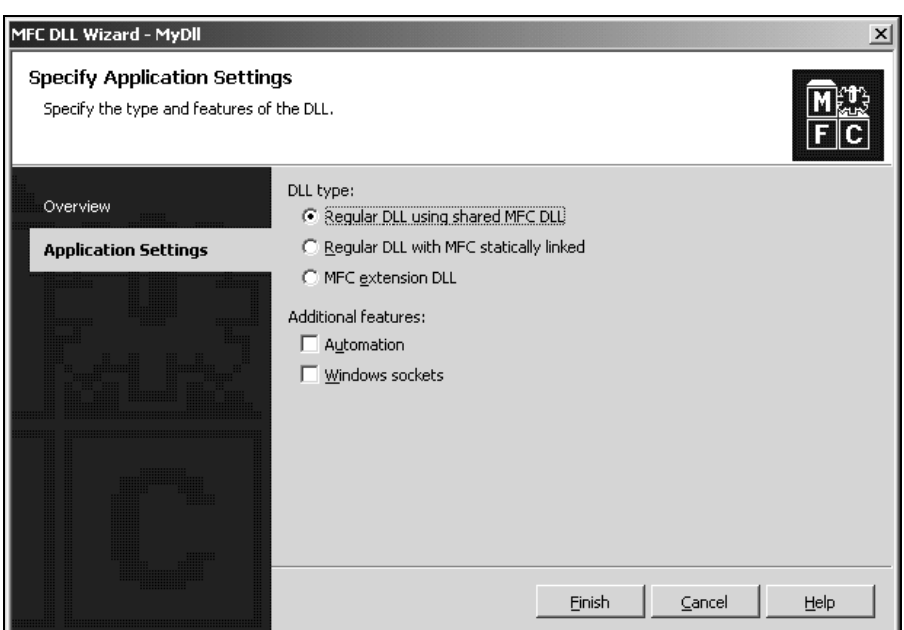


Рис. 15.1. Диалоговое окно **MFC DLL Wizard - MyDll**, вкладка **Application Settings**

В программе, созданной мастером MFC DLL Wizard, функция `DllMain` отсутствует, хотя без нее работа с библиотекой динамической компоновки невозможна. Оказывается, эта функция скрывается данной библиотекой, а пользовательский доступ к ней заканчивается на уровне класса, производного от класса `CWinApp` (и глобального объекта этого класса), как это имеет место и в программах EXE. Преимущество использования класса `CWinApp` состоит в том, что это позволяет программировать пользовательскую библиотеку динамической компоновки так же, как и любой класс, производный от класса `CWinApp`. Например, для инициализации и завершения работы библиотеки динамической компоновки могут использоваться функции `CWinApp::InitInstance` и `CWinApp::ExitInstance`. Функция `DllMain` вызывает функцию `InitInstance` при подключении библиотеки динамической компоновки процессом и функцию `ExitInstance` при ее отключении процессом.

После загрузки библиотеки динамической компоновки к ней может обращаться любая программа C или C++, не использующая MFC. При этом не требуется никакой дополнительной инициализации или защиты. Однако, если программа использует MFC и между программой и библиотекой динамической компоновки осуществляется обмен объектами MFC, при создании библиотеки динамической компоновки необходимо произвести некоторые дополнительные операции.

MFC хранит некоторую внутреннюю глобальную информацию о состоянии приложения и библиотеки динамической компоновки. Поэтому, если библиотека динамической компоновки динамически подключается к MFC, необходимо использовать макрос `AFX_MANAGE_STATE` для поддержания корректной информации о глобальном состоянии. Ниже приведен пример:

```
extern "C" int OnDlg(HWND hwParent)
{
  AFX_MANAGE_STATE(AfxGetStaticModuleState());
  CWnd *    lpParent = CWnd::FromHandle(hwParent);
  CDialog  myDlg(IDD_MY_DIALOG, pParent);
  return    myDlg.DoModal();
}
```

По умолчанию MFC загружает шаблон ресурсов с помощью дескриптора ресурсов главного приложения. Поскольку экспортируемая функция `OnDlg` вызывает диалоговое окно, опираясь на шаблон документа, хранящийся в модуле библиотеки динамической компоновки, для использования корректного дескриптора состояние модуля должно быть переключено.

Библиотеки динамической компоновки расширения MFC

В то время как регулярные библиотеки динамической компоновки MFC обычно используются приемниками, не поддерживающими MFC, *библиотеки динамической компоновки расширения MFC* (MFC extension DLL's) экспортируют

функции и классы, расширяющие возможности MFC. Например, предположим, что создан класс с именем `CPerson`, являющийся потомком класса MFC `CObject`. Для экспорта этого класса его необходимо поместить в библиотеку динамической компоновки расширения. Как и регулярные библиотеки динамической компоновки, библиотеки динамической компоновки расширения MFC могут создаваться мастером MFC DLL Wizard. Однако между этими двумя типами библиотек динамической компоновки существуют фундаментальные различия. Для понимания этих различий рассмотрим файл реализации, созданный мастером MFC DLL Wizard для библиотеки динамической компоновки расширения MFC (все комментарии и сообщения в данном файле переведены на русский язык):

```
// Файл Extension.cpp : Содержит процедуры инициализации DLL.
//

#include "stdafx.h"
#include <afxdllx.h>

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

static AFX_EXTENSION_MODULE ExtensionDLL = { NULL, NULL };

extern "C" int APIENTRY
DllMain(HINSTANCE hInstance, DWORD dwReason, LPVOID lpReserved)
{
  // Удалите следующую строку, если аргумент lpReserved используется
  UNREFERENCED_PARAMETER(lpReserved);

  if (dwReason == DLL_PROCESS_ATTACH)
  {
    TRACE0("Инициализация Extension.DLL!\n");



    // Инициализация библиотеки динамической компоновки
    // расширения MFC в процессе выполнения приложения
    if (!AfxInitExtensionModule(ExtensionDLL, hInstance))
      return 0;
```

```
    // Включите DLL в цепь ресурсов
    // ПРИМЕЧАНИЕ: Если библиотека динамической компоновки
    // расширения MFC явным образом подключается
    // к регулярной библиотеке динамической компоновки
    // расширения MFC (например, к элементу управления ActiveX),
    // а не как приложению MFC, то следующую строку
    // следует удалить из функции DllMain и поместить ее
    // в отдельную функцию, экспортируемую из данной библиотеки
    // динамической компоновки расширения MFC.
    // Регулярная библиотека динамической компоновки,
    // использующая данную библиотеку динамической компоновки
    // расширения MFC должна явным образом вызвать эту функцию
    // для инициализации библиотеки динамической компоновки
    // расширения MFC. В противном случае объект
    // CDynLinkLibrary будет подключен к цепи ресурсов
    // регулярной библиотеки динамической компоновки,
    // что может привести к серьезным последствиям.

    new CDynLinkLibrary(ExtensionDLL);

  }
  else if (dwReason == DLL_PROCESS_DETACH)
  {
    TRACE0("Extension.DLL Завершает работу!\n");

    // Закрытие библиотеки перед вызовом деструкторов
    AfxTermExtensionModule(ExtensionDLL);
  }
  return 1;  // Нормальное продолжение работы
}
```

Обратите внимание, что в отличие от регулярных библиотек динамической компоновки MFC, в библиотеке динамической компоновки расширения MFC отсутствует объект класса, производного от класса `CWinApp`. В нем, однако, присутствует функция `DllMain`.

Функция *DllMain*

Первым оператором функции `DllMain` является вызов макроса `UNREFERENCED_PARAMETER`. Этот макрос используется для отключения предупреждений транслятора; однако в Visual C++ 7.0 в нем нет необходимости.

После этого стоит условный оператор, определяющий операцию, производимую с библиотекой динамической компоновки (подключение к процессу или отключение от него). При подключении процесса необходимо инициализировать состояние модуля библиотеки динамической компоновки. Эту операцию производит функция `AfxInitExtensionModule` с двумя аргументами: ссылкой на объект структуры `AFX_EXTENSION_MODULE` и дескриптором экземпляра библиотеки динамической компоновки. В процессе инициализации в объект структуры `AFX_EXTENSION_MODULE` записывается состояние библиотеки динамической компоновки, позволяющее библиотеке корректно работать с MFC. Если функция `AfxInitExtensionModule` возвращает значение `FALSE`, то произошло что-то поистине ужасное, поэтому по нулевому значению производится выход из функции `DllMain`. Если функция `AfxInitExtensionModule` возвращает значение `TRUE`, это означает, что создан объект класса `CDynLinkLibrary`. Объект этого класса позволяет библиотеке динамической компоновки расширения экспортировать объекты класса `CRuntimeClass` или свои ресурсы в приложение приемника. Оболочка поддерживает связный список объектов класса `CDynLinkLibrary`.

При отключении библиотеки динамической компоновки от процесса необходимо освободить любые локальные области памяти, выделенные модулю, и удалить все входные точки в кэше карты сообщений. Эти операции производятся функцией `AfxTermExtensionModule`.

Если в функции `DllMain()` необходимо произвести некоторые дополнительные операции, все они при подключении должны производиться после вызова функции `AfxInitExtensionModule`, а при отключении — до вызова функции `AfxTermExtensionModule`.

Экспорт классов и функций

Как правило, библиотеки динамической компоновки расширения MFC создаются для экспорта классов, производных от классов библиотеки MFC. Чтобы сделать класс доступным, необходимо добавить класс C++ в проект и ввести при его объявлении макрос `AFX_EXT_CLASS`, как это показано ниже:

```
class AFX_EXT_CLASS CPerson : public CObject
```

Эта модификация должна быть произведена в файле заголовка класса. В проекте библиотеки динамической компоновки и в проекте приемника должен использоваться один и тот же файл заголовка. Макрос `AFX_EXT_CLASS` создает различные коды в различных ситуациях: он экспортирует класс при создании библиотеки динамической компоновки и импортирует класс в приложение приемника.

Иногда нет необходимости экспортировать весь класс. В этом случае все равно используется макрос `AFX_EXT_CLASS`, но вместо того, чтобы помещать его перед именем класса, его просто указывают перед именем экспортируемой функции:

```
class CPerson : public CObject
{
```

```
public:
  AFX_EXT_CLASS CPerson(LPCTSTR name, UINT id, double salary);
  CPerson ();
  CPerson(CPerson & aPerson);
. . .
}
```

Обратите внимание, что экспортируется только первый конструктор класса, а второй и третий конструкторы не экспортируются. Это означает, что приемник может создать объект класса `CPerson` (то есть нового служащего), если известны его имя, идентификатор и жалование. Однако приемник при создании объектов класса `CPerson` не может использовать ни конструктор по умолчанию, ни конструктор копий.

Передача ресурсов

Как уже отмечалось, в библиотеке динамической компоновки могут содержаться ресурсы. Для доступа к этим ресурсам используется дескриптор содержащего их модуля. Процесс получения дескриптора в MFC существенно упрощается благодаря использованию функции `AfxGetResourceHandle`, которая возвращает указанный дескриптор. При работе в контексте библиотеки динамической компоновки MFC возвращает дескриптор модуля библиотеки динамической компоновки; в противном случае она возвращает дескриптор модуля выполняемого приложения. После получения дескриптора модуля он может использоваться в функциях `LoadResource` и `FindResource`.

Другая ситуация, в которой может потребоваться вызов функции `AfxGetResourceHandle`, возникает при попытке установки заданного по умолчанию модуля, используемого MFC для размещения в нем требуемых ресурсов. Потребность в этой установке возникает в двух случаях: при необходимости ускорить доступ к ресурсам и при наличии в различных модулях двух похожих ресурсов с одним и тем же идентификатором.

Как только приложение требует от MFC загрузить ресурс, MFC сначала пытается загрузить ресурс из текущего модуля. Если искомый ресурс не найден в текущем модуле, MFC просматривает связанный с приложением список объектов класса `CDynLinkLibrary` и пытается найти ресурс там. Просмотр списка означает, что ресурс, находящийся в его начале, будет найден быстрее, чем ресурс, находящийся в его конце. Это означает также, что ресурс, имеющий тот же идентификатор, что и просмотренный ранее ресурс, будет скрыт. К счастью, последовательность поиска может быть изменена. Предположим, что первыми следует просмотреть библиотеки динамической компоновки расширения. Ниже приведен пример решения этой задачи:

```
// Получение дескриптора ресурса исполняемого файла
HINSTANCE hInstance = AfxGetResourceHandle();
```

```
// Использование вместо него дескриптора библиотеки
// динамической компоновки
AfxSetResourceHandle(::GetModuleHandle("MyLib.dll"));
CString  szString;
szString.LoadString(IDS_SOME_RESOURCE);

// Восстановление дескриптора ресурса исполняемого файла
AfxSetResourceHandle(::GetModuleHandle(hInstance));
```

Примечание

Библиотеки динамической компоновки расширения MFC могут перемещать указатели на объекты, являющиеся потомками объектов MFC, в пределах приложения библиотеки динамической компоновки, а регулярные библиотеки динамической компоновки MFC этого делать не могут.

Использование файлов DEF

Файлы определения модуля (def), как уже говорилось выше, используются для передачи компоновщику информации об экспортируемых функциях, атрибутах и другой информации, связанной с компонуемыми программами. Чаще всего файлы определения модуля используются при работе с библиотеками динамической компоновки. Использование этих файлов не является обязательным, но оно позволяет получить полный контроль над интерфейсом библиотеки динамической компоновки, и существенно упрощает работу с ней для конечного пользователя.

Включение файла определения модуля в приложение, не экспортирующее информацию, увеличивает размеры исполняемого файла и замедляет загрузку приложения.

При работе с файлом определения модуля следует помнить о следующем.

- Выражения, ключевые слова атрибутов и определенные пользователем идентификаторы используют символы различных регистров, поэтому заглавная и строчная буквы считаются различными символами.
- Длинные имена файлов, содержащие пробелы или символ точки с запятой (`;`) должны помещаться в кавычки (`"`).
- Для отделения ключевых слов выражения от его аргументов, а также для разделения выражений между собой могут использоваться один или несколько пробелов, знак табуляции или знак перевода строки. Знак двоеточия (`;`) или знак равенства (=), предшествующие аргументу, могут выделяться любым количеством пробелов, знаков табуляции или знаков перевода строки, а могут и никак не выделяться.
- Если в файле используется выражение `NAME` или `LIBRARY`, то оно должно предшествовать всем остальным выражениям.

- Выражения SECTIONS и EXPORTS могут использоваться многократно. Каждое выражение может содержать несколько спецификаций, разделяемых одним или несколькими пробелами, знаками табуляции или знаками перевода строки. Ключевое слово в этих выражениях должно стоять перед его первой спецификацией и повторяться перед каждой дополнительной спецификацией.
- Многие выражения имеют эквивалентные им ключи командной строки.
- Строки комментариев в файле определения модуля начинаются с символа точки с запятой (;). Комментарий не может располагаться в одной строке с выражением, но может располагаться между спецификациями выражений, занимающих несколько строк (например SECTIONS и EXPORTS).
- Числовые значения аргументов задаются в десятичной или в шестнадцатеричной системе счисления.
- Если значения строковых аргументов совпадают с зарезервированными словами, они должны заключаться в кавычки (").

В файле определения модуля могут использоваться следующие выражения:

- DESCRIPTION — имеет формат DESCRIPTION "text". Это выражение помещает строку в секцию .rdata. Текст в данном выражении может быть помещен в одиночные (') или двойные (") кавычки. Для использования этих символов в тексте достаточно заключить всю строку в кавычки другого типа.
- EXETYPE — имеет формат EXETYPE:dynamic | dev386. Это выражение может использоваться только в файле определения модуля драйвера виртуального устройства (VxD). Если в файле определения модуля драйвера виртуального устройства отсутствует выражение EXETYPE и в командной строке отсутствует ключ компилятора /EXETYPE, используется статическая загрузка (dev386).
- EXPORTS — имеет формат

  ```
  EXPORTS
  definitions
  ```

  Выражение EXPORTS используется для обозначения секции, содержащей одно или несколько определений, используемых при экспорте функций или данных. Каждое определение должно располагаться в отдельной строке. Ключевое слово EXPORTS может располагаться на той же строке, что и первое определение, или на предыдущей строке. Файл определения модуля может содержать одно или несколько выражений EXPORTS.

  Определение, используемое в данном выражении, имеет следующий синтаксис:

  ```
  entryname[=internalname] [@ordinal [NONAME]] [PRIVATE] [DATA],
  ```

  где:

  - entryname — имя экспортируемой функции или переменной. Этот компонент является обязательной частью определения;
  - Internalname — имя экспортируемой функции в библиотеке динамической компоновки. Указывается в том случае, если экспортируемое имя

отличается от имени данного компонента в библиотеке динамической компоновки. Если экспортируемая функция в библиотеке динамической компоновки имеет имя `Some_Func`, а она должна использоваться под именем `Other_Func`, следует использовать следующее выражение:

```
EXPORTS
Other_Func = Some_Func
```

- `@ordinal` — устанавливает порядковый номер, который будет помещен в экспортную таблицу библиотеки динамической компоновки вместо имени функции. Это позволит уменьшить размер библиотеки динамической компоновки. При этом файл lib будет содержать таблицу соответствия между порядковыми номерами и функциями, что позволит использовать имена функций, как будто бы в библиотеке динамической компоновки ничего не изменилось
- `NONAME` — необязательный параметр, позволяющий экспортировать компоненты библиотеки динамической компоновки только по ее имени, что позволяет уменьшить размер экспортной таблицы в данной библиотеке. Однако в этом случае при вызове функции `GetProcAddress` необходимо указывать порядковый номер компонента, поскольку его имя не может использоваться.
- `PRIVATE` — необязательный параметр, препятствующий записи `entryname` в файл импорта, создаваемый компоновщиком. Этот параметр не оказывает никакого влияния на экспорт в файл образа, который также создается компоновщиком.
- `DATA` — необязательный параметр, указывающий на то, что экспортируются данные, а не программа. Например, для экспорта переменной может использоваться следующее выражение:

  ```
  EXPORTS
  i DATA
  ```

  Если в определении одновременно присутствуют ключевые слова `PRIVATE` и `DATA`, то ключевое слово `PRIVATE` должно предшествовать ключевому слову `DATA`.

Существует три метода экспорта компонентов библиотек динамической компоновки. Ниже они расположены в порядке целесообразности их использования:

- директива `__declspec(dllexport)` в тексте программы;
- выражение `EXPORTS` в файле определения модуля;
- ключ `/EXPORTS` в команде LINK.

В одной и той же программе одновременно могут использоваться все три метода. Если при компиляции используется файл определения модуля, компилятор создает импортную библиотеку.

Ниже приведен пример секции EXPORTS.

```
EXPORTS
  DllCanUnloadNow  @1  PRIVATE  DATA
  DllWindowName = Name        DATA
  DllGetClassObject  @4  NONAME  PRIVATE
  DllRegisterServer  @7
  DllUnregisterServer
```

Экспорт компонента библиотеки динамической компоновки в файле определения модуля делает излишним использование для него директивы __declspec(dllexport). Однако в любом файле, применяющем библиотеку динамической компоновки, необходимо использовать директиву __declspec(dllimport).

- LIBRARY — имеет формат LIBRARY [library][BASE=address]. Это выражение указывает компилятору на необходимость создания библиотеки динамической компоновки. При этом, если не указан файл определения модуля, компилятор создает импортную библиотеку.

  Аргумент library определяет имя библиотеки динамической компоновки. Ту же самую функцию выполняет ключ компилятора /OUT.

  Аргумент BASE=address устанавливает адрес, начиная с которого операционная система должна загружать библиотеку динамической компоновки. По умолчанию библиотека динамической компоновки загружается с адреса 0x10000000. Ту же самую функцию выполняет ключ компилятора /BASE.

- NAME — имеет формат NAME [application][BASE=address]. Определяет имя главного выходного файла. Ту же самую функцию выполняет ключ компилятора /OUT. Если в компиляторе установлен ключ /OUT, а в файле определения модуля присутствует выражение NAME, значение этого выражения игнорируется. При создании библиотеки динамической компоновки выражение NAME определяет только имя файла библиотеки динамической компоновки.

- SECTIONS — имеет формат

  ```
  SECTIONS
  definitions
  ```

  Выражение SECTIONS создает секцию из одного или нескольких определений, устанавливающих режим доступа к разделам выходного файла проекта. Каждое определение должно располагаться в отдельной строке. Ключевое слово SECTIONS может располагаться на той же строке, что и первое определение, или на предыдущей строке. Файл определения модуля может содержать одно или несколько выражений SECTIONS. Вместо ключевого слова SECTIONS может использоваться ключевое слово SEGMENTS.

  Выражение SECTIONS устанавливает атрибуты для одной или нескольких секций файла образа. Оно может быть использовано для изменения атрибутов, установленных по умолчанию для каждого типа секции.

Определение в данном выражении имеет следующий формат:

```
.section_name specifier ,
```

где

- `.section_name` — имя секции в образе программы;
- `specifier` — один или несколько модификаторов доступа из следующего списка:
  - ◊ `EXECUTE`
  - ◊ `READ`
  - ◊ `SHARED`
  - ◊ `WRITE`

Модификаторы доступа в списке разделяются пробелами. Например:

```
SECTIONS
.rdata READ WRITE
```

Для поддержки предыдущих версий Visual C++ может использоваться следующий синтаксис:

```
section [CLASS 'classname'] specifier
```

Ключевое слово `CLASS` поддерживается для совместимости, но при синтаксическом анализе игнорируется.

Для задания атрибутов секции может использоваться ключ компилятора `/SECTION`.

- ❒ `STACKSIZE` — имеет формат `STACKSIZE reserve[,commit]`. Определяет размер стека в байтах. Для решения этой задачи может использоваться и ключ компилятора `/STACK`. Это выражение не оказывает никакого влияния на библиотеки динамической компоновки.
- ❒ `STUB` — имеет формат `STUB:filename`. В файле определения модуля, используемом при создании драйвера виртуального устройства (VxD), выражение `STUB` позволяет определить имя файла, содержащего объект структуры `IMAGE_DOS_HEADER` (определенной в файле winnt.h), используемого в VxD, а не заголовок, используемый по умолчанию. Для решения этой задачи может применяться и ключ компилятора `/STUB`.
- ❒ `VERSION` — имеет формат `VERSION major[.minor]`. Это выражение сообщает компоновщику о необходимости поместить в файл заголовка исполняемого файла или файла библиотеки динамической компоновки некоторое число. Аргументы `major` и `minor` представляют собой десятичные числа в диапазоне от 0 до 65 535. По умолчанию указывается версия 0.0. Для указания номера версии может быть использован ключ компилятора `/VERSION`.
- ❒ `VXD` — имеет формат `VXD filename`. Позволяет указать имя драйвера виртуального устройства (VxD). По умолчанию VxD получает то же имя, что и

первый объектный файл. Вместо данного выражения можно использовать ключ компилятора `/VXD` для задания драйвера виртуального устройства и ключ компилятора `/OUT` для задания имени выходного файла.

Пример создания и использования библиотеки динамической компоновки

Данный пример состоит из трех приложений, два из которых являются библиотеками динамической компоновки. Первое приложение представляет собой регулярную библиотеку динамической компоновки, содержащую функцию и класс. Второе приложение создает библиотеку динамической компоновки расширения MFC, содержащую ресурс диалогового окна и связанный с ним класс. Второе приложение использует библиотеку динамической компоновки, создаваемую первым приложением. Третье приложение используется для работы с диалоговым окном, создаваемым вторым приложением.

Создание регулярной библиотеки динамической компоновки

Исходный текст регулярной библиотеки динамической компоновки Library расположен в одноименной папке на дискете, прилагаемой к данной книге. Чтобы самостоятельно создать регулярную библиотеку динамической компоновки в среде программирования Visual C++:

1. Выберите команду меню **File | New | Project** (Файл | Создать | Проект), нажмите комбинацию клавиш <Ctrl>+<Shift>+<N> или кнопку **New Project** (Новый проект) на панели инструментов **Standard** (Стандартная). Появится диалоговое окно **New Project** (Новый проект), изображенное на рис. 15.2.
2. В окне **Templates** (Шаблоны) выделите значок **MFC DLL** (Библиотеки динамической компоновки MFC), введите в текстовое поле **Name** (Имя) имя проекта "Library" и нажмите кнопку **OK**. Появится диалоговое окно **MFC DLL Wizard - Library** (Мастер создания библиотеки динамической компоновки MFC), изображенное на рис. 15.3.
3. Нажмите кнопку **Finish** (Готово). Мастер MFC DLL Wizard (Мастер создания библиотеки динамической компоновки MFC) создаст заготовку нового приложения.
4. Откройте окно **Class View** (Просмотр классов) и щелкните правой кнопкой мыши на папке **Library** (Библиотека).
5. Выберите в появившемся контекстном меню команду **Add | Add Class** (Добавить | Добавить класс). Появится диалоговое окно **Add Class - Library** (Добавить класс), изображенное на рис. 15.4.

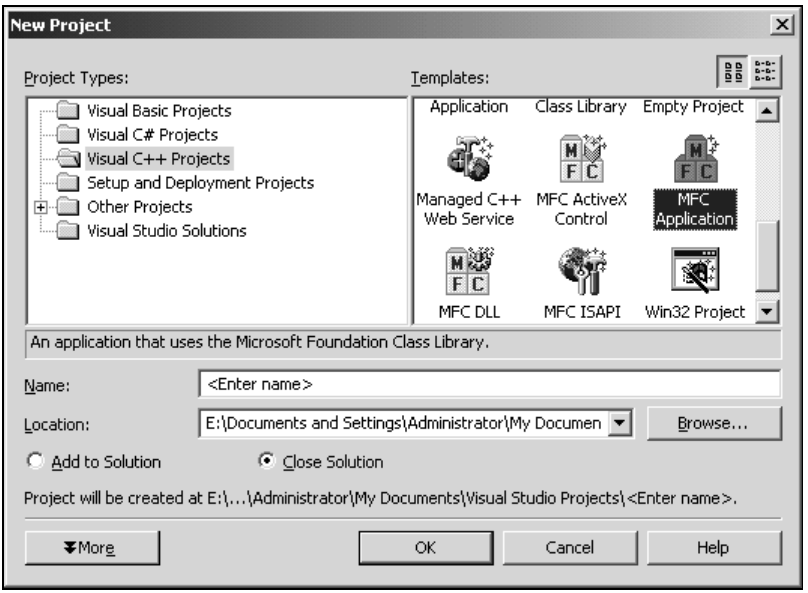


Рис. 15.2. Диалоговое окно **New Project**

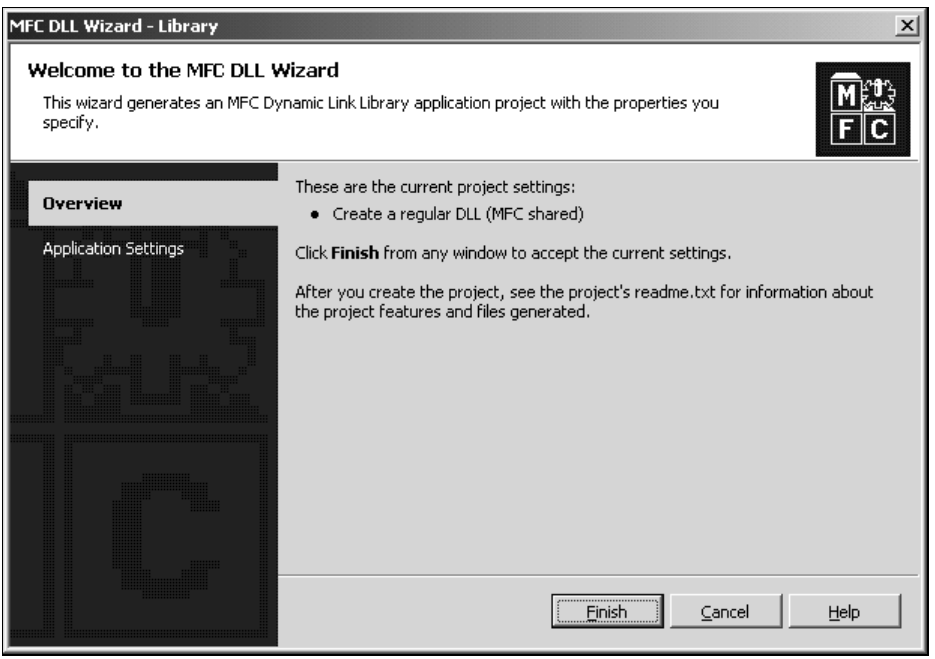


Рис. 15.3. Диалоговое окно **MFC DLL Wizard - Library**

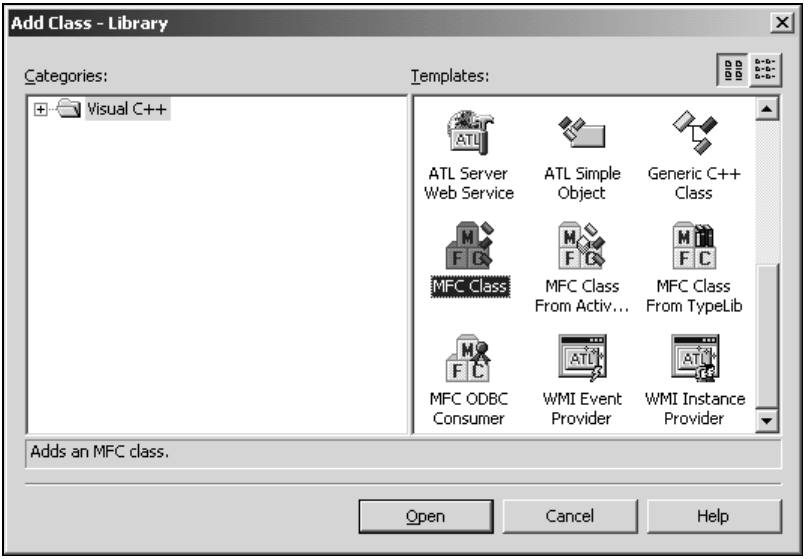


Рис. 15.4. Диалоговое окно **Add Class - Library**

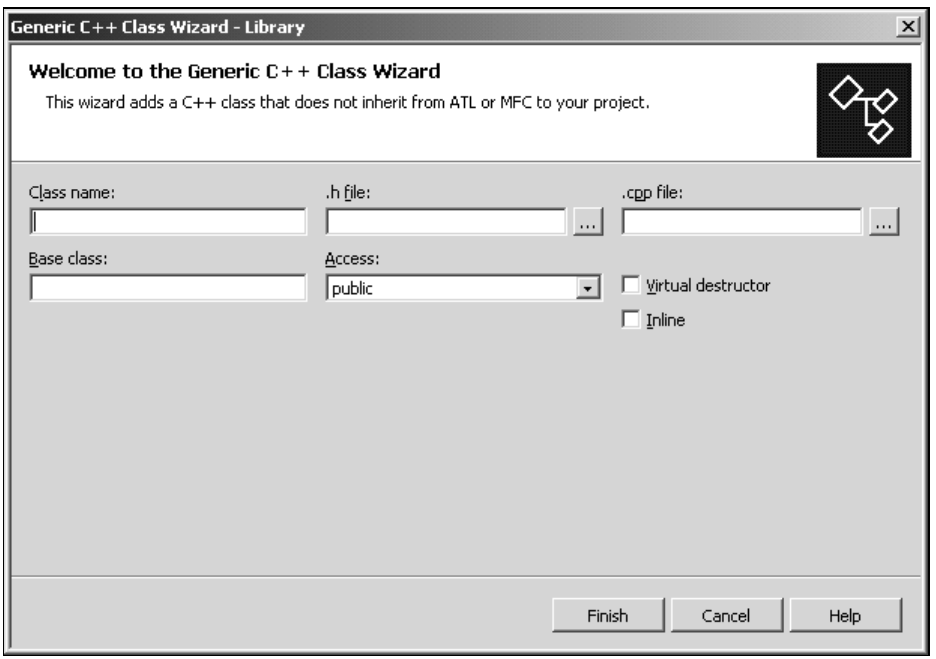


Рис. 15.5. Диалоговое окно **Generic C++ Class Wizard - Library**

6. В окне списка **Templates** (Шаблоны) выделите значок **Generic C++ Class** (Обобщенный класс) и нажмите кнопку **Open** (Открыть). Появится диалоговое окно **Generic C++ Class Wizard - Library** (Мастер создания обобщенного класса), изображенное на рис. 15.5.
7. Введите в текстовое поле **Class name** (Имя класса) имя класса `Median`, установите флажок **Virtual destructor** (Виртуальный деструктор) и нажмите кнопку **Finish** (Готово). В приложение будет включен новый класс.
8. В окне **Class View** (Просмотр классов) раскройте папку **Library** (Библиотека) и дважды щелкните левой кнопкой мыши на папке **Median**. Откроется окно редактирования файла Median.h.
9. Замените текст этого файла текстом листинга 15.1.

Листинг 15.1. Файл Median.h

```
 #pragma once

// Макросы задания режима включения объявлений функций и классов
#ifndef __DEMO_LIB
#ifndef __MY_LIBRARY__
#define __DEMO_LIB  __declspec(dllimport)
#else
#define __DEMO_LIB  __declspec(dllexport)
#endif
#endif

// Объявление шаблона функции
template<class Type>
__DEMO_LIB  void  Sort( Type*, int, int );
template<class Type>
void  Sort( Type* Buffer, int n)  { Sort( Buffer, 0, n-1 );};

// Объявление шаблона класса
template<class Type>
class  __DEMO_LIB  Median
{
  int        ah;
  int        m;
  double      mcu;
  double      mcl;
  double      mv;
  double*      x;
```

```
public :
  Median  ( int );
  virtual ~Median(void);

  Type  Get ( Type );
};
```

10. Раскройте папку `Median<Type>` и дважды щелкните левой кнопкой мыши на имени конструктора класса. Откроется окно редактирования файла Median.cpp.
11. Поместите в этот файл текст листинга 15.2.

Листинг 15.2. Файл Median.cpp

```
#include "StdAfx.h"

#ifndef __MY_LIBRARY__
#define __MY_LIBRARY__
#endif

#include "median.h"

template                          void Sort<int>( int*, int, int );
template  __declspec(dllexport)  void Sort<double>( double*, int, int );
template  class                         Median<int>;
template  class  __declspec(dllexport)  Median<double>;

// Сортировка по убыванию (больше номер — меньше величина)
template<class Type>
void  Sort( Type* Buffer, int lo, int hi )
{
  int   i = lo;
  int    j = hi;
  Type  x = Buffer[ (lo+hi) >> 1 ];

  while( i < j )
  {
    while( Buffer[i] > x )
      i++;
    while( Buffer[j] < x )
      j--;
```

```
    if( i < j )
    {
      Type y = Buffer[i];
      Buffer[i++] = Buffer[j];
      Buffer[j--] = y;
    }
  }
  if( lo < hi — 1 )
  {
    if ( lo < j )
      Sort( Buffer, lo, j );
    if ( i < hi )
      Sort( Buffer, i, hi );
  }
}

//////////////////////////////////////////////////////////////////
// Конструкторы и деструкторы
//////////////////////////////////////////////////////////////////

template<class Type>
Median<Type>:: Median ( int a )
{
  ah  = (a >> 1) + 1;
  x  = new double[ah];
  m  = 0;
  mcu = 0.0;
  mcl = 0.0;
  mv  = 0.0;

  for(int i=0; i < ah; i++)
    x[i] = 0.0;
}

template<class Type>
Median<Type>:: ~Median ( void )
{
  delete  x;
}
```

```
////////////////////////////////////////////////////////////////////
// Реализация методов
////////////////////////////////////////////////////////////////////

template<class Type>
Type  Median<Type>:: Get ( Type is )
{
  int    i;
  double  lv;

  m    = (m + 1) % ah;
  lv    = x[m];
  x[m]  = is;

  if( is > mcu )
    mcu = is;

  if( is < mcl )
    mcl = is;

  if( lv >= mcu )
  {
    mcu = x[0];

    for( i=1; i < ah; i++ )
    {
      if( x[i] > mcu )
        mcu = x[i];
    }
  }

  if( lv <= mcl )
  {
    mcl = x[0];

    for( i=1; i < ah; i++ )
    {
      if( x[i] < mcl )
        mcl = x[i];
    }
  }
```

```
  if( mv > mcu )
    mv = mcu;

  if( mv < mcl )
    mv = mcl;

  return  (Type) mv;
}
```

12. Раскройте окно редактирования файла Library.def, доступное как вкладка окна редактирования.
13. Поместите в этот файл текст листинга 15.3.

Листинг 15.3. Файл Library.def

```
; Library.def : Declares the module parameters for the DLL.
LIBRARY      "Library"
EXPORTS
    ; Explicit exports can go here
?Sort@@YAXPAHHH@Z
?Get@?$Median@H@@QAEHH@Z
```

14. Выберите команду меню **Build | Build Solution** (Создать | Компоновать решение) или нажмите комбинацию клавиш <Ctrl>+<Shift>+<B>. Библиотека динамической компоновки будет скомпилирована.

Создание библиотеки динамической компоновки расширения MFC

Исходный текст библиотеки динамической компоновки расширения MFC Extension расположен в одноименной папке на дискете, прилагаемой к данной книге. Чтобы самостоятельно создать библиотеку расширения MFC:

1. Выберите команду меню **File | New | Project** (Файл | Создать | Проект), нажмите комбинацию клавиш <Ctrl>+<Shift>+<N> или кнопку **New Project** (Новый проект) на панели инструментов **Standard** (Стандартная). Появится диалоговое окно **New Project** (Новый проект).
2. В окне списка **Templates** (Шаблоны) выделите значок **MFC DLL** (Библиотека динамической компоновки MFC), введите в текстовое поле **Name** (Имя) имя проекта "Extension" и нажмите кнопку **OK**. Появится диалоговое окно **MFC DLL Wizard - Library** (Мастер создания библиотеки динамической компоновки MFC).

3. Раскройте вкладку **Application Settings** (Настройки приложения), установите переключатель **DLL Type** (Тип библиотеки динамической компоновки) в положение **MFC extension DLL** (Расширение библиотеки динамической компоновки MFC) и нажмите кнопку **Finish** (Готово).
4. Откройте окно **Resource View** (Просмотр ресурсов), щелкните правой кнопкой мыши на папке **Extension.rc** и выберите в появившемся контекстном меню команду **Add Resource** (Добавить ресурс). Появится диалоговое окно **Add Resource** (Добавить ресурс), изображенное на рис. 15.6.

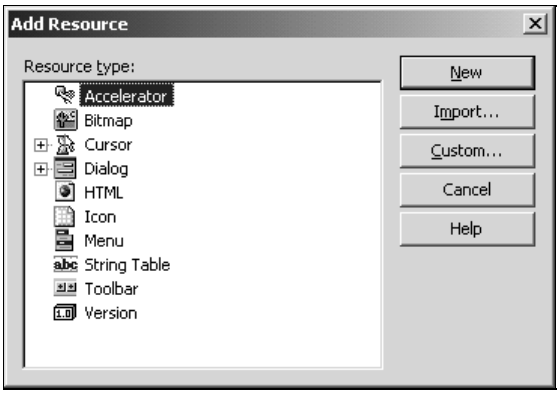


Рис. 15.6. Диалоговое окно **Add Resource**

5. В окне списка **Resource type** (Тип ресурса) выделите значок **Dialog** (Диалоговое окно) и нажмите кнопку **New** (Создать). В окне **Resource View** (Просмотр ресурсов) раскроются папки **Extension.rc** и **Dialog** и будет выделен идентификатор ресурса `IDD_DIALOG1`. Кроме того, будет раскрыто окно редактирования ресурса диалогового окна.
6. В окне **Resource View** (Просмотр ресурса) щелкните правой кнопкой мыши на идентификаторе ресурса `IDD_DIALOG1` и выберите в появившемся контекстном меню команду **Properties** (Свойства). Откроется диалоговое окно **Properties** (Свойства).
7. В текстовое поле раскрывающегося списка **ID** (Идентификатор ресурса) окна **Properties** (Свойства) введите идентификатор ресурса `IDD_MEDIAN`.
8. Щелкните левой кнопкой мыши на заготовке диалогового окна и введите в текстовое поле **Caption** (Заголовок) окна **Properties** (Свойства) текст "Медиана".
9. Растяните заготовку диалогового окна до размера 200×300 и переместите кнопки **OK** и **Cancel** (Отмена) в нижнюю часть заготовки диалогового окна.
10. Выделите кнопку **Cancel** (Отмена) и введите в текстовое поле **Caption** (Заголовок) окна **Properties** (Свойства) текст "Отмена".

11. Раскройте вкладку **Toolbox** (Инструментарий), выделите в нем элемент управления **Static Text** (Статический текст) и перетащите его в левую верхнюю часть заготовки диалогового окна.
12. Введите в текстовое поле **Caption** (Заголовок) окна **Properties** (Свойства) текст "Массив".
13. Во вкладке **Toolbox** (Инструментарий) выделите элемент управления **Edit Control** (Текстовое поле) и перетащите его под только что введенный статический текст.
14. Растяните текстовое поле по высоте на все свободное место.
15. В текстовое поле раскрывающегося списка **ID** (Идентификатор ресурса) окна **Properties** (Свойства) введите идентификатор ресурса `IDC_ARRAY`.
16. В раскрывающемся списке **MultiLine** (Несколько строк), расположенного в группе **Behavior** (Поведение), выделите строку **True**.
17. Произведите ту же операцию в раскрывающемся списке **ReadOnly** (Только для чтения), расположенного в той же группе, и в раскрывающихся списках **VerticalScroll** (Вертикальная полоса прокрутки), **StaticEdge** (Статическая рамка), **Number** (Число) и **RightAlignText** (Выравнивание по правому краю), расположенных в группе **Appearance** (Внешний вид).
18. Повторите п.п. 11—12 для статического текста "Апертура", поместив его по центру в верхней части заготовки диалогового окна.
19. Во вкладке **Toolbox** (Инструментарий) выделите элемент управления **Edit Control** (Текстовое поле) и перетащите его под только что введенный статический текст.
20. В текстовое поле раскрывающегося списка **ID** (Идентификатор ресурса) окна **Properties** (Свойства) введите идентификатор ресурса `IDC_APERTURE`.
21. В раскрывающемся списке **MultiLine** (Несколько строк), расположенного в группе **Behavior** (Поведение), выделите строку **True**.
22. Повторите п.п. 18—21 для создания в правой верхней части заготовки диалогового окна текстового поля с идентификатором `IDC_VALUE` и заголовком "Новое".
23. Повторите п.п. 18—21 для создания под текстовым полем **Апертура** нового текстового поля с идентификатором `IDC_RECURSIVE` и заголовком "Рекурсивная".
24. В раскрывающемся списке **ReadOnly** (Только для чтения), выделите строку **True**.
25. Повторите п.п. 23 и 24 для создания под текстовым полем **Новое** текстового поля с идентификатором `IDC_SIMPLE` и заголовком "Обычная".
26. Во вкладке **Toolbox** (Инструментарий) выделите элемент управления **Button** (Кнопка) и перетащите в заготовку диалогового окна, поместив справа от кнопки **Отмена**.

27. Введите в текстовое поле раскрывающегося списка **ID** (Идентификатор ресурса) идентификатор ресурса `IDC_ADD`, а в текстовое поле раскрывающегося списка **Caption** (Заголовок) введите заголовок "Добавить".
28. Щелкните правой кнопкой мыши в окне редактирования ресурса диалогового окна и выберите в появившемся контекстном меню команду **Add Class** (Добавить класс). Появится диалоговое окно **Add Class - Library** (Добавить класс), изображенное на рис. 15.4.
29. В окне списка **Templates** (Шаблоны) выделите значок **MFC Class** (Класс MFC) и нажмите кнопку **Open** (Открыть). Появится диалоговое окно **MFC Class Wizard - Extension** (Мастер создания класса MFC), изображенное на рис. 15.7.

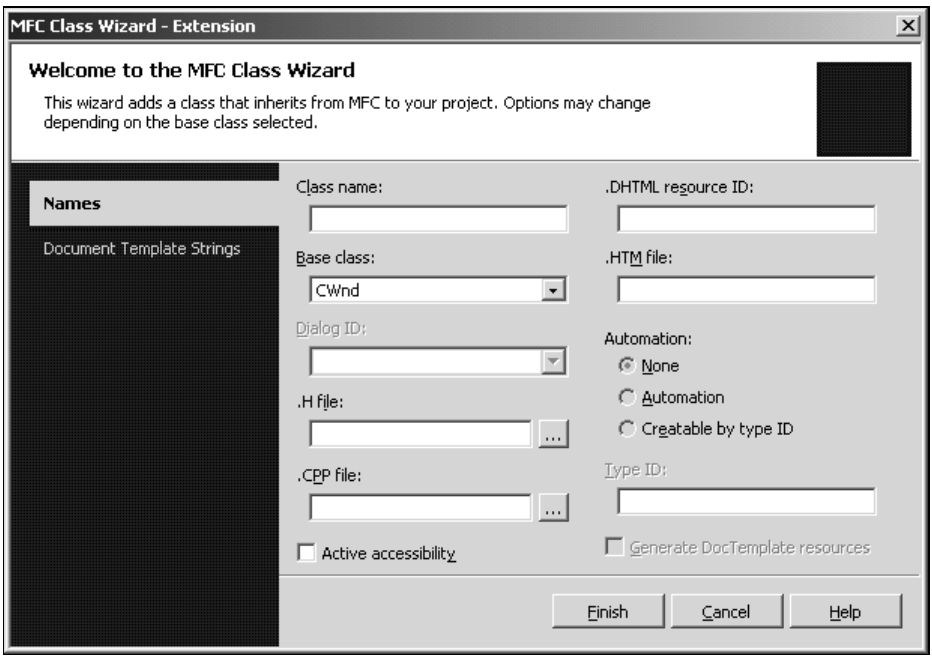


Рис. 15.7. Диалоговое окно **MFC Class Wizard - Extension**

30. Введите в текстовое поле **Class name** (Имя класса) имя нового класса `CMedianDlg`, выделите в раскрывающемся списке **Base class** (Базовый класс) имя базового класса `CDialog`, выберите в ставшем после этого доступным раскрывающемся списке **Dialog ID** (Идентификатор ресурса диалогового окна) — `IDD_MEDIAN` и нажмите кнопку **Finish** (Готово). В библиотеку динамической компоновки будет включен класс диалогового окна.
31. Откройте окно **Class View** (Просмотр класса), раскройте в нем папку **Extension**, щелкните правой кнопкой мыши на папке **CMedianDlg** и выберите в

появившемся контекстном меню команду **Properties** (Свойства). Откроется окно **Properties** (Свойства).

32. В окне **Properties** (Свойства) нажмите кнопку **Overrides** (Перегружаемые виртуальные функции базовых классов). Раскроется список виртуальных функций базовых классов, которые могут быть перегружены в данном классе.
33. Выделите строку `OnInitDialog`. В соответствующем текстовом поле появится значок раскрывающегося списка.
34. Раскройте этот список и выделите в нем единственную строку. В класс `CMedianDlg` будет добавлена соответствующая функция.
35. В окне **Properties** (Свойства) нажмите кнопку **Events** (События). Раскроется список событий, обрабатываемых данным диалоговым окном.
36. Раскройте в этом списке папку **IDC_ADD**, выделите в нем строку `BN_CLICKED`, раскройте расположенный в ней список и выделите в нем единственную строку. В класс `CMedianDlg` будет добавлена функция обработки данного сообщения.
37. В окне **Class View** (Просмотр классов) щелкните правой кнопкой мыши на папке **CMedianDlg** и в появившемся контекстном меню выберите команду **Add | Add Variable** (Добавить | Добавить переменную). Появится диалоговое окно **Add Member Variable Wizard - Extension** (Мастер добавления переменной в класс), изображенное на рис. 15.8.

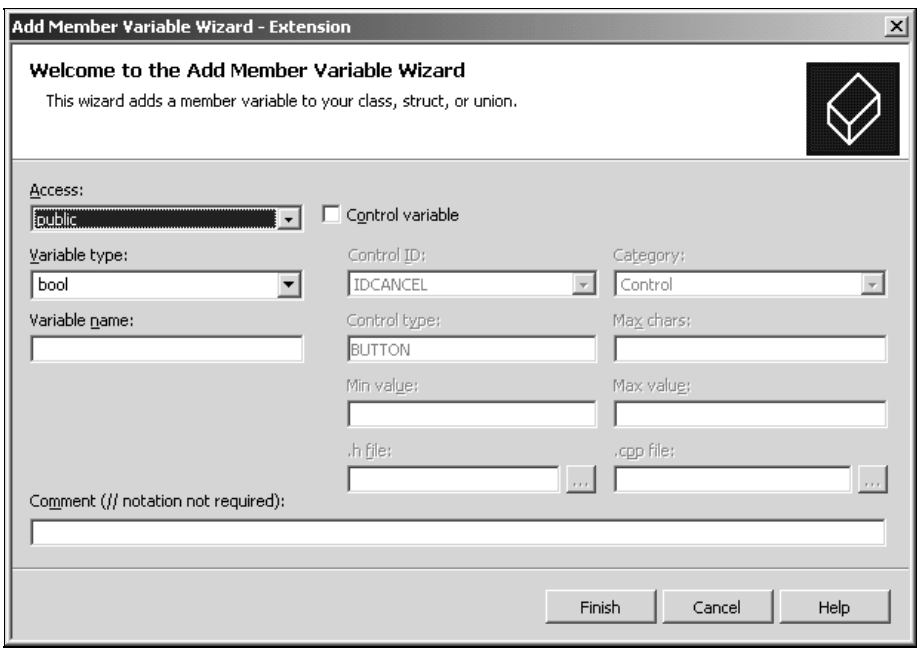


Рис. 15.8. Диалоговое окно **Add Member Variable Wizard - Extension**

38. Установите флажок **Control variable** (Связь с элементом управления), выделите в ставшем после этого доступным раскрывающемся списке **Control ID** (Идентификатор элемента управления) идентификатор элемента управления `IDC_ARRAY`, введите в текстовое поле **Variable name** (Имя переменной) идентификатор `m_Array` и нажмите кнопку **Finish** (Готово). В класс `CMedianDlg` будет добавлена новая переменная.
39. Повторите п.п. 37 и 38 для сопоставления идентификатору ресурса `IDC_RECURSIVE` переменной `m_Recursive`.
40. Повторите п.п. 37 и 38 для сопоставления идентификатору ресурса `IDC_SIMPLE` переменной `m_Simple`.
41. Повторите п.п. 37 и 38 для сопоставления идентификатору ресурса `IDC_APERTURE` переменной `m_Aperture`.
42. Повторите п.п. 37 и 38 для сопоставления идентификатору ресурса `IDC_APERTURE` переменной `m_nAperture`, но перед тем, как нажать кнопку **Finish** (Готово), выделите в раскрывающемся списке **Category** (Категория) строку **Value** (Переменная) и выделите в раскрывающемся списке **Variable type** (Тип переменной) — `int`.
43. Повторите п. 42 для сопоставления идентификатору ресурса `IDC_VALUE` переменной `m_Value`.
44. В окне **Class View** (Просмотр класса) щелкните правой кнопкой мыши на папке **CMedianDlg** и в появившемся контекстном меню выберите команду **Add | Add Variable** (Добавить | Добавить переменную). Появится диалоговое окно **Add Member Variable Wizard - Extension** (Мастер добавления переменной в класс).
45. Введите в текстовое поле раскрывающегося списка **Variable type** (Тип переменной) тип переменной `Median<int>*`, введите в текстовое поле **Variable name** (Имя переменной) идентификатор `m_Median` и нажмите кнопку **Finish** (Готово). В класс `CMedianDlg` будет добавлена новая переменная.
46. Повторите п.п. 44 и 45 для создания переменной `m_lpArray` типа `int*`.
47. Повторите п.п. 44 и 45 для создания переменной `Aperture` типа `int`.
48. Дважды щелкните левой кнопкой мыши на папке **CMedianDlg**. Откроется окно редактирования файла MedianDlg.h.
49. После строки `#pragma once` вставьте строки

```
#include  <Median.h>
#include  "Resource.h"
```

50. Откройте окно редактирования файла MedianDlg.cpp.
51. Измените конструктор класса в соответствии с текстом листинга 15.4.

Листинг 15.4. Конструктор и деструктор класса `CMedianDlg`

```
// Класс диалогового окна CMedianDlg

/////////////////////////////////////////////////////////////////////////////
// Конструкторы и деструкторы
/////////////////////////////////////////////////////////////////////////////


IMPLEMENT_DYNAMIC(CMedianDlg, CDialog)
CMedianDlg::CMedianDlg(CWnd* pParent /*=NULL*/)
  : CDialog(CMedianDlg::IDD, pParent)
  , m_lpArray( new int[3])
  , Aperture(3)
{
  m_Value    = 0;
  m_nAperture  = Aperture;
  m_Median  = new Median<int>( Aperture );
}

CMedianDlg::~CMedianDlg()
{
  delete  m_lpArray;
  delete  m_Median;
}
```

52. Измените текст функций `OnInitDialog` и `OnBnClickedAdd` в соответствии с текстом листинга 15.5.

Листинг 15.5. Функции `OnInitDialog` и `OnBnClickedAdd`

```
// Функции обработки сообщений класса CMedianDlg

// Инициализация диалогового окна
BOOL CMedianDlg::OnInitDialog(void)
{
  CDialog::OnInitDialog();

  // Инициализация текстовых полей Рекурсивная и Обычная
  SetDlgItemText( IDC_RECURSIVE, "0" );
  SetDlgItemText( IDC_SIMPLE, "0" );
```

```
  // Подготовка текста в поле Массив
  CString  szText  = "0\r\n";
  m_lpArray[0]  = 0;

  for(int i=1; i < Aperture; i++)
  {
    m_lpArray[i]  = 0;
    szText += "0\r\n";
  }

  // Вывод текста в поле Массив
  SetDlgItemText( IDC_ARRAY, szText );

  return TRUE;  // Значение TRUE возвращается в том случае,
                // если фокус ввода не передан элементу управления
                // Исключение: Вкладки OCX должны
                // возвращать значение FALSE
}

// Добавление нового значения
void CMedianDlg::OnBnClickedAdd(void)
{
  CString  szText;
  int    i;

  UpdateData();

  if( m_nAperture != Aperture )  // Апертура изменилась
  {
    // Инициализация текстовых полей и массивов
    m_nAperture   | = 1;
    m_Value    = 0;

    UpdateData( FALSE );

    SetDlgItemText( IDC_RECURSIVE, "0" );
    SetDlgItemText( IDC_SIMPLE, "0" );

    Aperture  = m_nAperture;
```

```
  delete  m_lpArray;
  delete  m_Median;

  m_lpArray  = new int[Aperture];
  m_Median  = new Median<int>( Aperture );

  szText      = "0\r\n";
  m_lpArray[0]  = 0;

  for(i=1; i < Aperture; i++)
  {
    m_lpArray[i]  = 0;
    szText      += "0\r\n";
  }

  SetDlgItemText( IDC_ARRAY, szText );
}
else
{
  // Вычисление новых значений медиан
  int*  lpTemp  = new int[Aperture];

  for(i=1; i < Aperture; i++)
  {
    m_lpArray[i-1]  = m_lpArray[i];
    lpTemp[i-1]    = m_lpArray[i];
  }

  m_lpArray[Aperture-1]  = m_Value;
  lpTemp[Aperture-1]    = m_Value;

  Sort( lpTemp, Aperture );

  // Вывод полученных значений
  szText.Format("%d", m_Median-> Get( m_Value));
  SetDlgItemText( IDC_RECURSIVE,  szText );
  szText.Format("%d", lpTemp[ Aperture >> 1 ]);
  SetDlgItemText( IDC_SIMPLE, szText );

  // Модификация выведенного массива
  m_Array.SetSel( 0, m_Array.LineLength(0) + 2 );
```

```
    m_Array.ReplaceSel( "" );
    m_Array.SetSel( 10000000, 10000000);
    szText.Format("%d\r\n", m_Value);
    m_Array.ReplaceSel( szText );

    delete  lpTemp;
  }
}
```

53. В начале файла уничтожьте строку `#include "Extension.h"`
54. Откройте окно **Class View** (Просмотр класса), щелкните правой кнопкой мыши на папке **Extension** (Расширение) и выберите в появившемся контекстном меню команду **Properties** (Свойства). Появится диалоговое окно **Extension Property Pages** (Вкладки свойств).
55. В раскрывающемся списке **Configuration** (Конфигурация) выделите конфигурацию **All Configurations** (Все конфигурации), а в окне иерархического списка раскройте папку **C/C++**. Диалоговое окно **Extension Property Pages** (Вкладки свойств) примет вид, изображенный на рис. 15.9.

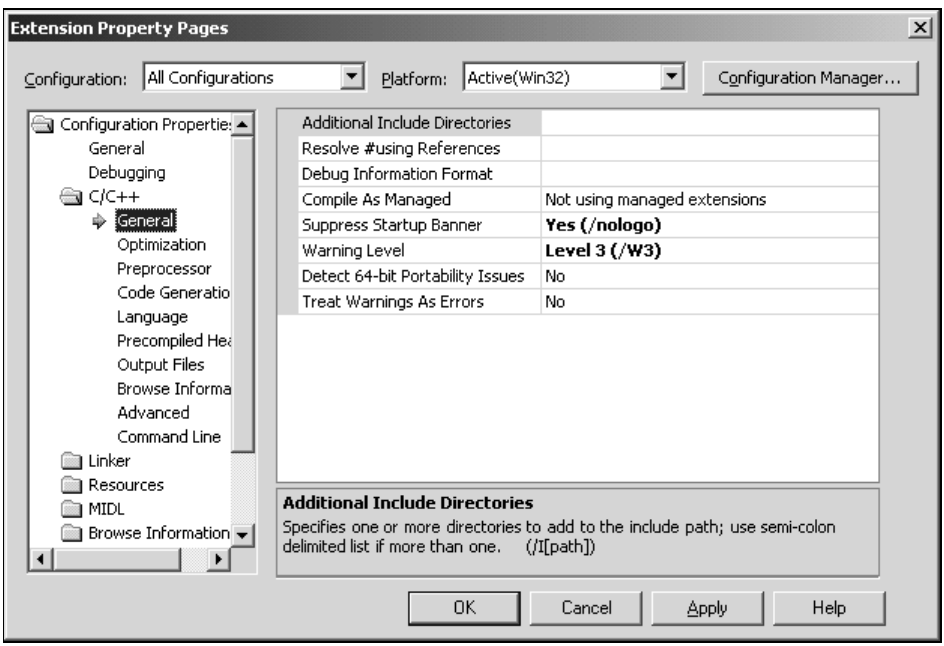


Рис. 15.9. Диалоговое окно **Extension Property Pages**, папка **C/C++**

56. В окне списка выделите строку **Additional Include Directories** (Дополнительные каталоги включаемых файлов) и введите в ее текстовое поле полный путь к проекту библиотеки динамической компоновки Library (в моем случае "E:\Documents and Settings\Nick\My Documents\Visual Studio Projects\Library").
57. В окне иерархического списка раскройте папку **Linker** (Редактор связей). Диалоговое окно **Extension Property Pages** (Вкладки свойств) примет вид, изображенный на рис. 15.10.

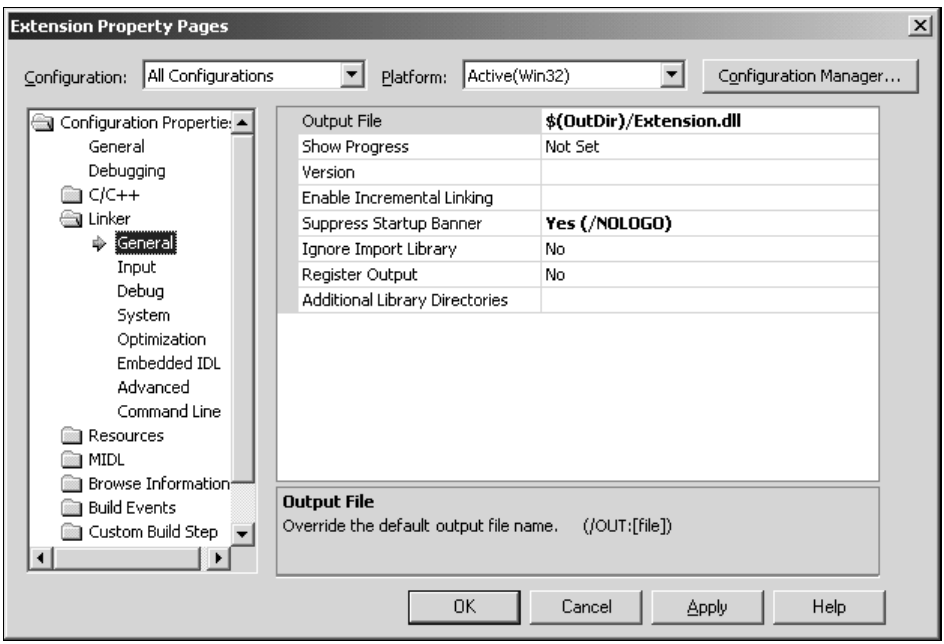


Рис. 15.10. Диалоговое окно **Extension Property Pages**, папка **Linker**

58. В окне списка выделите строку **Additional Library Directories** (Дополнительные каталоги библиотек) и введите в ее текстовое поле полный путь к библиотеке динамической компоновки Library (в моем случае "E:\Documents and Settings\Nick\My Documents\Visual Studio Projects\Library\Debug").
59. В папке **Linker** (Редактор связей) окна иерархического списка диалогового окна **Extension Property Pages** (Вкладки свойств) выделите строку **Input** (Окно ввода). Диалоговое окно **Extension Property Pages** (Вкладки свойств) примет вид, изображенный на рис. 15.11.
60. В окне списка выделите строку **Additional Dependencies** (Дополнительные входные файлы) и введите в текстовое поле ее раскрывающегося списка имя библиотеки Library.lib.

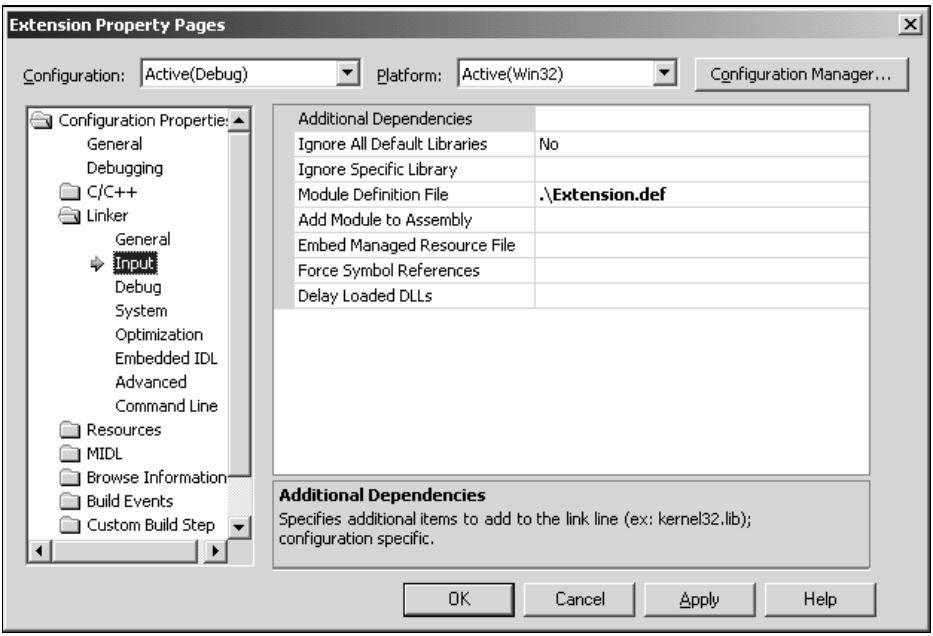


Рис. 15.11. Диалоговое окно **Extension Property Pages**

61. Нажмите кнопку **OK**. Диалоговое окно **Extension Property Pages** (Вкладки свойств) закроется.
62. Выберите команду меню **Build | Build Solution** (Создать | Компоновать решение) или нажмите комбинацию клавиш <Ctrl>+<Shift>+<B>. Библиотека динамической компоновки будет скомпилирована.
63. Закройте приложение Microsoft Visual C++, сохранив все внесенные в проект изменения.

Для того чтобы созданной библиотекой можно было бы пользоваться в других приложениях, необходимо указать к ней путь. К сожалению, этого нельзя сделать средствами среды программирования Visual Studio.NET. Как уже говорилось выше, при поиске библиотеки динамической компоновки приложение сначала просматривает папку с исполняемым файлом приложения, затем текущую папку процесса, после этого — папку операционной системы и, наконец, папку, указанную в системной переменной `PATH`. Эта последовательность поиска логична для распространяемой версии приложения, но создает существенные неудобства при параллельной разработке приложения и библиотеки динамической компоновки (то есть при внесении в эту библиотеку изменений, диктуемых создаваемым приложением). Как видно из данной последовательности поиска, у пользователя имеется только одна альтернатива размещения файла библиотеки динамической компоновки: в папке операционной системы или в папке, указанной в системной переменной `PATH`, поскольку эта папка должна быть неизменной, а все остальные папки зависят от компилируемого приложения.

В данном случае библиотеки динамической компоновки компилировались в свои каталоги, поэтому единственным оставшимся выходом является указание этих каталогов в системной переменной `PATH`. Для этого достаточно отредактировать файл autoexec.bat в любом текстовом редакторе. Данный скрытый файл располагается в корневом каталоге диска C:. Однако, уважая разработчиков операционной системы, сделавших этот файл скрытым, рассмотрим более цивилизованные способы добавления информации в данную системную переменную. В Windows 98 и в Windows 2000 эти процедуры происходят по-разному.

Чтобы добавить в системную переменную `PATH` новые пути при работе под управлением операционной системы Windows 2000:

1. На рабочем столе щелкните правой кнопкой мыши на значке **My Computer** (Мой компьютер) и выберите в появившемся контекстном меню команду **Properties** (Свойства). Появится диалоговое окно **System Properties** (Свойства системы), изображенное на рис. 15.12.

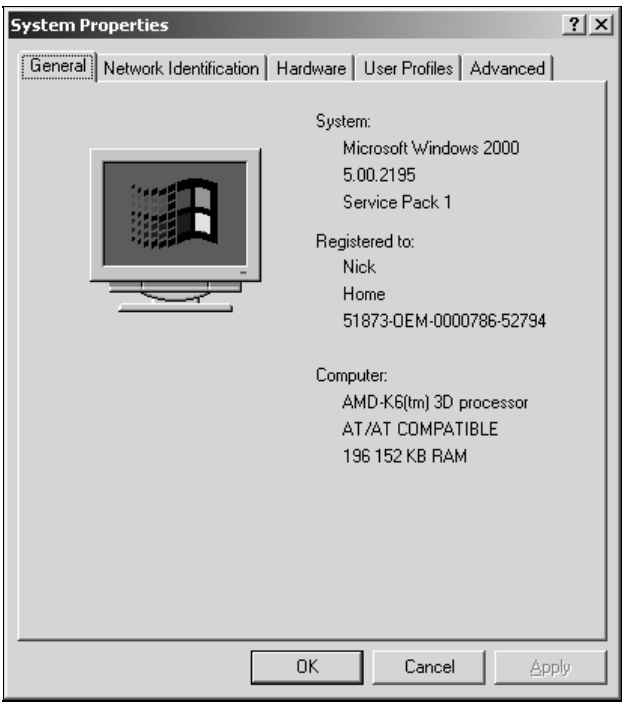


Рис. 15.12. Диалоговое окно **System Properties**

2. Раскройте вкладку **Advanced** (Дополнительно). Диалоговое окно **System Properties** (Свойства системы) примет вид, изображенный на рис. 15.13.

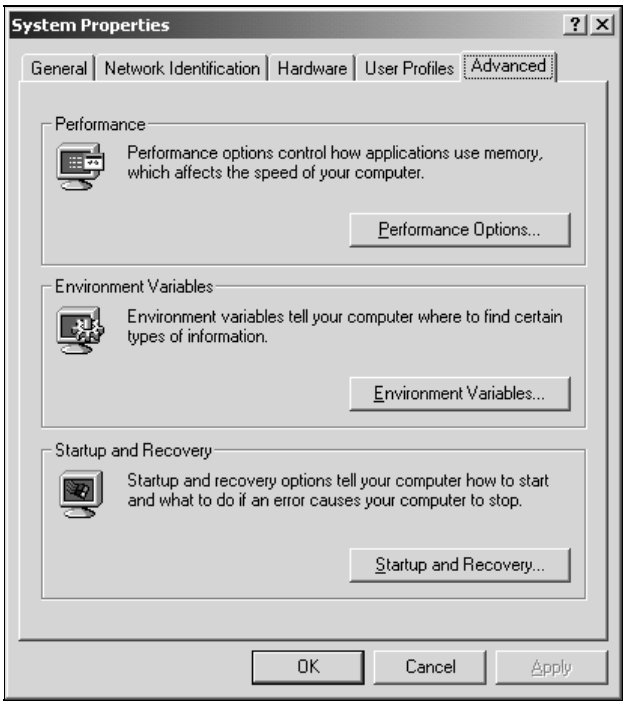


Рис. 15.13. Диалоговое окно **System Properties**, вкладка **Advanced**

3. Нажмите кнопку **Environment Variables** (Переменные среды). Появится диалоговое окно **Environment Variables** (Переменные среды), изображенное на рис. 15.14.
4. В окне списка **System variables** (Системные переменные) выделите строку **Path** (Путь) и нажмите кнопку **Edit** (Изменить). Появится диалоговое окно **Edit System Variable** (Изменение системной переменной), изображенное на рис. 15.15.
5. Добавьте в конец текста в поле **Variable Value** (Значение переменной) путь к библиотекам динамической компоновки Extension и Library (в моем случае "E:\Documents and Settings\Nick\My Documents\Visual Studio Projects\Extension\Debug;E:\Documents and Settings\Nick\My Documents\Visual Studio Projects\Library\Debug") и нажмите кнопку **OK**.
6. Нажмите кнопку **OK** в диалоговом окне **System Properties** (Свойства системы). Это диалоговое окно закроется.
7. Перезагрузите Windows, чтобы внесенные изменения вступили в силу (иногда изменения вступают в силу и без перезагрузки операционной системы).

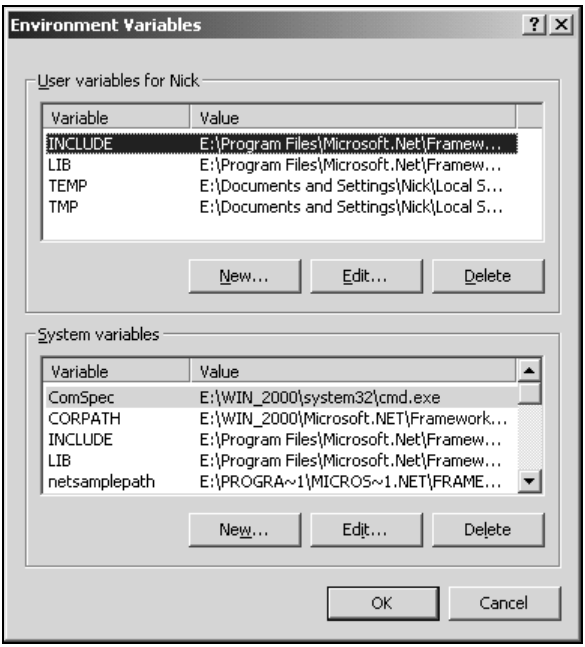


Рис. 15.14. Диалоговое окно **Environment Variables**

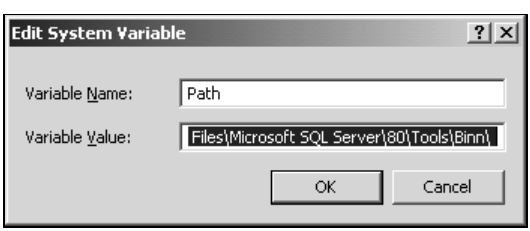


Рис. 15.15. Диалоговое окно **Edit System Variable**

Чтобы добавить в системную переменную `PATH` новые пути при работе под управлением операционной системы Windows 98:

1. В Панели задач выберите команду **Пуск | Выполнить**. Появится диалоговое окно **Запуск программы**, изображенное на рис. 15.16.
2. Введите в текстовое поле имя приложения msconfig и нажмите кнопку **OK**. Появится диалоговое окно **Программа настройки системы**, изображенное на рис. 15.17.
3. Раскройте вкладку **Autoexec.bat**. Диалоговое окно **Программа настройки системы** примет вид, изображенный на рис. 15.18.

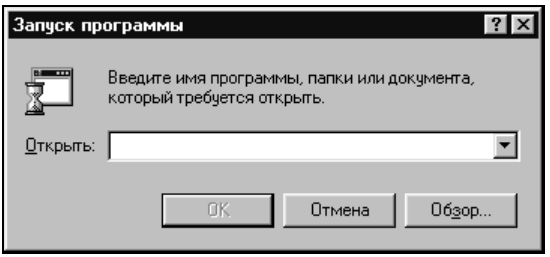


Рис. 15.16. Диалоговое окно **Запуск программы**

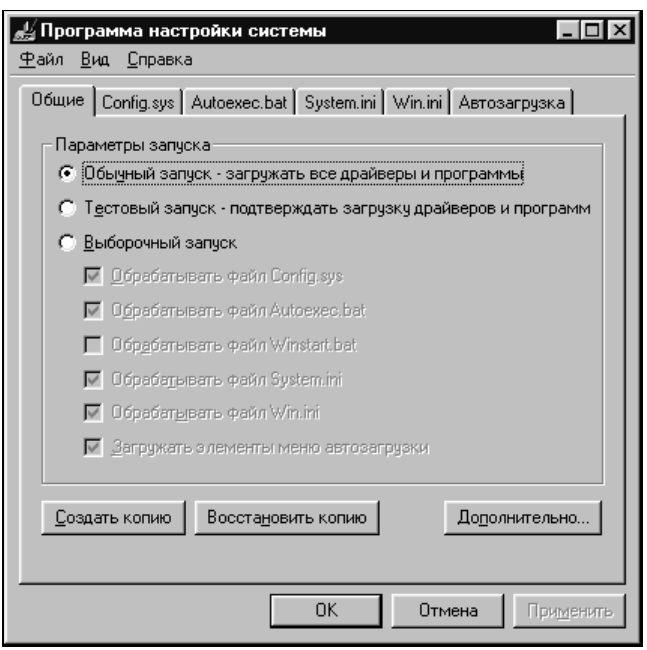


Рис. 15.17. Диалоговое окно
Программа настройки системы

4. В окне списка выделите строку **PATH C:WIN_98\COMMAND;C:\DRIVERS** и нажмите кнопку **Изменить**. Соответствующая строка будет выделена пунктирной рамкой и в ней появится текстовый курсор.
5. Добавьте в конец данной строки путь к библиотекам динамической компоновки Extension и Library (в моем случае "E:\Documents and Settings\Nick\My Documents\Visual Studio Projects\Extension\Debug;E:\Documents and Settings\Nick\My Documents\Visual Studio Projects\Library\Debug") и нажмите кнопку **OK**. Диалоговое окно **Программа настройки системы** закроется.

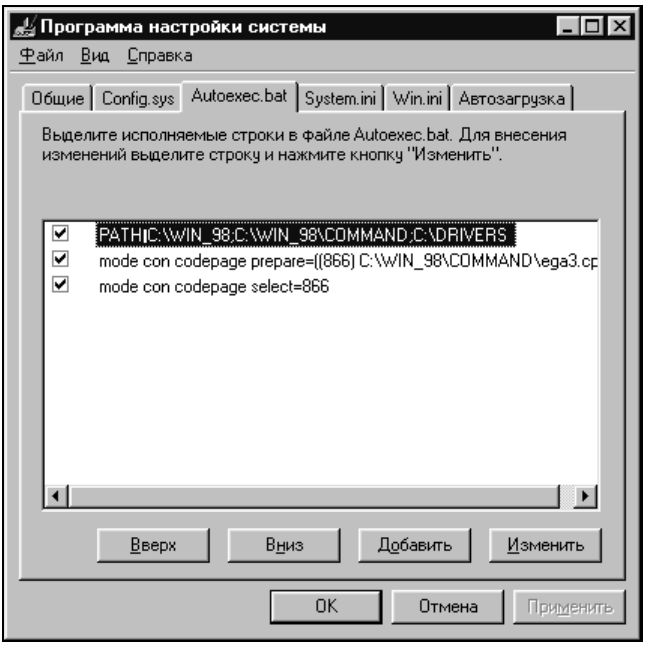


Рис. 15.18. Вкладка **Autoexec.bat**

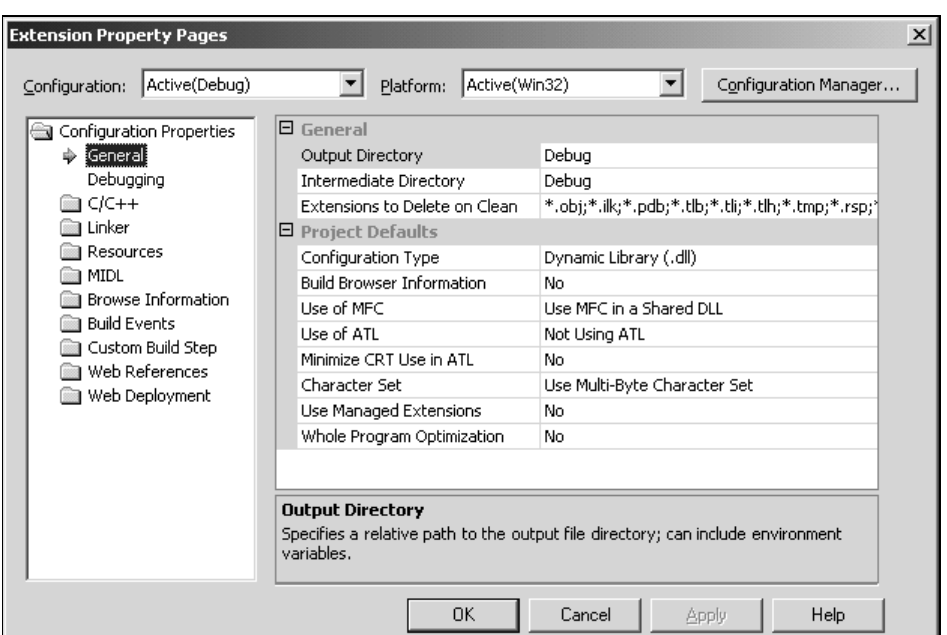


Рис. 15.19. Диалоговое окно **Extension Property Pages**

Хотя в приведенном примере путь к библиотеке динамической компоновки помещался в системную переменную `PATH`, опишем действия, необходимые для реализации другого подхода.

Чтобы поместить выходные файлы библиотеки динамической компоновки в каталог операционной системы:

1. Откройте окно **Class View** (Просмотр классов), щелкните правой кнопкой мыши на папке **Extension** (Расширение) и выберите в появившемся контекстном меню команду **Properties** (Свойства). Появится диалоговое окно **Extension Property Pages** (Вкладки свойств).
2. В раскрывающемся списке **Configuration** (Конфигурация) выделите конфигурацию **All Configurations** (Все конфигурации). Диалоговое окно **Extension Property Pages** (Вкладки свойств) примет вид, изображенный на рис. 15.19.
3. Введите в текстовое поле **Output Directory** (Каталог вывода) путь в системный каталог (в моем случае E:\Win_2000) и нажмите кнопку **OK**.

Создание демонстрационного приложения

Чтобы продемонстрировать использование библиотек динамической компоновки, было создано приложение DemoLib, текст которого можно найти в одноименном каталоге на дискете, прилагаемой к данной книге. Чтобы самостоятельно создать это приложение:

1. По методике, описанной в *главе 1*, создайте диалоговое приложение с именем DemoLib.
2. Откройте окно **Resource View** (Просмотр ресурсов), раскройте в нем папку **DemoLib.rc**, а в ней — папку **Dialog** (Диалог).
3. Выделите значок `IDD_ABOUTBOX` и нажмите клавишу <Del>. Данный ресурс будет удален из приложения.
4. Повторите п. 3 для идентификатора диалогового окна `IDD_DEMOLIB_DIALOG`.
5. Раскройте папку **String Table** (Таблица строковых ресурсов), выделите в ней значок **String Table** и нажмите клавишу <Del>. Данный ресурс будет удален из приложения.
6. Откройте окно **Solution Explorer** (Проводник решения), раскройте в нем папку **DemoLib**, а в ней — папку **Source Files**.
7. Выделите имя файла DemoLibDlg.cpp и нажмите клавишу <Del>. Соответствующий файл будет удален из приложения.
8. Раскройте папку **Header Files** (Файлы заголовков) и повторите п. 6 для файла DemoLibDlg.h.
9. Выберите команду меню **File | Open | File** (Файл | Открыть файл), нажмите комбинацию клавиш <Ctrl>+<O> или кнопку **Open** (Открыть) на панели инструментов **Standard** (Стандартная). Появится диалоговое окно **Open File** (Открыть файл), изображенное на рис. 15.20.

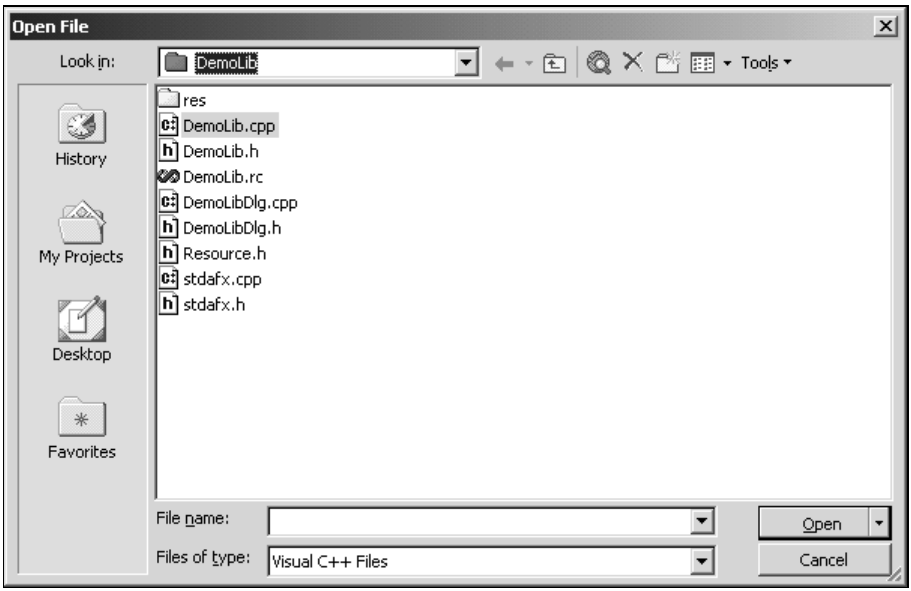


Рис. 15.20. Диалоговое окно **Open File**

10. В окне списка выделите имена файлов DemoLibDlg.cpp и DemoLibDlg.h (для одновременного выделения нескольких файлов удерживайте нажатой клавиши <Shift> или <Ctrl>) и щелкните правой кнопкой мыши на выделении.
11. В появившемся контекстном меню выберите команду **Delete** (Удалить). Появится окно сообщения **Confirm Multiple File Delete** (Подтверждение уничтожения нескольких файлов), изображенное на рис. 15.21.

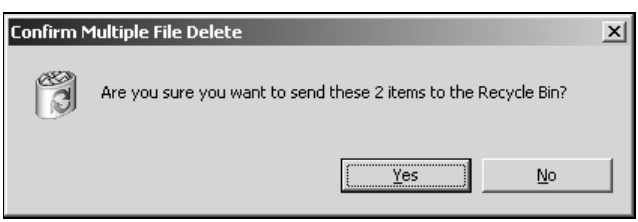


Рис. 15.21. Диалоговое окно **Confirm Multiple File Delete**

12. Нажмите кнопку **Yes** (Да), подтвердив тем самым свое согласие на помещение этих файлов в корзину. Отмеченные файлы исчезнут из окна списка.
13. Нажмите кнопку **Cancel** (Отмена) в диалоговом окне **Open File** (Открыть файл). Диалоговое окно закроется.

14. В окне **Solution Explorer** (Проводник решения) щелкните правой кнопкой мыши на папке **DemoLib** и выберите в появившемся контекстном меню команду **Properties** (Свойства). Появится диалоговое окно **Property Pages** (Вкладки свойств).
15. В раскрывающемся списке **Configuration** (Конфигурация) выделите конфигурацию **All Configurations** (Все конфигурации), а в окне иерархического списка раскройте папку **C/C++**.
16. В окне списка выделите строку **Additional Include Directories** (Дополнительные включаемые каталоги) и введите в ее текстовое поле полные пути к проектам библиотек динамической компоновки Library и Extension (в моем случае "E:\Documents and Settings\Nick\My Documents\Visual Studio Projects\Extension;E:\Documents and Settings\Nick\My Documents\Visual Studio Projects\Library").
17. В окне иерархического списка раскройте папку **Linker** (Редактор связей).
18. В окне списка выделите строку **Additional Library Directories** (Дополнительные каталоги включаемых файлов) и введите в ее текстовое поле полный путь к библиотеке динамической компоновки Extension (в моем случае "E:\Documents and Settings\Nick\My Documents\Visual Studio Projects\Extension\Debug").
19. В папке **Linker** (Редактор связей) окна иерархического списка диалогового окна **Property Pages** (Вкладки свойств) выделите строку **Input** (Окно ввода).
20. В окне списка выделите строку **Additional Dependencies** (Дополнительные входные файлы) и введите в текстовое поле ее раскрывающегося списка имя библиотеки Extension.lib.
21. Нажмите кнопку **OK**. Диалоговое окно **Property Pages** (Вкладки свойств) закроется.
22. В папке **Source Files** (Файлы реализации) окна **Solution Explorer** (Проводник решения) дважды щелкните левой кнопкой мыши на имени файла DemoLib.cpp. Откроется окно редактирования этого файла.
23. В начале файла замените строку `#include "DemoLibDlg.h"` строкой

    `#include "MedianDlg.h"`
24. Измените функцию `InitInstance` в соответствии с текстом листинга 15.6.

Листинг 15.6. Функция `InitInstance`

```
// Инициализация класса диалогового окна CDemoLibApp

BOOL CDemoLibApp::InitInstance()
{
  // Вызов метода базового класса
  CWinApp::InitInstance();
```

```
  AfxEnableControlContainer();

  // Создание и вызов диалогового окна
  CMedianDlg dlg;
  m_pMainWnd = &dlg;
  int nResponse = dlg.DoModal();

  // Обработка кода завершения
  if (nResponse == IDOK)
  {
    // Была нажата кнопка OK
  }
  else if (nResponse == IDCANCEL)
  {
    // Была нажата кнопка Отмена
  }

  // Поскольку диалоговое окно уже закрыто, возвращается значение FALSE,
  // приводящее к выходу из приложения,
  // а не к запуску цикла обработки сообщений
  return FALSE;
}
```

25. Нажмите клавишу <F5> и запустите приложение на исполнение (предполагается, что в приложении установлен режим загрузки последнего загруженного приложения). Появится диалоговое окно **Медиана**, изображенное на рис. 15.22.
26. Введите в текстовое поле **Новое** новое входное значение и нажмите кнопку **Добавить**. Это значение появится в нижней части окна списка **Массив**. Чтобы сохранить размер массива, из него будет удалено его верхнее значение. В текстовом поле **Рекурсивная** будет выведено значение рекурсивной медианы входного вектора, а в текстовом поле **Обычная** будет выведено значение обычной медианы.
27. Несколько раз повторите п. 25 для изучения свойств обычной и рекурсивной медианы.
28. Введите новое значение в текстовое поле **Апертура**. Размерность массива в окне списка изменится, а все его значения обнулятся. Соответствующим образом изменятся и значения в текстовых полях **Рекурсивная** и **Обычная**. Обратите внимание на то, что при вводе в текстовое поле **Апертура** четного значения оно увеличивается на единицу, чтобы стать нечетным.
29. Повторите п.п. 25—27 несколько раз.
30. Нажмите кнопку **Отмена** или **OK** и закройте диалоговое окно **Медиана**.

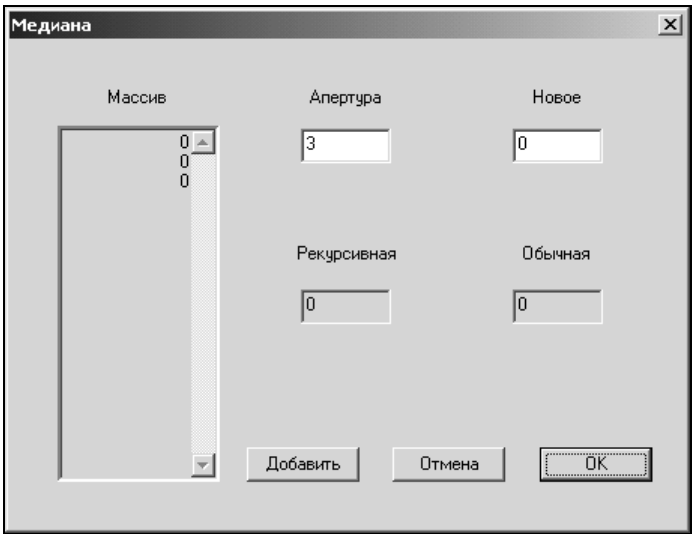


Рис. 15.22. Диалоговое окно **Медиана**

Описание созданных проектов

Данный пример использования библиотек динамической компоновки состоит из трех взаимосвязанных приложений, поэтому их описание сведено в одно место.

В приложении Library создается регулярная библиотека динамической компоновки. Для простоты она содержит одну функцию и один класс. Как в функции, так и в классе использованы шаблоны, что позволяет применять их для работы с различными типами данных. Для демонстрации различных подходов к экспорту функций и классов целочисленные версии функций данной библиотеки экспортируются с использованием файла определения модуля, а версии для чисел с плавающей запятой используют директивы `__declspec(dllexport)` и `__declspec(dllimport)`.

Для выбора соответствующей директивы в файле заголовка Median.h используется оператор условной трансляции, различным образом определяющий переменную `__DEMO_LIB`. Аргументом этого оператора условной трансляции является переменная `__MY_LIBRARY__`, определяемая в файле Median.cpp. Поскольку этот файл используется только при компиляции библиотеки динамической компоновки, то данная переменная будет определена только в этом случае, что позволяет разделять вызов файла заголовка при компиляции и использовании библиотеки динамической компоновки.

Чтобы определение переменной `__MY_LIBRARY__` оказало действие на оператор условной трансляции в файле заголовка, оно должно быть произведено до включения этого файла в файл реализации, однако для того, чтобы эта опера-

ция могла произойти, необходимо вызвать до нее файл StdAfx.h, поэтому определение этой переменной производится между включением этих файлов.

Как уже говорилось в *главе 11*, функции и классы, использующие шаблоны, будут создаваться только в том случае, если они вызываются в том же файле, где они реализованы. Другим способом указания на необходимость создания подобной функции или класса является использование директивы `template`, как это сделано в данном случае. Если соответствующая функция не экспортируется в файле определения модуля, эта директива должно включать в себя директиву `__declspec(dllexport)`, как это и сделано в данном случае. Если бы в функциях данной библиотеки динамической компоновки не применялись шаблоны, то для экспорта этих функций не было бы необходимости использовать директивы `__declspec(dllexport)` в файле реализации. Достаточно было бы использовать оператор условной трансляции в файле заголовка *(см. главу 11)*.

Поскольку в данной библиотеке используются шаблоны функций, то в файле определения модуля необходимо указать полные имена экспортируемых функций, что существенно снижает читабельность этого файла для неподготовленных пользователей. Интересно, что для работы библиотеки нужно экспортировать только функцию `Median::Get` и нет необходимости экспортировать конструктор и деструктор данного класса.

Данный пример демонстрирует работу с медианой случайной последовательности. В теории вероятности медианой распределения случайной величины называется значение, соответствующее значению 0.5 функции вероятности (интеграла от функции плотности распределения вероятности). В математической статистике медианой называют значение, занимающее центральное место после ранжировки случайной выборки. Если случайная выборка имеет четное количество членов, медианой считается полусумма центральных членов его ранжировки. Однако в большинстве случаев для простоты рассматривают только значения медианы для выборок с нечетным числом членов.

Медианой отсчета случайной последовательности называется значение медианы, определенной для окна, центр которого приходится на данный отсчет. Последовательность значений медиан отсчетов случайной выборки образуют некоторую новую последовательность. Если значения этой последовательности записывать в исходную последовательность, результат будет отличаться от того, который получен без внесения изменений в исходную последовательность. Эта новая процедура называется рекурсивной медианой, поскольку ее новое выходное значение зависит от предыдущих выходных значений.

Для проверки правильности работы функций библиотеки Library в библиотеке динамической компоновки Extension создается диалоговое окно. Процедура создания диалогового окна подробно описана в *главе 3*. Поэтому здесь мы остановимся только на вопросах взаимодействия его элементов управления.

В конструкторе класса `CMedianDlg` производится инициализация его членов. При этом некоторые операции по инициализации взяла на себя среда программирования и включила их в заголовок конструктора класса. Остальные операции по

инициализации производятся в теле данного конструктора. В деструкторе данного класса уничтожаются используемые в нем динамические переменные.

Функция `OnInitDialog` вызывается при вызове функции `DoModal` перед созданием диалогового окна. В ней нужно производить все операции по инициализации элементов управления. В ней с использованием функции `SetDlgItemText` заполняются текстовые поля **Рекурсивная**, **Обычная** и **Массив**. Поскольку текстовое поле **Массив** может содержать несколько строк, число которых определяется значением апертуры медианы, то выводимый в нем текст создается в цикле. Этот же цикл используется для заполнения массива медианы.

Функция `OnClickedAdd` является функцией обработки сообщения о нажатии пользователем кнопки **Добавить**. Если после нажатия этой кнопки выясняется, что пользователь изменил значение апертуры, то производится инициализация переменных для нового значения апертуры. Прежде всего, полученное значение апертуры принудительно делается нечетным. После этого производится инициализация, аналогичная той, которая производилась в функции `OnInitDialog`. Поскольку задание размера массива и апертуры рекурсивной медианы возможно только в процессе их создания, то старые объекты уничтожаются и вместо них создаются новые объекты с новыми параметрами.

Если же апертура медианы осталась неизменной, то содержимое массива медианы сдвигается на один отсчет, а на освободившееся место записывается введенное пользователем значение. Одновременно значение этого массива копируется во временный массив. Этот временный массив упорядочивается в функции `Sort` и значение его центрального элемента выводится в текстовое поле **Обычная**. В текстовое поле **Рекурсивная** выводится значение, возвращаемое функцией `Median::Get`. Для преобразования этих значений в текстовый вид используется функция `CString::Format`.

Для организации прокрутки значений в текстовом поле **Массив** в нем с использованием функции `CEdit::SetSel` выделяется первая строка и уничтожается функцией `CEdit::ReplaceSel`, в качестве аргумента которой указана пустая строка. Здесь необходимо обратить внимание на то, что в функции `CEdit::SetSel` выделено на два символа больше, чем длина строки, возвращенная функцией `CEdit::LineLength`. Это связано с тем, что данная функция не учитывает символы возврата каретки и перевода строки, располагающиеся в конце каждой строки. Использование функции `CEdit::ReplaceSel` связано с тем, что функции работы с буфером обмена, в том числе и функции `CEdit::Clear` и `CEdit::Cut`, отказываются работать в данном классе. Это справедливо не только для Visual C++ 7.0, но и для Visual C++ 6.0.

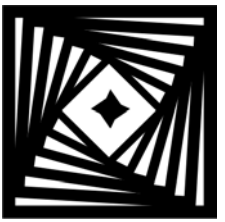
Функция `CEdit::SetSel` с запредельно большими аргументами используется для помещения текстового курсора в конец текста. После этого вызывается функция `CEdit::ReplaceSel` для вывода введенного пользователем значения.

В данном приложении экспорт функций осуществлен с использованием файла определения модуля. Интересно, что в данном случае требуется экспортировать только конструктор и деструктор класса диалогового окна и нет необходимости экспортировать функцию `DoModal` для данного класса.

Диалоговое приложение DemoLib используется для вывода диалогового окна, созданного в приложении Extension, поэтому нет никакой необходимости сохранять в нем ресурсы, связанные с его собственным диалоговым окном и файлы заголовка и реализации этого окна. Для вывода нужного диалогового окна достаточно только изменить тип создаваемого объекта класса диалогового окна в функции `CDemoLibApp::InitInstance` и заменить используемый в данном файле файл заголовка класса диалогового окна.

Основные операции, произведенные в данном приложении, связаны с подключением к нему библиотек динамической компоновки. При этом в диалоговом окне **Property Pages** (Вкладки свойств) указывается путь к используемым в данном приложении файлам заголовков и файлам библиотек, а также указывается используемая в нем библиотека. Обратите внимание на то, что необходимо указать пути ко всем файлам заголовка, даже не используемым непосредственно в данном приложении, и указать только ту библиотеку и путь к ней, которая непосредственно используется в приложении. При этом следует помнить, что в данном случае речь идет не о самой библиотеке динамической компоновки, а о ее lib-файле.

Глава 16



Создание простейшего приложения Internet

Microsoft позиционирует Visual Studio.NET, прежде всего, как средство создания многоуровневых систем в Internet. Данная книга посвящена созданию простейших приложений, и в ней, конечно же, не будет рассматриваться создание распределенных приложений, однако нельзя полностью обойти вопрос создания Internet-приложений.

В свое время корпорация Microsoft не обратила должного внимания на Internet и ей пришлось достаточно агрессивно отвоевывать себе место на этом рынке. Памятуя об этом, Microsoft теперь старается внедрить средства для работы с Internet во все разрабатываемые ею приложения. Конечно же, при этом она не обошла вниманием разрабатываемую визуальную среду программирования. Для работы с Internet Microsoft создала библиотеку WinInet, являющуюся подмножеством библиотеки MFC.

Классы WinInet

Классы WinInet позволяют разработчику использовать для создания Internet-приложения вызовы функций высокого уровня, выполняющие бóльшую часть работы. Таким образом, процесс создания Internet-приложения практически ничем не отличается от процесса создания любого другого приложения MFC. Классы WinInet позволяют разработчику выбрать тот уровень абстракции, который необходим для его приложения: он может воспользоваться небольшим набором стандартных функций или произвести низкоуровневую настройку соединения. Основные классы WinInet приведены в табл. 16.1.

Таблица 16.1. *Основные классы WinInet*

Класс	Описание
`CInternetSession`	Создает сеанс Internet и управляет им. Все приложения, использующие WinInet перед тем, как создать объект любого другого класса WinInet, должны создать объект класса `CInternetSession`
`CInternetConnection`	Создает соединение Internet и управляет им. Этот класс является базовым для классов `CFtpConnection`, `CGopherConnection` и `CHttpConnection`

Таблица 16.1 *(окончание)*

Класс	Описание
`CFtpConnection`	Создает соединение FTP и управляет им
`CGopherConnection`	Создает соединение gopher и управляет им
`CHttpConnection`	Создает соединение HTTP и управляет им
`CInternetFile`	Разрешает удаленный доступ к файлам на сервере Internet. Этот класс является базовым для классов `CGopherFile` и `CHttpFile`
`CGopherFile`	Управляет обменом информации с файлом на сервере gopher
`CHttpFile`	Управляет обменом информации с файлом на сервере HTTP
`CFileFind`	Позволяет программе производить поиск файлов. Этот класс является базовым для классов `CFtpFileFind` и `CGopherFileFind`
`CFtpFileFind`	Позволяет программе производить поиск файлов на сервере FTP
`CGopherFileFind`	Позволяет программе производить поиск файлов на сервере gopher
`CGopherLocator`	Позволяет получить локатор с сервера gopher
`CInternetException`	Обрабатывает исключения, вызываемые другими классами WinInet

Чтение Web-страницы

В данной главе будет рассмотрено приложение WebRead, использующее классы библиотеки WinInet для создания приложения, позволяющего считывать содержимое Web-страницы в формате HTML. Текст этого приложения расположен в одноименной папке на дискете, прилагаемой к данной книге. Чтобы самостоятельно создать это приложение:

1. Выберите команду меню **File | New | Project** (Файл | Создать | Проект) или нажмите комбинацию клавиш <Ctrl>+<Shift>+<N>. Появится диалоговое окно **New Project** (Новый проект).
2. В окне списка **Templates** (Шаблоны) выделите значок **MFC Application** (Приложение MFC), в текстовое поле **Name** (Имя) введите имя приложения WebRead и нажмите кнопку **OK**. Появится диалоговое окно **MFC Application Wizard** (Мастер создания приложений MFC).

3. Раскройте вкладку **Application Type** (Тип приложения). Установите в ней переключатель **Select application type** (Выбрать тип приложения) в положение **Single document** (Однооконное приложение).
4. Раскройте вкладку **Generated Classes** (Создаваемые классы), выделите в окне списка **Generated classes** (Создаваемые классы) имя класса `CWebReadView`, раскройте список **Base class** (Базовый класс) и выделите в нем имя базового класса `CEditView`.
5. Нажмите кнопку **Finish** (Готово). Мастер MFC Application Wizard создаст заготовку нового приложения.
6. Откройте окно **Resource View** (Просмотр ресурсов) и раскройте в нем папку **WebRead.rc**.
7. Щелкните правой кнопкой мыши на папке **Dialog** (Диалог) и выберите в появившемся контекстном меню команду **Insert Dialog** (Вставить диалоговое окно). Откроется окно редактирования ресурса, в которое будет помещена заготовка нового диалогового окна.
8. Щелкните правой кнопкой мыши на заготовке диалогового окна и выберите в появившемся контекстном меню команду **Properties** (Свойства). Откроется окно **Properties** (Свойства).
9. В окне **Properties** (Свойства) выделите строку **ID** (Идентификатор ресурса) и введите в текстовое поле связанного с ней раскрывающегося списка идентификатор ресурса `IDD_URLDLG`.
10. В окне **Properties** (Свойства) выделите строку **Caption** (Заголовок) и введите в текстовое поле связанного с ней раскрывающегося списка строку заголовка диалогового окна "Connect".
11. В окне **Toolbox** (Инструментарий) выберите элемент управления **Static Text** (Статический текст) и поместите его в левый верхний угол диалогового окна.
12. В окне **Properties** (Свойства) выделите строку **Caption** (Заголовок) и введите в текстовое поле связанного с ней раскрывающегося списка строку "Enter URL:".
13. В окне **Toolbox** (Инструментарий) выберите элемент управления **Edit Control** (Текстовое поле) и поместите его под статическим текстом.
14. В окне **Properties** (Свойства) выделите строку **ID** (Идентификатор ресурса) и введите в текстовое поле связанного с ней раскрывающегося списка идентификатор ресурса `IDC_URL`.
15. Растяните текстовое поле по горизонтали, чтобы оно заняло все свободное место.
16. Переместите нижнюю границу диалогового окна таким образом, чтобы между нижней границей текстового поля и нижней границей диалогового окна не осталось свободного места. В результате этих операций заготовка диалогового окна примет вид, изображенный на рис. 16.1.

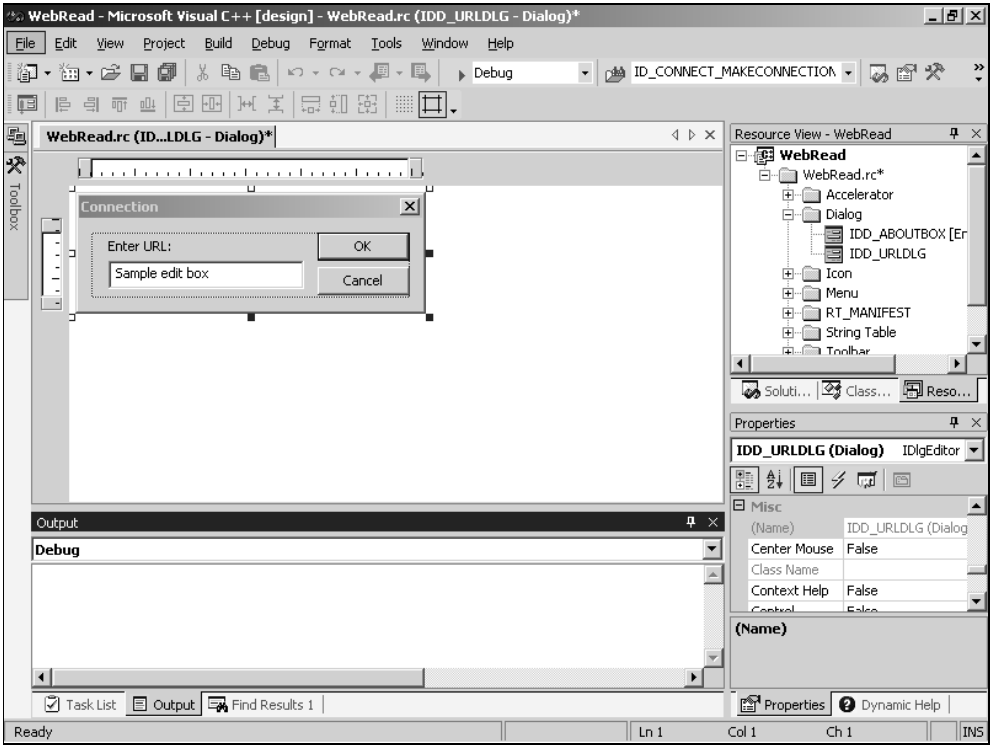


Рис. 16.1. Окно редактирования ресурса диалогового окна

17. В окне **Resource View** (Просмотр ресурсов) раскройте папку **Menu** (Меню) и дважды щелкните левой кнопкой мыши на идентификаторе ресурса `IDR_MAINFRAME`. Откроется окно редактирования ресурса меню.
18. Щелкните левой кнопкой мыши на белом квадратике с надписью "Type Here", расположенном в строке меню, и введите в него текст "&Connect". При этом новое меню раскроется.
19. Щелкните левой кнопкой мыши на белом квадратике с надписью "Type Here", расположенном в раскрытом меню, и введите в него текст "&Make Connection".
20. Поместите указатель мыши на заголовок раскрывающегося меню **Connect** (Соединить), нажмите левую кнопку мыши и, не отпуская ее, перетащите данное меню, поместив его между меню **View** (Вид) и меню **Help** (Справка).
21. Выделите идентификатор раскрывающегося меню **Edit** (Правка) и нажмите клавишу <Del>. Данное меню будет уничтожено.
22. Нажмите кнопку **Save All** (Сохранить все) на панели инструментов **Standard** (Стандартная) для сохранения внесенных изменений. В результате описан-

ных выше манипуляций окно редактирования ресурса меню примет вид, изображенный на рис. 16.2.

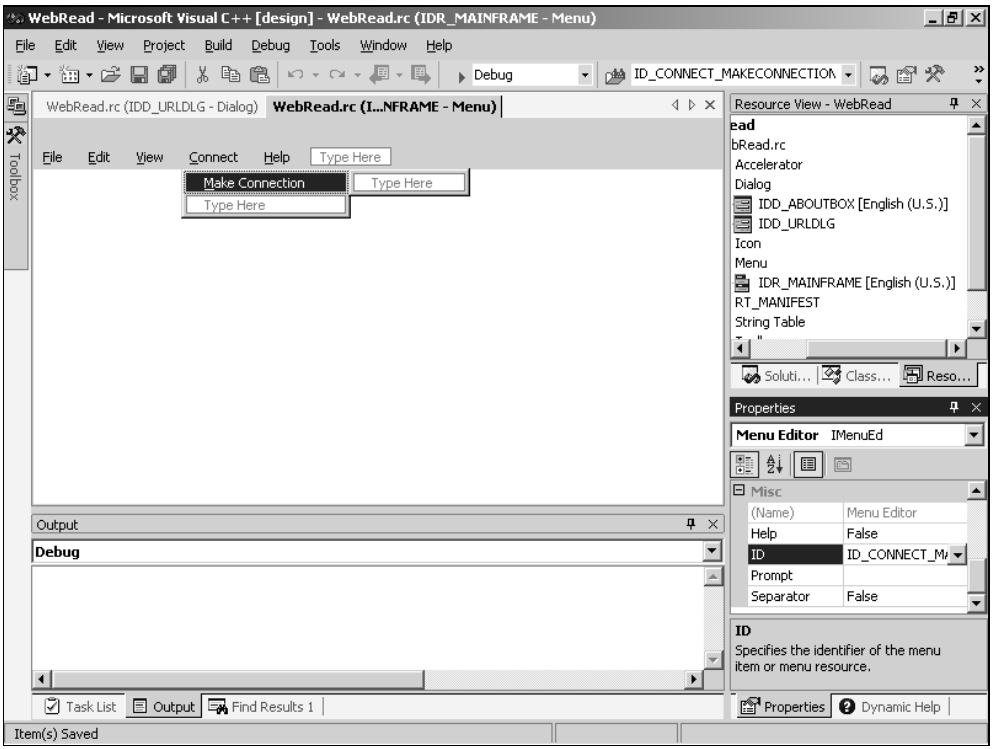


Рис. 16.2. Окно редактирования ресурса меню

23. Откройте окно **Class View** (Просмотр классов), щелкните правой кнопкой мыши на папке **WebRead** и выберите в появившемся контекстном меню команду **Add | Add Class** (Добавить | Добавить класс). Появится диалоговое окно **Add Class - WebRead** (Добавить класс), изображенное на рис. 16.3.
24. В окне списка **Templates** (Шаблоны) выделите значок **MFC Class** (Класс MFC) и нажмите кнопку **Open** (Открыть).
25. Появится диалоговое окно **MFC Class Wizard - WebRead** (Мастер создания классов MFC), изображенное на рис. 16.4.
26. Введите в текстовое поле **Class name** (Имя класса) `CURLDlg`, выделите в раскрывающемся списке **Base class** (Базовый класс) имя базового класса `CDialog` и нажмите кнопку **Finish** (Готово). В приложение будет включен новый класс.
27. В окне **Class View** (Просмотр классов) раскройте папку **WebRead**.

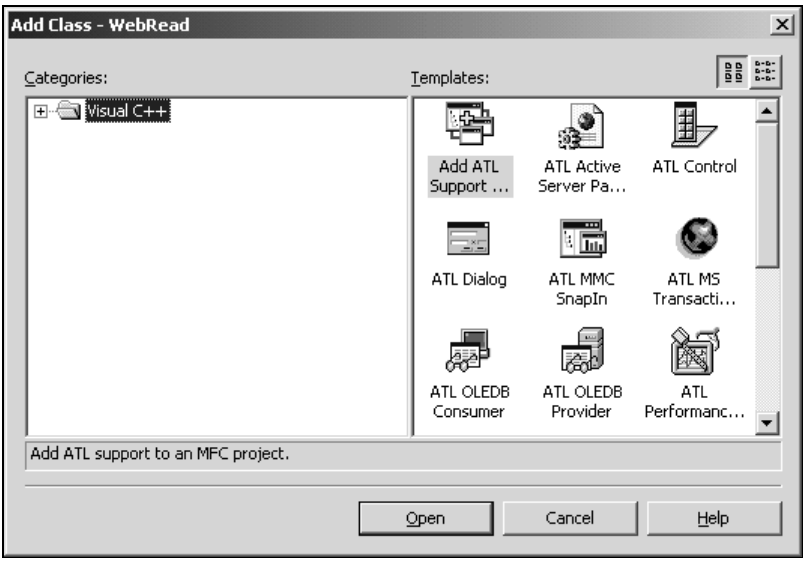


Рис. 16.3. Диалоговое окно **Add Class - WebRead**

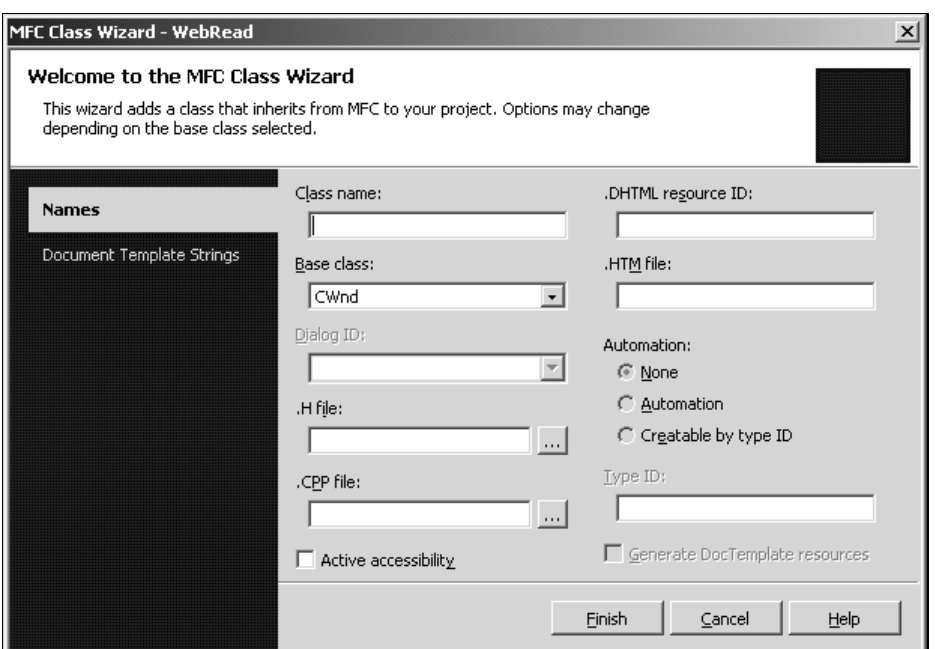


Рис. 16.4. Диалоговое окно **MFC Class Wizard - WebRead**

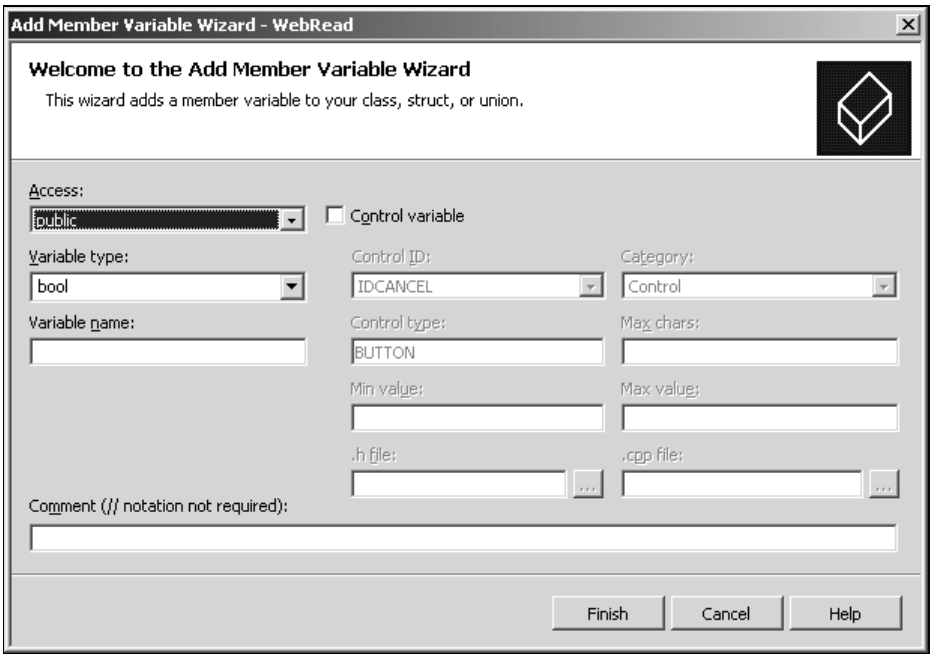


Рис. 16.5. Диалоговое окно **Add Member Variable Wizard**

28. Щелкните правой кнопкой мыши на имени класса `CURLDlg` и выберите в появившемся контекстном меню команду **Add | Add Variable** (Добавить | Добавить переменную). Появится диалоговое окно **Add Member Variable Wizard - WebRead** (Мастер добавления переменной в класс), изображенное на рис. 16.5.
29. Установите флажок **Control variable** (Связь с элементом управления), выделите в ставшем после этого доступным раскрывающемся списке **Control ID** (Идентификатор элемента управления) идентификатор ресурса `IDC_URL`, в раскрывающемся списке **Category** (Категория) выделите строку **Value** (Переменная), введите в текстовое поле **Variable name** (Имя переменной) идентификатор переменной `m_URL` и нажмите кнопку **Finish** (Готово). В класс `CURLDlg` будет добавлена новая переменная.
30. В окне **Class View** (Просмотр классов) щелкните правой кнопкой мыши на имени класса `CWebReadView` и выберите в контекстном меню команду **Properties** (Свойства). Откроется окно **Properties** (Свойства).
31. На панели инструментов окна **Properties** (Свойства) нажмите кнопку **Events** (События). Раскроется список событий, обрабатываемых данным классом.
32. В списке событий раскройте папку **ID_CONNECT_MAKECONNECTION** и выделите в ней команду `COMMAND`.
33. Раскройте связанный с этой командой список и выделите в нем единственную строку. В класс `CWebReadView` будет добавлена функция обработки дан-

ного сообщения. Одновременно откроется окно редактирования файла WebReadView.cpp и текстовый курсор будет помещен в первую строку заготовки данной функции.

34. Измените функцию `OnConnectMakeconnection` в соответствии с текстом листинга 16.1.

Листинг 16.1. Функция `CWebReadView::OnConnectMakeconnection`

```
// Функции обработки сообщений класса CWebReadView
void CWebReadView::OnConnectMakeconnection(void)
{
  // Создание класса диалогового окна
  CURLDlg  dlg(this);

  // Задание URL, используемого по умолчанию
  dlg.m_URL = "http://www.mcp.com";

  // Вывод диалогового окна
  int result = dlg.DoModal();

  if (result == IDOK)  // Была нажата кнопка OK
  {
    // Открытие сеанса связи
    CInternetSession  INetSession;
    CString    csTemp;
    LPCTSTR    szTemp;
    CEdit&      edit = GetEditCtrl();

    try
    {
      // Открытие файла HTTP
      szTemp = dlg.m_URL.GetBuffer(256);
      CHttpFile* httpFile =(CHttpFile*)INetSession.OpenURL( szTemp );
      dlg.m_URL.ReleaseBuffer();

      // Установка размера буфера чтения
      httpFile->SetReadBufferSize(4096);

      // Чтение файла и вывод его на экран
      while( httpFile->ReadString( csTemp ) )
        edit.ReplaceSel( csTemp + "\r\n" );
```

```
      // Закрытие файла
      httpFile->Close();
    }

    // Обработка исключения
    catch (CInternetException* pException)
    {
      edit.ReplaceSel( "CInternetException received!" );
      pException->ReportError();
    }

    // Закрытие сеанса связи
    INetSession.Close();

    // Перерисовка окна
    Invalidate();
  }
}
```

35. Перейдите в начало файла WebReadView.cpp и после строки `#include "WebReadView.h"` вставьте следующие строки:

```
#include <afxinet.h>
#include "URLDlg.h"
```

36. Нажмите клавишу <F5> и запустите приложение на исполнение.
37. Выберите команду меню **Connect | Make Connection** (Соединить | Создать соединение). Появится диалоговое окно **Connection** (Соединение), изображенное на рис. 16.6.

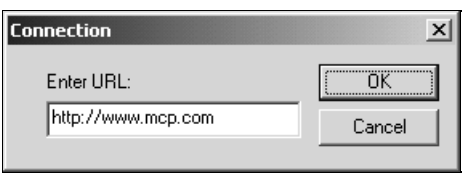


Рис. 16.6. Диалоговое окно **Connection**

38. Введите в текстовое поле **Enter URL** (Ввод URL) адрес Web-страницы или путь к файлу формата HTML и нажмите кнопку **OK**. Текст этой страницы или файла появится в главном окне приложения, как это показано на рис. 16.7. (Для того чтобы лишний раз не выходить в Сеть, здесь приведен текст одного из файлов справочной системы HTML.)

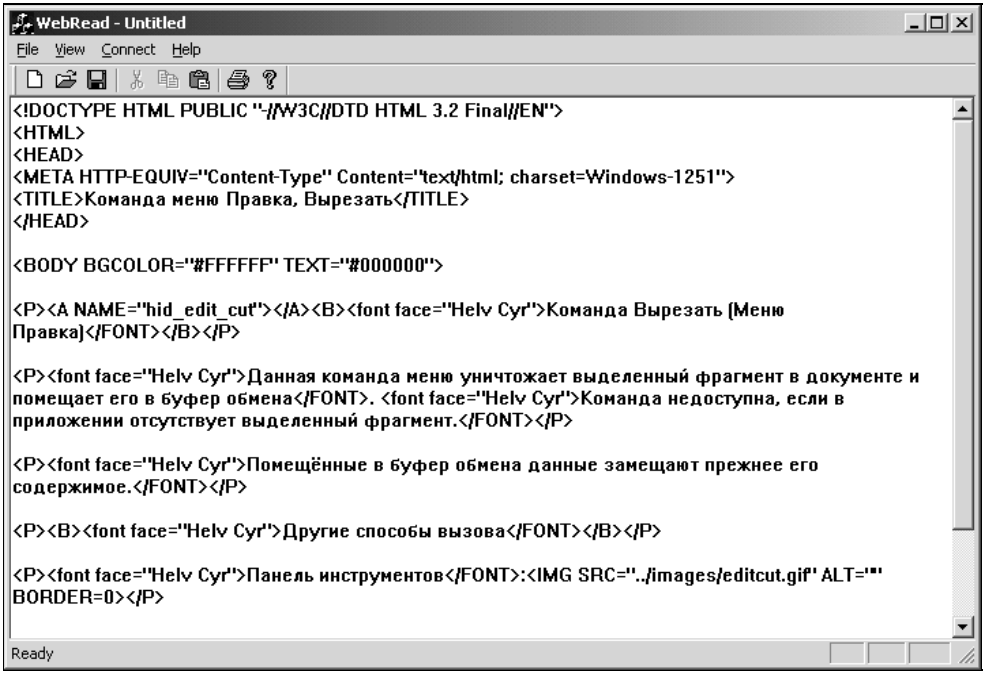


Рис. 16.7. Текст файла HTML

Как видно из приведенного примера, прочитать содержимое Web-страницы с использованием классов WinInet не просто, а очень просто. Все было бы хорошо, если бы дело ограничивалось только этой операцией, однако, по непонятной причине, большинство пользователей хотят видеть на экране не текст в формате HTML, а графическую Web-страницу. Однако эта операция, требующая синтаксического анализа полученного текста и выполнения содержащихся в нем команд, выходит за рамки рассматриваемого вопроса, а ее описанию посвящено множество других книг.

Для выполнения поставленной задачи в заготовку приложения, созданного мастером MFC Application Wizard, были внесены минимальные изменения: было создано диалоговое окно для задания пользователем адреса Web-страницы и добавлено меню для инициирования требуемой операции. Вопросы создания диалогового окна и работы с меню уже многократно обсуждались в этой книге, поэтому здесь им не будет уделено внимания. Рассматриваться будет только функция обработки сообщения `OnConnectMakeconnection`, в которой производятся все операции по созданию соединения, получению информации по данному соединению и по разрыву соединения.

В функции `OnConnectMakeconnection`, прежде всего, создается объект класса диалогового окна для получения адреса Web-страницы, содержимое которой хочет считать пользователь. Для простоты в текстовое поле данного диалогового

окна выводится адрес страницы Microsoft, что избавляет пользователя от необходимости вводить какую-либо информацию при отладке приложения. После этого для объекта класса диалогового окна вызывается функция `DoModal`, выводящая данное диалоговое окно на экран, и проверяется возвращаемое ею значение.

Если пользователь закрыл диалоговое окно нажатием кнопки **OK**, полученный адрес используется для доступа к Web-странице. При этом производятся следующие операции:

1. Создается объект класса `CInternetSession`.
2. Для этого объекта вызывается функция `OpenURL`, возвращающая указатель на объект класса `CHttpFile`, используемый для работы с файлом HTML, определяемым заданным URL.
3. Для объекта класса `CHttpFile` в цикле вызывается функция `ReadString`, считывающая в нем одну строку, которая тут же выводится на экран.
4. После того как вся информация из файла будет считана, для объекта класса `CHttpFile` вызывается функция `Close`, уничтожающая данный объект.
5. После этого одноименная функция вызывается для уничтожения объекта класса `CInternetSession`.

Создание объекта класса `CInternetSession` не представляет собой никаких затруднений, поскольку при его создании использовались значения аргументов конструктора данного класса, установленные по умолчанию. Значения этих аргументов подходят для большинства приложений.

Поскольку для простоты вывода в приложении текстовой информации в качестве базового класса представления был выбран класс `CEditView`, то для обращения из его функций к функциям элемента управления текстового поля в них, с использованием функции `CEditView::GetEditCtrl`, получается ссылка на объект класса `CEdit`.

Поскольку при создании соединения и при работе с ним могут быть вызваны исключения, весь программный текст, реализующий эти функции, помещен в блок `try`. Первый аргумент функции `CInternetSession::OpenURL` имеет тип `LPCTSTR`, а адрес URL содержится в объекте класса `CString`, то для получения указателя на внутренний блок данного объекта используется функция `CString::GetBuffer`. После использования данного указателя он освобождается функцией `CString::ReleaseBuffer`. Результатом вызова функции `OpenURL` является установление соединения с удаленным файлом, идентифицируемым указанным URL, и возвращение указателя на объект класса `CHttpFile`, используемого для доступа к данному файлу. Так как функция `OpenURL` возвращает указатель на объект класса `CStdioFile`, то этот указатель преобразуется к истинному типу объекта.

После получения указателя на объект класса `CHttpFile` для него с использованием функции `CInternetFile::SetReadBufferSize` задается размер буфера. Это связано с тем, что применяемые данным объектом класса WinInet API не под-

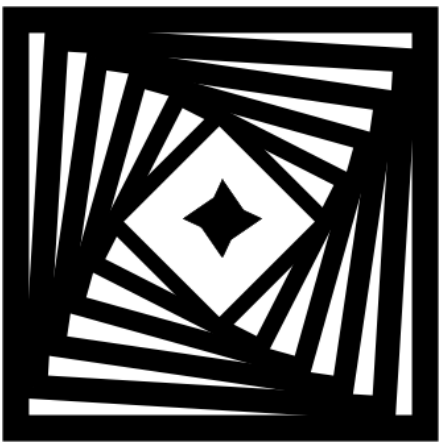
держивают буферизации передаваемых данных. Однако в данном случае вызов этой функции является излишней предосторожностью, поскольку буфер такого размера будет автоматически создан при первом вызове функции `CInternetFile::ReadString`, если перед ним не будет вызвана функция `SetReadBufferSize`.

Для чтения информации из удаленного файла используется функция `CInternetFile::ReadString`. Эта функция считывает информацию из файла по строкам, причем в конце получаемой строки отсутствуют символы возврата каретки и перевода строки. Поэтому данные символы приходится добавлять при выводе информации на экран. Эта операция производится функцией `CEdit::ReplaceSel`, заменяющей текст выделенного фрагмента указанным текстом. Если в тексте отсутствует выделенный фрагмент, текст вставляется в текущую позицию текстового курсора, после чего текстовый курсор перемещается в конец вставленного текста.

После чтения всей информации из файла соединение разрывается вызовом функции `CInternetFile::Close`, закрывающей объект класса `CInternetFile` и освобождающей связанные с ним ресурсы.

Если в процессе работы с соединением было вызвано исключение, управление передается блоку `catch`, в котором с использованием функции `CEdit::ReplaceSel` на экран выдается краткое сообщение об ошибке, а затем вызывается функция `CException::ReportError`, выводящая окно сообщения с более подробным описанием ошибки.

При завершении работы функции `OnConnectMakeconnection` в ней вызывается функция `CInternetSession::Close`, закрывающая объект класса `CInternetSession`, и функция `CWnd::Invalidate`, осуществляющая перерисовку главного окна приложения.



ЧАСТЬ IV

Приложения

Приложение 1. Объектно-ориентированное программирование и классы

Приложение 2. Интерфейс пользователя Visual C++

Приложение 3. Описание дискеты

Приложение 1

Объектно-ориентированное программирование и классы

Объектно-ориентированный подход лежит в основе программирования в среде Windows. Однако многие программисты, работавшие до этого под управлением DOS, и даже использовавшие библиотеки классов, например библиотеку Turbo Vision, разработанную фирмой Borland для своей системы программирования Borland C++, имеют довольно-таки смутное представление о классах и пользуются при программировании заготовками, предоставляемыми системой программирования, внося в них минимальные изменения. Такой подход возможен и в среде программирования Visual C++, однако для использования всех предоставляемых ею возможностей необходимо четко понимать, что такое классы, что они позволяют, а что не позволяют. В данном приложении дано краткое описание объектно-ориентированного подхода к программированию и описаны основные свойства классов. Данное изложение не претендует на полноту. Основное его назначение помочь читателю сформировать целостное представление по данному вопросу, а технические детали достаточно подробно описаны в справочной системе среды программирования.

Обзор объектно-ориентированных методов программирования

Прежде чем перейти к написанию собственных классов, вам необходимо, прежде всего, подробнее ознакомиться с концепцией объектно-ориентированного программирования. Эта концепция позволяет рассматривать программу как набор модулей, каждый из которых выполняет отведенную ему функцию. Если вам надо отправить письмо, то вы не идете на вокзал, не отыскиваете человека, едущего в тот же город и не просите его передать письмо по определенному адресу. Вы просто опускаете письмо в почтовый ящик и надеетесь, что оно вовремя дойдет до адресата. Точно так же, если вам необходимо вызвать на экран окно, то совсем не обязательно разбираться в том, каким образом отображается его рамка, а тем более, каким образом производится обработка поступающих в него и отправляемых им сообщений. Если вы попытаетесь это сделать, то что-нибудь упустите (а это естественно при отсутствии у вас полного описания системы) и, скорее всего, вызовете крах всей системы.

Однако концепция объектно-ориентированного программирования (ООП) не ограничивается только предоставлением пользователю удобных библиотек для решения сложных задач. Эти вопросы могут быть решены и без использования данной концепции, например, в рамках структурного программирования. Программные объекты были разработаны для облегчения процесса программирования и приближения структуры программ к привычным для человека формам.

Можно смело сказать, что именно способность обобщать, классифицировать и генерировать абстракции явилась причиной превращения обезьяны в человека. Без этой способности невозможно было бы создание языка общения, поскольку каждое имя существительное является по сути своей классом объектов, обладающих некоторым набором атрибутов или отличительных черт. С другой стороны, каждый подобный класс является базовым для других классов, формируемых, например, путем добавления к этому существительному конкретизирующего его прилагательного. Например, "Белый дом".

В основе концепции объектно-ориентированного программирования лежат три основных принципа.

- *Инкапсуляция*:

  совмещение структур данных с функциями (методами), предназначенными для манипулирования этими данными. Инкапсуляция достигается путем введения нового механизма структурирования и типизации данных — класса.

- *Наследование*:

  создание новых, производных классов, которые наследуют данные и функции от одного или нескольких ранее определенных базовых классов. При этом возможно переопределение или добавление новых данных и методов. В результате создается иерархия классов.

- *Полиморфизм*:

  присвоение методу единого имени или идентификатора в рамках иерархии классов таким образом, чтобы любой класс в иерархии имел возможность посвоему выполнять связанные с этим методом действия.

Одновременно с появлением и детализацией концепции появились и основанные на ней языки программирования. Одним из первых языков явился алгоритмический язык Modula 2. Язык программирования Turbo Pascal, разработанный фирмой Borland, начиная с версии 5.5, стал объектно-ориентированным. Но наиболее последовательное свое воплощение концепция объектно-ориентированного программирования нашла в алгоритмическом языке C++, разработанном сотрудником Bell Laboratories AT&T Брайаном Страуструпом на базе языка программирования C как средство моделирования больших телефонных коммутационных систем.

Рассмотрим подробнее основополагающие принципы ООП.

Инкапсуляция

Одним из главных отличий стандартного структурного программирования от объектно-ориентированного программирования является инкапсуляция. Совмещение структур данных с методами, предназначенными для манипулирования этими данными, позволяет сосредоточить в одном месте всю информацию, касающуюся определенных данных и методов их обработки и, тем самым, облегчить доступ к этим методам и данным из любой точки программы.

Действительно, представьте себе, что все данные в программе имеют глобальную область видимости (что неизбежно при тесном взаимодействии всех частей программы). При этом вам придется для всех данных и для каждого метода использовать уникальный идентификатор, причем этот идентификатор должен быть инфор-

мативным, т. е. длинным. Программы с такими идентификаторами станут практически нечитаемыми, особенно на экранах дисплеев. Вам придется постоянно следить, чтобы новые имена не пересекались со старыми, чтобы в каждой функции использовались именно те данные, для обработки которых она была написана. Кроме того, вам придется отказаться от таких элегантных возможностей, как перегрузка операций и создание собственных типов данных, позволяющих автоматически проверять корректность выполняемых над ними операций.

Кроме того, инкапсуляция позволяет вам скрыть данные и функции, используемые исключительно членами данного класса или членами производного от него класса, что упрощает описание класса для его использования внешними функциями и повышает надежность его работы, поскольку критически важные данные могут быть таким образом защищены от несанкционированного доступа.

Хорошим тоном считается скрытие всех данных, используемых данным классом, и обеспечение доступа к ним только посредством соответствующих методов класса. Это, прежде всего, устраняет все вопросы, связанные с областью видимости переменных, и позволяет пользователю использовать в каждом классе простые и информативные идентификаторы, не заботясь о том, что подобные идентификаторы уже используются в других классах.

Обеспечение доступа к данным исключительно через методы класса позволяет избегать потенциально очень опасных ситуаций. Предположим, что в процессе разработки какого-либо проекта одному из разработчиков была поставлена задача написать обработчик для некоторого массива, содержащего указатели на символьные строки. На каком-то этапе этот разработчик решил заменить этот массив связанным списком, и соответствующим образом переписал методы обращения к нему.

Если он оформил этот массив в виде класса и обеспечил доступ к массиву исключительно через методы данного класса, то другие разработчики ни при каких обстоятельствах не должны почувствовать последствия этой замены (разве что она может отразиться на скорости выполнения приложения). Но если к этому массиву был обеспечен прямой доступ, то никто не может поручиться, что другой программист, работающий с теми же данными, не захочет воспользоваться собственными, более эффективными, с его точки зрения, методами работы с массивом, которого уже нет.

Наследование

Принцип наследования в объектно-ориентированном программировании позволяет создать класс, включающий в себя все данные и методы некоторого другого класса, и, кроме того, содержащий свои, присущие только ему, данные и методы обработки. Такой подход позволяет строить целые иерархии классов, каждый следующий уровень которых отличается все большей детализацией по сравнению с предшествующими ему уровнями. Достоинства такого подхода наиболее полно проявляются при программировании достаточно сложных объектов, к которым по праву относится сама операционная система Windows.

При всем своем различии окно представления и диалоговое окно являются окнами Windows и, как таковые, имеют много общего. Представьте себе, во что превратилось бы описание класса диалогового окна, если бы в нем непосредственно были бы описаны все обрабатываемые им данные и все методы обработки, учитывая тот факт, что каждое текстовое поле и каждый переключатель, по сути, являются само-

стоятельным и полноправным окном Windows. Не говоря уже о том, сколько места займет описание каждого класса и сколько в них будет повторяющихся и практически аналогичных по тексту функций.

Наследование позволяет описать сначала общий класс окна Windows, на его основе описать общий класс диалогового окна Windows, а уже на основе этого класса сформировать класс конкретного диалогового окна, используемого для реализации конкретного пользовательского интерфейса.

Полиморфизм

Последней отличительной чертой объектно-ориентированного программирования является полиморфизм. Использование полиморфизма позволяет существенно расширить набор функций в базовом классе, что, в свою очередь, приводит к существенному сокращению текста программы и улучшению ее читабельности.

Греческое слово "полиморфизм" было заимствовано из биологии, где оно означает наличие в пределах одного вида резко отличающихся по облику особей. В C++ полиморфизм реализуется с помощью так называемых виртуальных функций, которые позволяют в пределах иерархии классов использовать одно имя метода для выполнения одной и той же операции над различными типами данных. Решение о том, какая именно версия данного метода будет использована в каждом конкретном случае, определяется на этапе исполнения программы и носит название позднего связывания.

Суть полиморфизма может быть наглядно проиллюстрирована на следующем примере. Предположим, что вы решили создать иерархию классов для манипулирования отображаемыми на экране графическими объектами. Вы создали базовый класс, в который включили, среди прочих, функцию перемещения графических объектов. Казалось бы, данную функцию очень просто реализовать: необходимо только определить координаты нового положения объекта, уничтожить его изображение на старом месте и нарисовать на новом.

Однако при ближайшем рассмотрении перед вами встает, казалось бы, неразрешимая задача: функции отображения конкретных графических объектов задаются в производных классах, а обращение к ним предполагается производить в базовом классе. Конечно, у данной проблемы существует простое решение: перенести функцию перемещения объекта из базового в производный класс и для каждого класса написать свой вариант функции. Однако, если следовать такой логике, в базовом классе практически не останется методов обработки и все преимущества наследования будут практически сведены на нет.

Поэтому, как уже упоминалось выше, в рамках ООП был разработан аппарат *виртуальных функций*. Если в производном и базовом классах одна и та же функция объявлена виртуальной (для этого функции должны иметь не только одно и то же имя, но и один и тот же набор формальных параметров и одинаковый тип возвращаемой величины), то при обращении к этой функции из любого метода, унаследованного данным классом от своих базовых классов, будет вызываться функция, определенная в данном классе, а не функция соответствующего базового класса, как это имело бы место в противном случае.

Если виртуальные функции имеют различный набор формальных параметров, то они рассматриваются как различные функции. Различный тип возвращаемой вели-

чины у виртуальных функций с одинаковым именем и набором формальных параметров рассматривается как ошибка.

Классы

Концепция ООП нашла свое выражение в классах C++, основные свойства которых будут рассмотрены в этом разделе.

Классы как типы данных

В сущности, классы являются просто определяемыми пользователем типами данных. Аналогично всем другим типам данных они позволяют создавать столько своих копий, сколько требуется для решения конкретной задачи. Возьмем, к примеру, стандартный тип данных `int`. Было бы странно, если бы в программе было разрешено использовать только одну переменную данного типа. Конечно, вы можете использовать столько целочисленных переменных, сколько вам понадобится.

То же самое относится и к классам. После того как вы описали новый класс, вы можете создавать столько переменных, имеющих тип данного класса, сколько будет нужно. Каждая переменная, называемая *объектом* данного класса, имеет полный доступ к функциям данного класса и свою копию хранящихся в нем данных. Поскольку каждый объект имеет свою копию данных, то изменения, внесенные в данные одного объекта, никак не отражаются на данных другого объекта.

Файлы заголовков и файлы реализации

Весь текст относительно простой программы может быть расположен в одном файле. Но такой подход становится недопустимым, когда размеры программы превышают определенные границы.

Во-первых, с таким файлом становится неудобно работать, поскольку часто приходится практически полностью пролистать весь файл, чтобы найти описание нужной функции. Любому понятно, что пролистать один из десяти файлов намного проще, чем один файл, включающий в себя содержимое этих десяти.

Во-вторых, не следует забывать, что объектно-ориентированное программирование развивалось в те времена, когда IBM PC имела объем памяти 64 Кбайт, внешним запоминающим устройством для нее служил кассетный магнитофон и никому, даже в страшном сне не могло привидеться, что кому-то может не хватить 640 Кбайт оперативной памяти. Из всего вышесказанного очевидно, что в те времена очень остро стоял вопрос о размерах файлов, загруженных в оперативную память. Поэтому при разработке алгоритмических языков особое внимание уделялось возможности компоновки программ из нескольких исходных файлов.

Поскольку классы представляли собой пользовательские типы данных, то можно было ожидать, что любой конкретный класс будет использоваться в любом из исходных программных файлов. Имевшиеся в то время средства обеспечения видимости переменных требовали, чтобы все обращения к любой переменной происходили, как минимум, в пределах одного файла. Для того чтобы обойти это ограничение, была предпринята попытка разделить описание классов и программную реализацию

используемых в них методов обработки. В результате появились *файлы заголовков* (header files)и *файлы реализации* (implementation files). Файлы заголовков обычно имеют расширение h и включают в себя краткое описание классов и других типов данных для включения их в файлы реализации, обычно имеющие расширение cpp, в которых предполагается использование классов и переменных, определенных в данном файле заголовка.

Рассмотрим типичный файл заголовка, созданный мастером MFC Application Wizard в Visual C++ версии 7.0 для приложения MDI, в котором все опции были выбраны по умолчанию. Для повышения информативности все его комментарии переведены на русский язык.

Листинг П1.1 Файл заголовка класса

```
// Файл MDIDoc.h содержит интерфейс класса CMDIDoc

#pragma once

class CMDIDoc : public CDocument
{
protected: // Создается только с использованием потоковых файлов
  CMDIDoc();
  DECLARE_DYNCREATE(CMDIDoc)

// Атрибуты
public:

// Операции
public:

// Перегруженные
  public:
  virtual BOOL OnNewDocument();
  virtual void Serialize(CArchive& ar);

// Реализация
public:
  virtual ~CMDIDoc();
#ifdef _DEBUG
  virtual void AssertValid() const;
  virtual void Dump(CDumpContext& dc) const;
#endif
```

```
protected:

// Функции обработки сообщения
protected:
  DECLARE_MESSAGE_MAP()
};
```

Первые строки листинга П1.1 являются строками комментария, в которых указано имя файла заголовка и дано краткое его описание: интерфейс класса `CClassDoc`. После комментария идет оператор условной компиляции `#pragma once`. Данный оператор включен для того чтобы следующий за ним текст был скомпилирован только однажды. Дело в том, что файлы заголовка включаются в программу оператором `#include`, осуществляющим непосредственную подстановку вместо себя содержимого указанного за ним файла. Оператор `#include` может быть включен и в файлы заголовков, если в них используются классы, определенные в других файлах заголовков. Таким образом, определение одного и того же класса может оказаться несколько раз включенным в файл реализации, что недопустимо.

Для того чтобы избежать такой ситуации, и используются *операторы условной трансляции*, которые проверяют, был ли уже включен в проект данный файл заголовка и если это так, текст, следующий за этим оператором, не включается в текст приложения.

Строка `class CMDIDoc : public CDocument` указывает на то, что последующий блок будет содержать описание класса `CMDIDoc`, базовым классом для которого является класс `CDocument`. Ключевое слово `public` перед именем базового класса означает, что образуемый класс будет иметь максимальные права доступа к данным и методам базового класса.

Блок описания класса представляет собой специальным образом организованный список элементов класса, к которым относятся хранимые в нем данные и методы их обработки. В данном случае блок описания класса открывается *атрибутом права доступа*. В данном случае это `protected`. Об атрибутах права доступа следует сказать подробнее.

Как уже говорилось выше, одним из преимуществ концепции объектно-ориентированного программирования является скрытие данных, т. е. запрет на доступ к некоторым данным класса со стороны внешних функций и классов. Различают три уровня ограничения доступа к компонентам класса.

Наиболее строго ограничен доступ к частным компонентам класса. К ним могут обращаться только методы данного класса. По умолчанию все компоненты класса являются частными. Для того чтобы установить подобный режим доступа для компонентов, описание которых следует за компонентами, имеющими другой режим доступа, необходимо перед их описанием поставить атрибут права доступа `private`, а после него поставить двоеточие. Естественно, что классом, в котором имеются только частные компоненты, невозможно воспользоваться, поскольку в нем недоступны не только данные, но и методы их обработки.

Наиболее свободным режимом доступа является глобальный режим. К компонентам с глобальным режимом доступа могут обращаться любые внешние функции и классы. Для того чтобы установить этот режим доступа для компонентов, необходимо

перед их описанием поставить атрибут права доступа `public` с последующим двоеточием.

Защищенный режим доступа занимает промежуточное положение между частным и глобальным режимами. К защищенным компонентам класса могут обращаться только члены данного и производного от него классов. Для того чтобы установить этот режим доступа для компонентов, необходимо перед их описанием поставить атрибут права доступа `protected` с последующим двоеточием.

Использование каждого из вышеописанных режимов доступа для каждого конкретного компонента класса определяется исключительно требованиями к надежности хранения и манипулирования хранящимися в классе данными, а также удобством пользования данным классом.

Атрибуты права доступа делят список элементов класса на секции, начинающиеся с данного атрибута, все элементы которых имеют указанный при открытии секции режим доступа. Единственным исключением может являться первая секция, для которой можно не указывать атрибут права доступа. В этом случае все перечисленные в ней элементы класса будут иметь режим доступа по умолчанию, т. е. являться частными элементами класса.

Обычно первыми методами, перечисляемыми при описании класса, являются его конструкторы. *Конструктор класса* представляет собой метод, вызываемый при создании объекта данного класса, и используемый для установки начальных величин хранящихся в нем данных, если имеется необходимость в такой установке. Если данные класса не нуждаются в начальной установке, то конструктор все равно описывается, но в него не включается никаких операторов. Такой конструктор называется пустым конструктором. Класс может иметь неограниченное количество различных конструкторов, различающихся набором своих формальных параметров. В языке C++ конструктор имеет то же самое имя, что и класс. В данном случае конструктор имеет имя `CMDIDoc`.

Другим обязательным методом любого класса является его *деструктор класса*, представляющий собой метод, вызываемый при уничтожении объекта данного класса. Класс может иметь только один деструктор, который не должен иметь списка формальных параметров. Обычно деструктор используется для уничтожения всех динамических переменных, созданных в процессе работы методов класса. Деструктор имеет то же самое имя, что и класс, но ему предшествует символ ~. В данном случае деструктор является виртуальной функцией с именем `~CMDIDoc`.

Непосредственно за конструктором класса следует макрос `DECLARE_DYNCREATE (CMDIDoc)`, имеющий синтаксис `DECLARE_DYNCREATE (class_name)`. Его единственным аргументом является переменная `class_name`, представляющая собой имя данного класса (не заключенное в кавычки). Данный макрос позволяет объектам, произведенным от класса `CObject`, динамически создаваться в процессе выполнения программы. Операционная система использует эту возможность для динамического создания новых объектов, например, при чтении объекта с диска, что необходимо, например, при работе с документами и представлениями.

Данный пример содержит описание двух генерируемых мастером MFC Application Wizard виртуальных функций: `OnNewDocument`, имеющей выходную величину логического типа, и `Serialize`, не возвращающую значение и имеющую своим аргументом ссылку на объект класса `CArchive`. В него также включены отладочные функции `AssertValid` и `Dump`.

Рассмотрим теперь файл реализации для данного класса.

Листинг П1.2. Файл реализации класса

```
// Файл MDIDoc.cpp содержит реализацию класса CMDIDoc
//
#include "stdafx.h"
#include "MDI.h"

#include "MDIDoc.h"

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

// Класс CMDIDoc
IMPLEMENT_DYNCREATE(CMDIDoc, CDocument)

BEGIN_MESSAGE_MAP(CMDIDoc, CDocument)
END_MESSAGE_MAP()

// Конструктор и деструктор класса CMDIDoc
CMDIDoc::CMDIDoc()
{
  // Сюда помещается текст конструктора
  // времени выполнения приложения
}

CMDIDoc::~CMDIDoc()
{
}
BOOL CMDIDoc::OnNewDocument()
{
  if (!CDocument::OnNewDocument())
    return FALSE;

  // Сюда помещают процедуру функции, исполняемую при создании нового
  // документа (Документы SDI будут повторно использовать старый документ)
```

```
  return TRUE;
}

// Потоковое сохранение и восстановление объекта класса CMDIDoc
void CMDIDoc::Serialize(CArchive& ar)
{
  if (ar.IsStoring())
  {
    // Сохранение документа
  }
  else
  {
    // Восстановление документа
  }
}

// Диагностические функции класса CMDIDoc
#ifdef _DEBUG
void CMDIDoc::AssertValid() const
{
  CDocument::AssertValid();
}

void CMDIDoc::Dump(CDumpContext& dc) const
{
  CDocument::Dump(dc);
}
#endif //_DEBUG

// Команды класса CMDIDoc
```

Файл реализации так же, как и файл заголовка, начинается с комментария, в котором указывается, что это за файл. Затем следуют операторы `#include` для всех файлов заголовка, которые необходимо включить в данную программу. После этого начинается непосредственное описание реализации методов данного класса.

Описание начинается с макроса `IMPLEMENT_DYNCREATE(CMDIDoc, CDocument)`, имеющего синтаксис `IMPLEMENT_DYNCREATE(class_name,base_class_name)`, где аргумент `class_name` соответствует имени данного класса, а аргумент `base_class_name` — имени базового класса. Этот макрос является парой для макроса

`DECLARE_DYNACREATE`, описанного выше, и может использоваться исключительно в паре с ним. Ни один из этих макросов не может использоваться поодиночке.

Сразу за этим макросом находится описание конструктора и деструктора данного класса. Данный пример является не совсем удачным, поскольку в нем конструктор базового класса не имеет формальных параметров, которые должны быть для него определены в производном классе. В противном случае заголовок описания метода конструктора выглядел бы не `CMDIDoc::CMDIDoc()`, как в данном примере, а, например `CMDIDoc::CMDIDoc(int par):CDocument(par)`, хотя в действительности класс `CDocument` не имеет конструктора, содержащего формальные параметры.

Такой способ передачи параметров в конструктор является одним из наиболее существенных недостатков языка программирования C++, поскольку существенно затрудняет преобразование параметров при их передаче конструктору базового класса. Например, в языке программирования Turbo Pascal конструкторы и деструкторы имели предопределенные имена и могли вызываться в любой точке конструктора производного класса.

Наибольшие сложности при передаче конструктору базового класса необходимых ему параметров способом, принятым в C++, возникают при проведении существенных преобразований параметров данного класса перед передачей их в базовый класс. Такие преобразования могут, например, потребовать использования кусочно-линейной аппроксимации, параметры которой зависят от некоторой величины перечислимого типа, или проведения достаточно сложных математических вычислений, например вычисления логарифма по основанию два и преобразования полученной величины к наименьшему целому, превышающему полученную величину. Эти примеры не выдуманы мною, а взяты из конкретной жизни.

Другим неприятным следствием такого метода передачи данных является невозможность вызова одного конструктора класса из другого. Предположим, что вам надо создать два конструктора, отличающихся исключительно необходимостью задания исходной величины одному единственному параметру, а решить этот вопрос с использованием значений по умолчанию не позволяет наличие уже имеющегося списка формальных параметров с заданными начальными значениями, не обеспечивающего для большинства практических случаев минимальное число задаваемых параметров. Казалось бы, существует простое решение данного вопроса: задать в конструкторе начальное значение данного параметра, а потом вызвать другой конструктор для инициализации остальных величин. Однако такой фокус не проходит, и вам придется писать два полноценных конструктора с практически идентичным текстом.

Описание деструктора и остальных функций ничем не отличатся от описания обычных неклассовых функций. Единственным исключением является необходимость указания перед именем каждой функции, реализующей тот или иной метод класса, имени этого класса. Такой способ позволяет помещать описания методов класса в любой файл проекта. Иногда такое разнесение описания класса по нескольким файлам оказывается очень удобным. При описании методов класса следует обращать особое внимание на права доступа к переменным, поскольку в самый неподходящий момент может оказаться, что вы не имеете права доступа к переменной, которую собираетесь использовать в данном классе.

Когда следует, а когда не следует использовать классы

Объектно-ориентированное программирование поднимает процесс программирования на новую ступень. Когда программист осознает все возможности, предоставляемые ООП, он начинает использовать классы где надо и где не надо. К счастью или к сожалению, для программиста, использующего библиотеку классов MFC, этот вопрос несколько теряет свою актуальность, поскольку он просто обязан практически каждое свое действие оформлять классом.

Однако и в этом случае необходимо трезво оценить глубину иерархии создаваемых пользователем классов. Не следует забывать, что C++ является не только объектно-ориентированным, но и процедурным языком. Другими словами, программисты, использующие C++, могут воспользоваться лучшим, что имеется в каждом из этих подходов к программированию и что наиболее соответствует сущности решаемой ими задачи.

Использование классов наиболее целесообразно для обработки разнородной информации, имеющей в своей основе общие черты. Например, целесообразно создать базовый класс для работы со строками, а потом на его основе создавать классы для работы с конкретными строками в текстовом редакторе. Это позволит достаточно просто определить множество разнообразных классов, обрабатывающих символьную информацию на основе единых принципов и методов обработки.

Однако за все приходится платить. При обращении к методу класса приходится использовать более сложный синтаксис, чем при обращении к обычной функции. При таком обращении необходимо указать не только имя самой функции, но и имя содержащего ее объекта. Кроме того, создание классов требует определенных затрат на формирование его структуры и разработку его концепции. Поэтому всегда следует оценивать дополнительные затраты и получаемые преимущества от применения ООП.

Одним из преимуществ классов является возможность создания их библиотек. Программисты, работающие в какой-либо определенной области, вынуждены постоянно пользоваться для решения своих задач одним и тем же набором функций, которые целесообразно объединить в классы и организовать дело таким образом, чтобы программирование каждой новой задачи сводилось бы к использованию нескольких объектов определенных заранее классов, в каждом из которых вызывалось бы ограниченное количество методов обработки. Такой подход не только позволит сократить количество ошибок при программировании задачи и ускорить сам процесс программирования, но и повысит читабельность программы.

Перегрузка функций и операторов

Одним из основных отличий языка программирования C от C++ является возможность перегрузки функций и операторов. В C каждая функция должна была иметь уникальное имя. Функции, выполняющие одну и ту же операцию над данными разных типов, должны были иметь различные имена. Применение в C++ *перегружаемых функций* позволило различать функции не только по имени, но и по типу формальных параметров, т. е. использовать одно и то же имя функции для выполнения сходных операций над данными, имеющими различный тип.

Перегрузка функций является мощным инструментом программирования, но всегда следует трезво оценить, является ли перегрузка функции лучшим решением в дан-

ном конкретном случае. Предположим, что вам нужно создать конструктор для класса, при инициализации которого необходимо задать два целочисленных параметра, однако, в большинстве случаев второй параметр принимается равным нулю.

Данная задача может быть решена с использованием перегрузки функций, когда будет создано две версии конструкторов, например:

```
SomeClass(int);
SomeClass(int,int);
```

Другим, более красивым решением данной задачи может явиться *использование аргументов по умолчанию*. В этом случае необходимо будет создать только один конструктор с двумя формальными параметрами, в котором второму формальному параметру будет присваиваться значение по умолчанию. Этот конструктор будет выглядеть следующим образом

```
SomeClass(int,int=0)
```

При создании объектов данного класса допустимо задавать как один, так и два параметра. В случае, когда будет задан только один параметр, он будет сопоставлен первому формальному параметру, а значение второго формального параметра будет принято равным нулю.

Одной из наиболее привлекательных возможностей, предоставляемых языком программирования C++, является *перегрузка операторов*. С точки зрения этого языка любой оператор является просто особой разновидностью функции и, как и любую другую функцию, его можно определить и перегрузить. Например, оператор `argument1+argument2` полностью эквивалентен функции `SomeClass argument1.operator +(argument2)`, где оба аргумента являются объектами класса `SomeClass`, в котором объявлена соответствующая функция.

Как с точки зрения простоты, так и с точки зрения читабельности программы? выражение `string1 += string2` предпочтительнее выражения `strcat(string1, string2)`, хотя для реализации первого выражения требуется создать специальный класс для работы со строковыми переменными и определить в нем оператор `+=`, а второе выражение является стандартной функцией C++. Но, при всей своей привлекательности, перегрузка операторов таит в себе множество потенциальных опасностей.

Набор символов стандартных операций языка C++ достаточно ограничен и все из них имеют интуитивно подразумеваемый смысл. Поэтому, например, использование знака операции `+` для реализации метода объединения двух массивов из двух элементов в один массив из четырех элементов вызовет у большинства пользователей неправильные ассоциации, что приведет к росту числа ошибок при его использовании. Хотя, с другой стороны, использование этого знака операции для объединения двух строк из двух символов в одну строку из четырех символов большинством пользователей будет воспринято адекватно.

Перегрузка префиксной или постфиксной *унарной операции* может быть выполнена при помощи нестатической функции компонента, не принимающего аргументов, либо при помощи функции, не являющейся методом, т. е. не являющейся членом класса, и принимающей один аргумент. Таким образом, оператор отрицания для объекта `x`, принадлежащего к классу `SomeClass`, может быть описан как

```
x. operator -()
```

или как

```
operator -(x)
```

Перегрузка бинарной операции может быть выполнена при помощи объявления нестатической функции компонента, принимающей один аргумент или при помощи функции, не являющейся методом и принимающей два аргумента. В последнем случае функция обычно объявляется дружественной (`friend`). Это практически единственный случай, в котором оправдано применение дружественных функций.

Дружественные функции не являются членами класса, хотя их описание располагается в блоке описания класса. От обычных функций они отличаются тем, что имеют доступ к частным и защищенным элементам данного класса. С точки зрения программиста, модификатор `friend` можно рассматривать просто как непременный атрибут описания функции, реализующей бинарную операцию и принимающей два формальных параметра. Таким образом, оператор вычитания для объекта `y` из объекта `x`, каждый из которых принадлежит к классу `SomeClass`, может быть описан как `x.operator -(y)` или как `operator -(x,y)`.

Операция присваивания = может быть перегружена только при помощи объявления нестатического метода. В отличие от прочих операций, функция операции присваивания не может наследоваться производными классами. Если эта операция не определена в классе, то по умолчанию ее результатом является покомпонентное присваивание элементов класса источника элементам класса приемника. Такое присваивание может привести к серьезным проблемам, если хотя бы один элемент данного класса является указателем.

Использование виртуальных функций

Виртуальные функции позволяют производным классам обеспечивать различные версии функций базового класса. Как уже отмечалось выше, виртуальные функции позволяют вызывать в методах базового класса методы производного класса. Необходимость в этом возникает в том случае, когда базовый класс обеспечивает описание некоторой базовой процедуры обработки, которая соответствующим образом детализируется в каждом из производных классов.

Необходимо отметить, что язык программирования C++ и без применения аппарата виртуальных функций допускает создание в базовом и производных классах функций, имеющих не только одно и то же имя, но и один и тот же список формальных параметров. Однако в этом случае в методах базового класса будет вызываться соответствующая функция базового класса, а в методах производного класса — функция производного класса, т. е. в процессе одной и той же обработки под одним и тем же именем могут вызываться две совершенно различные функции. Иногда это может быть полезно, но требует за собой пристального внимания со стороны программиста.

Перед тем как решиться на использование виртуальной функции, оцените, чем отличаются классы в иерархии. Должны ли они выполнять различные действия? Или все различия сводятся к обработке различных значений? Например, при отображении различных графических объектов виртуальные функции использовались для отображения этих объектов. В каждом объекте имелся метод, ответственный за его вывод на экран, и именно он использовался в функциях базового класса. С другой стороны, нет никакой необходимости использовать виртуальные функции, напри-

мер, для задания цвета объекта, поскольку реализация данного метода не зависит от конкретного объекта.

Ярчайшим примером применения виртуальных функций являются виртуальные деструкторы класса. Использование виртуальных деструкторов настолько важно, что можно сформулировать правило: "Деструктор должен быть виртуальным!"

Это правило связано с тем, что во многих функциях в качестве аргументов используются указатели на объект базового класса, а в действительности им передаются указатели на объект производного класса. Это позволяет существенно повысить область применения данных функций и исключить необходимость написания практически идентичных функций для всей иерархии классов.

В этом случае все идет нормально до тех пор, пока не возникает необходимость уничтожить этот объект. В некоторых случаях эта ситуация возникает в связи с тем, что передаваемый объект полностью используется в данной функции и нет нужны возвращать его приложению, а его уничтожение в этой функции избавляет программиста от требования каждый раз уничтожать объект после вызова этой функции. Это можно рассматривать как дополнительный сервис, без которого, в принципе, можно обойтись. Другим случаем, когда возникает необходимость уничтожения динамических переменных, является вызов исключения. Если перед ним не уничтожить динамические объекты, управление будет передано программе, не имеющей о них ни малейшего представления, и выделенная под них память будет утеряна.

Проблема, возникающая при уничтожении объекта, связана с тем, что, если класс не имеет виртуального деструктора, для объекта будет вызван деструктор того класса, указатель на объект которого был передан функции, т. е. деструктор базового класса. Во многих случаях это неприемлемо. Если же классы имеют виртуальные деструкторы, то для объекта всегда вызывается деструктор его класса независимо от того, указатель на объект какого класса был передан функции.

Единственным исключением из этого правила можно считать классы, используемые для расширения возможностей стандартных типов. Эти классы имеют одну переменную, имеющую расширяемый тип, и несколько функций, осуществляющих расширение и дополнительные обработки. В этом случае принципиально важно, чтобы размер объекта этого класса совпадал с размером объекта расширяемого типа. Поскольку при использовании виртуальных функций в класс включается таблица виртуальных функций, размеры класса при этом увеличиваются и это требование не выполняется.

Область действия класса

Имя элемента класса имеет область действия, локальную к области действия самого класса. Имя элемента класса может использоваться:

- в методах данного класса;
- в выражениях типа `X.M`, где `X` — объект данного класса, а `M` — имя элемента класса;
- в выражениях типа `lpX-> M`, где `lpX` — указатель на объект данного класса;
- в выражениях типа `SomeClass::M` или `DerivedClass::M`, где `SomeClass` — имя данного класса, а `DerivedClass` — имя производного от него класса;
- в ссылках вперед в пределах класса, к которому принадлежит данный элемент.

Классы, перечисляемые данные и имена `typedef`, определенные в пределах данного класса, либо имена функций, объявленных дружественными данному классу, не являются элементами данного класса и имеют ту же область действия, что и сам класс.

Имя может быть скрыто явным объявлением того же самого имени в окружающем блоке или классе. Скрытый элемент класса остается, тем не менее, доступным при помощи модификатора области действия, заданного с именем класса: `SomeClass::M`. Скрытое имя с глобальной областью действия допускает ссылку на него при помощи унарной операции `::`, например, `::M`.

Так, например, в любом методе класса, содержащего виртуальный или простой метод `SomeFunction(...)`, имеющий то же имя и список формальных параметров, что и метод базового класса, может быть непосредственно вызван соответствующий ему метод базового класса с использованием модификатора области действия `BaseClass::SomeFunction(...)`, где `BaseClass` — имя базового класса. С другой стороны, если имеется глобальная функция, имеющая то же самое имя, что и метод класса, для ее вызова следует воспользоваться следующим синтаксисом: `::SomeFunction(...)`.

Проверка допустимости имени в конкретной области действия производится в следующем порядке:

1. Сначала само имя проверяется на наличие неоднозначностей. Если в пределах области действия неоднозначности отсутствуют, то инициализируется последовательность доступа.
2. Если ошибок управления доступа не обнаружено, то проверяется тип объекта, класса и т. д.
3. Если имя используется вне функции или класса, либо имеет префикс унарной операции доступа к области действия `::`, если оно не уточняется бинарной операцией `::` или операциями выбора элемента класса `.` или `->`, то это имя должно быть именем глобального объекта, функции или нумератора.
4. Если идентификатор `SomeIdent` встречается в одном из следующих контекстов: `SomeClass::SomeIdent`, `SomeObj.SomeIdent` или `SomePtr-> SomeIdent`, где `SomeClass` — имя некоторого класса, `SomeObj` — имя объекта некоторого класса, а `SomePtr` — указатель на объект некоторого класса, то этот идентификатор `SomeIdent` является именем элемента данного класса или именем элемента производного от него класса.
5. Любое, не рассмотренное до сих пор имя, используемое в качестве статического или нестатического метода, должно быть объявлено в блоке, в котором оно встречается, либо в окружающем блоке, либо являться глобальным методом. Объявление в блоке идентификатора с именем, совпадающим с именем идентификатора, определенного в окружающем блоке, или именем глобальной переменной, скрывает имена внешних для блока идентификаторов. Имена в различных областях действия не могут быть перегружены.
6. Имя аргумента функции в определении функции находится в области действия самого внешнего блока данной функции. Аргумент по умолчанию вычисляется в каждой точке вызова.
7. Инициализатор конструктора вычисляется в области действия самого внешнего блока конструктора, и поэтому разрешает ссылки на имена аргументов конструктора.

Приложение 2

Интерфейс пользователя Visual C++

Microsoft Visual C++ является одним из компонентов среды разработки Visual Studio.NET, все компоненты которой используют единый интерфейс пользователя. Интерфейс Visual Studio представляет собой существенно измененный интерфейс Developer Studio, использовавшийся в Visual C++ версий 5.0 и 6.0.

Окно Visual Studio состоит из нескольких независимых окон, обычно отображаемых в виде панелей, и панелей инструментов, являющихся, по сути, полноценными окнами. Список панелей инструментов можно получить, щелкнув правой кнопкой мыши по любой отображаемой панели инструментов. Этот список представляет собой контекстное меню, позволяющее выбирать окна и панели инструментов, отображаемые в данный момент на экране. Окно Visual Studio со стандартным набором окон и раскрытым контекстным меню показано на рис. П2.1.

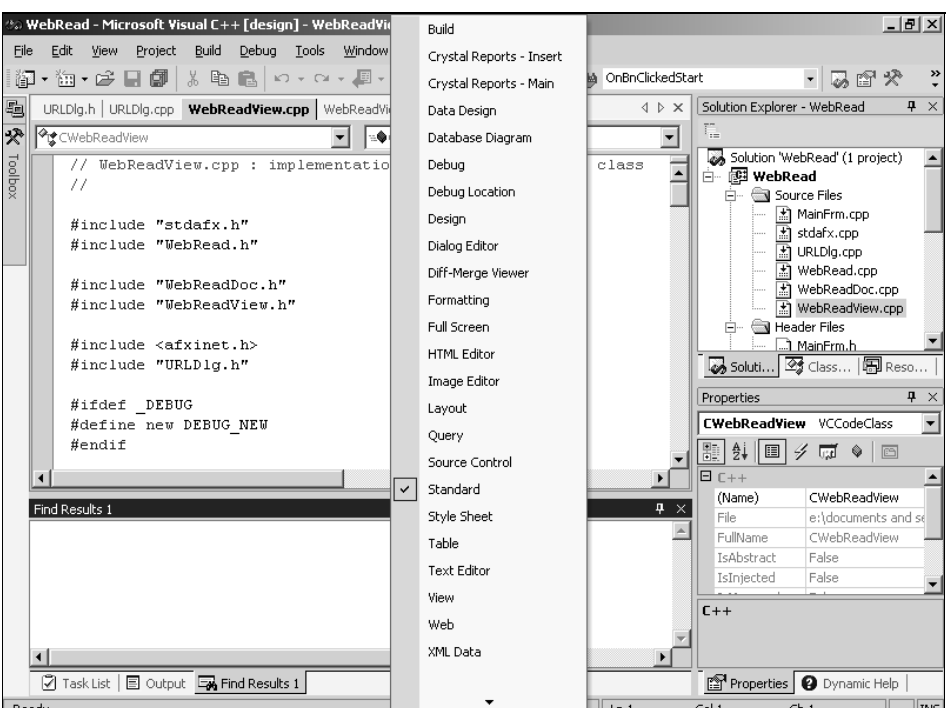


Рис. П2.1. Окно **Visual Studio**

Чтобы стандартные окна и панели инструментов перестали бы отображаться в виде панелей, а превратились бы в полноценные окна, их необходимо переместить в центр окна Visual Studio. Однако в этом практически никогда не возникает необходимости. Стандартные окна Visual C++ объединены в панели. Чтобы превратить эту панель в полноценное окно, достаточно дважды щелкнуть левой кнопкой мыши на ярлыке этой панели.

Часто этот двойной щелчок происходит непреднамеренно и разработчику приходится превращать это окно обратно в панель, что представляет собой достаточно неприятную задачу, особенно при отсутствии опыта. В большинстве случаев для того, чтобы превратить самостоятельное окно в панель, достаточно поместить указатель мыши на заголовок этого окна и переместить указатель на заголовок окна, вкладкой которого должно стать данное окно. При этом рамка перетаскиваемого окна принимает форму вкладки. После вставки новой вкладки в окно порядок следования ярлыков вкладок можно изменить простым перетаскиванием.

Интерфейс пользователя Visual C++ включает в себя комплекс, состоящий из окон, инструментальных средств, меню, панелей инструментов и других элементов, позволяющих использовать данную среду программирования для создания и отладки приложения. Кроме того, Visual C++ позволяет работать и с документами других приложений, таких как Microsoft Excel или Microsoft Word. Для этого используется технология ActiveX. Поскольку все эти приложения созданы одной фирмой, которая, к тому же, разработала технологию ActiveX, то можно надеяться, что работа с этими документами не приведет к зависанию системы.

Visual C++ позволяет настроить интерфейс в соответствии с предпочтениями пользователя. Он может создать собственные панели инструментов, меню и комбинации клавиш. Однако этим не стоит злоупотреблять, поскольку привыкнув к своей нестандартной конфигурации, вы будете испытывать серьезные затруднения при работе в стандартной конфигурации панелей, установленной на других машинах, что может существенно повредить вашей репутации программиста.

Первая страница

При первом запуске Visual C++ пользователь выходит в первую страницу данного приложения, изображенную на рис. П2.2 (на этом рисунке приведен не исходный вид данной страницы, а ее вид после нескольких сеансов работы с приложением). Чтобы получить доступ к первой странице в процессе работы с Visual C++, достаточно выбрать команду меню **Help | Show Start Page** (Справка | Вывести первую страницу).

На раскрывающейся при вызове данной страницы вкладке **Get Started** (Начало работы) имеется возможность открыть текущий, существующий или создать новый проект, а также просмотреть отчет об ошибках Visual Studio.NET.

Вкладка **What's New** (Что нового) первой страницы позволяет получить информацию о новых свойствах данной среды разработки, вкладка **Online Community** (Поддержка специалистов) — информацию по различным вопросам разработки приложений, размещенную в Internet, вкладка **Headlines** (Последние новости) — позволяет получить доступ к интерактивной справочной системе MSDN, также размещенной в Internet, а вкладка **Search Online** (Поиск в сети) позволяет производить поиск в Сети.

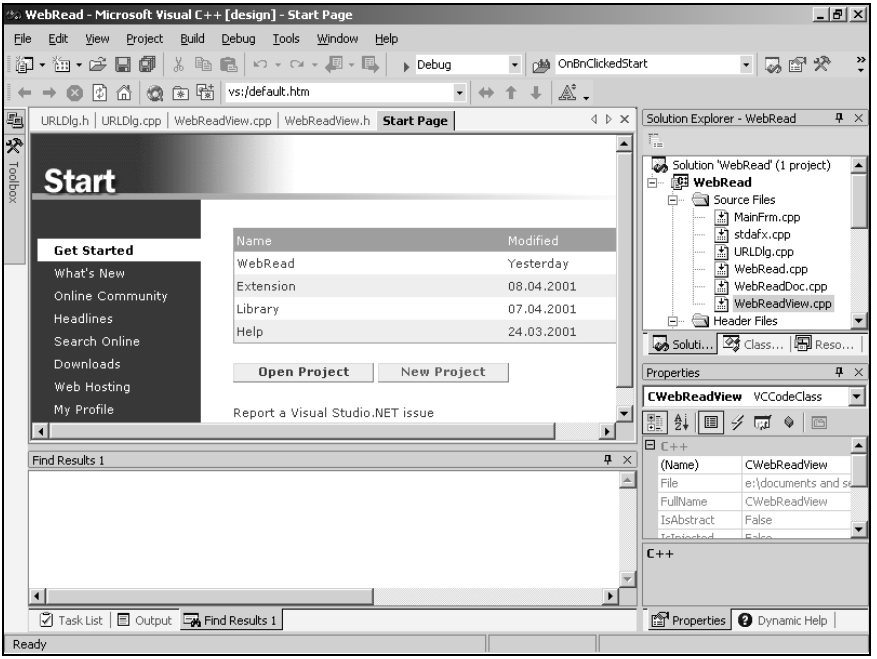


Рис. П2.2. Первая страница Visual C++

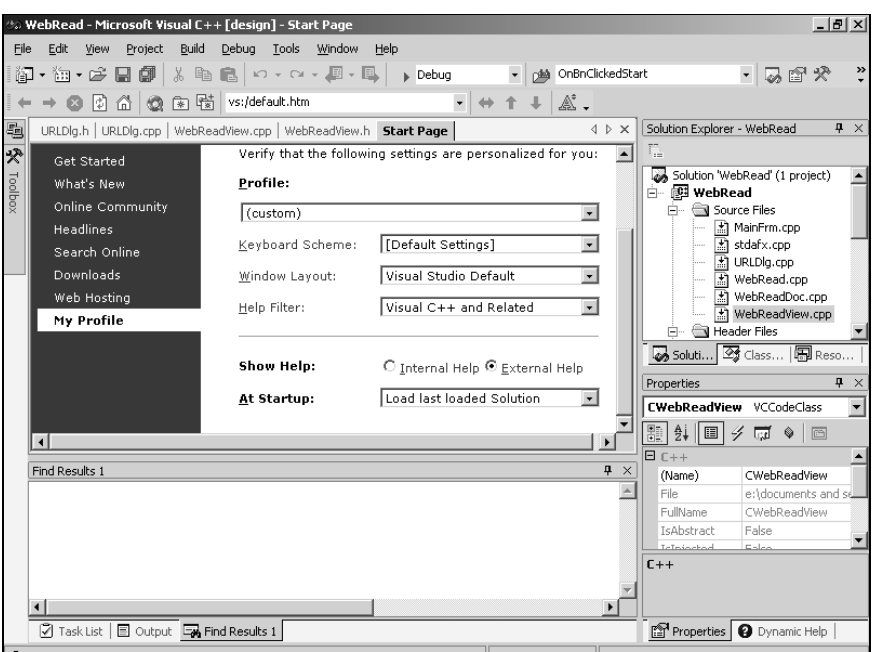


Рис. П2.3. Настройка профиля разработчика

Особое внимание следует обратить на вкладку **My Profile** (Мой профиль), приведенную на рис. П2.3. Эта вкладка позволяет настроить среду программирования Visual C++ в соответствии с требованиями конкретного разработчика.

Данная вкладка состоит из нескольких раскрывающихся списков, имеющих следующее назначение:

- **Profile** (Профиль) — позволяет выбрать профиль в соответствии с используемым для разработки языком программирования;
- **Keyboard Scheme** (Раскладка клавиатуры) — позволяет выбрать разработчику привычную раскладку клавиатуры;
- **Windows Layout** (Расположение окон) — определяет расположение окон в среде разработки;
- **Help Filter** (Фильтр справки) — ограничивает объем выводимой справочной информации;
- **At Startup** (Показать при запуске) — определяет, что будет выводиться на экран при запуске среды программирования.

Кроме того, данная вкладка содержит переключатель **Show Help** (Показать справку), определяющий способ вывода справочной информации в среде программирования. По умолчанию этот переключатель установлен в положение **Internal Help** (Внутренняя справка). При таком положении переключателя справочная информация выводится в стиле Visual C++ 5.0, т. е. в окно среды программирования. При установке данного переключателя в положение **External Help** (Внешняя справка) справочная информация выводится в стиле Visual C++ 6.0, т. е. в отдельное приложение.

Панели инструментов

При первом запуске Visual C++, использующего стандартную конфигурацию панелей инструментов, непосредственно под панелью меню отображается стандартная панель инструментов. Она содержит кнопки, необходимые при работе практически в любом режиме функционирования Visual C++. При работе пользователя в редакторе ресурсов, как правило, на экран выводятся дополнительные панели инструментов.

Конкретный набор панелей инструментов, отображаемых на экране, выбирается пользователем исходя из его предпочтений и используемого им разрешения дисплея. Помимо выбора отображаемых панелей инструментов, пользователь может выбрать и содержимое каждой панели инструментов. Ему предоставлена возможность убрать любую кнопку из стандартной панели инструментов и добавить в нее кнопки из некоторого набора.

Чтобы выбрать набор панелей инструментов, отображаемых на экране:

1. Щелкните правой кнопкой мыши по любой выведенной на экран панели инструментов.
2. В появившемся контекстном меню установите или сбросьте флажок у соответствующей панели инструментов. На экране появится или с экрана исчезнет данная панель инструментов.

3. Повторите п.п. 1 и 2 для других панелей инструментов.

   или

   Выберите команду меню **Tools | Customize** (Сервис, Настройка). Появится диалоговое окно **Customize** (Настройка).

4. Раскройте вкладку **Toolbars** (Панели инструментов). Диалоговое окно **Customize** (Настройка) примет вид, изображенный на рис. П2.4.

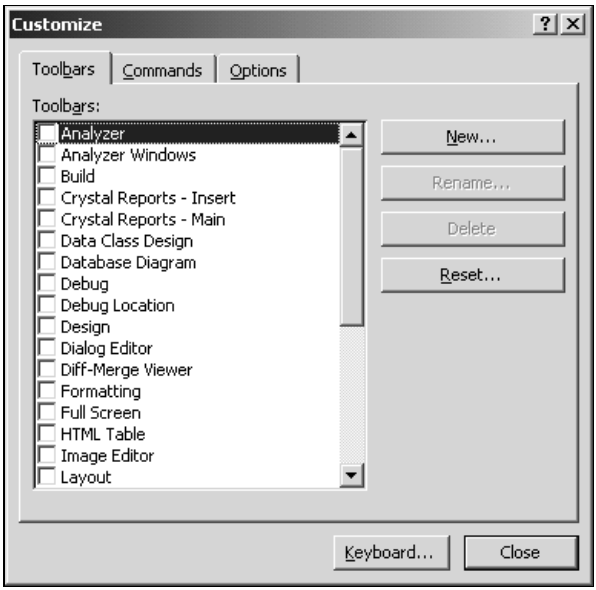


Рис. П2.4. Диалоговое окно **Customize**, вкладка **Toolbars**

5. Установите или сбросьте флажки в окне списка **Toolbars** (Панели инструментов). При установке или сбросе каждого флажка в окне будет появляться или исчезать соответствующая панель инструментов.

6. После установки нужной конфигурации нажмите кнопку **Close** (Закрыть).

Чтобы добавить или удалить кнопку в панели инструментов:

1. Установите требуемый набор панелей инструментов, как это описано выше.

2. Выберите команду меню **Tools | Customize** (Сервис | Настройка). Появится диалоговое окно **Customize** (Настройка).

3. Раскройте вкладку **Commands** (Команды). Диалоговое окно **Customize** (Настройка) примет вид, изображенный на рис. П2.5.

4. Поместите указатель мыши на кнопку панели инструментов, которую необходимо удалить.

5. Нажмите левую кнопку мыши и перетащите эту кнопку за пределы панели инструментов.

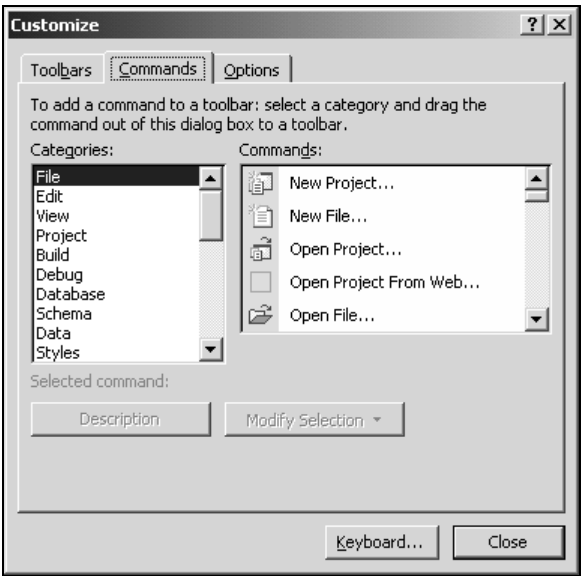


Рис. П2.5. Диалоговое окно **Customize**, вкладка **Commands**

6. Отпустите левую кнопку мыши. Данная кнопка будет удалена из панели инструментов.
7. Поместите указатель мыши на кнопку панели инструментов, которую необходимо переместить на другую панель инструментов.
8. Нажмите левую кнопку мыши и перетащите эту кнопку на другую панель инструментов.
9. Отпустите левую кнопку мыши. Данная кнопка переместится на другую панель инструментов.
10. В раскрывающемся списке **Categories** (Категории) выберите категорию, к которой относится новая кнопка, которую необходимо поместить на панель инструментов. (Категория, обычно, соответствует меню, в котором расположена данная команда, или панели инструментов, в которой уже расположена данная кнопка).
11. Нужная кнопка появится в окне списка **Commands** (Команды).
12. Поместите указатель мыши на значок новой кнопки в окне списка **Commands** (Команды).
13. Перетащите с помощью мыши эту кнопку на ту панель инструментов и в ту позицию панели инструментов, где она должна находиться.
14. Отпустите левую кнопку мыши. Данная кнопка появится на панели инструментов.
15. После установки нужной конфигурации нажмите кнопку **Close** (Закрыть).

Любая из панелей инструментов может быть сделана плавающей или зафиксирована на любой из границ рабочей области, как это показано на рис. П2.6.

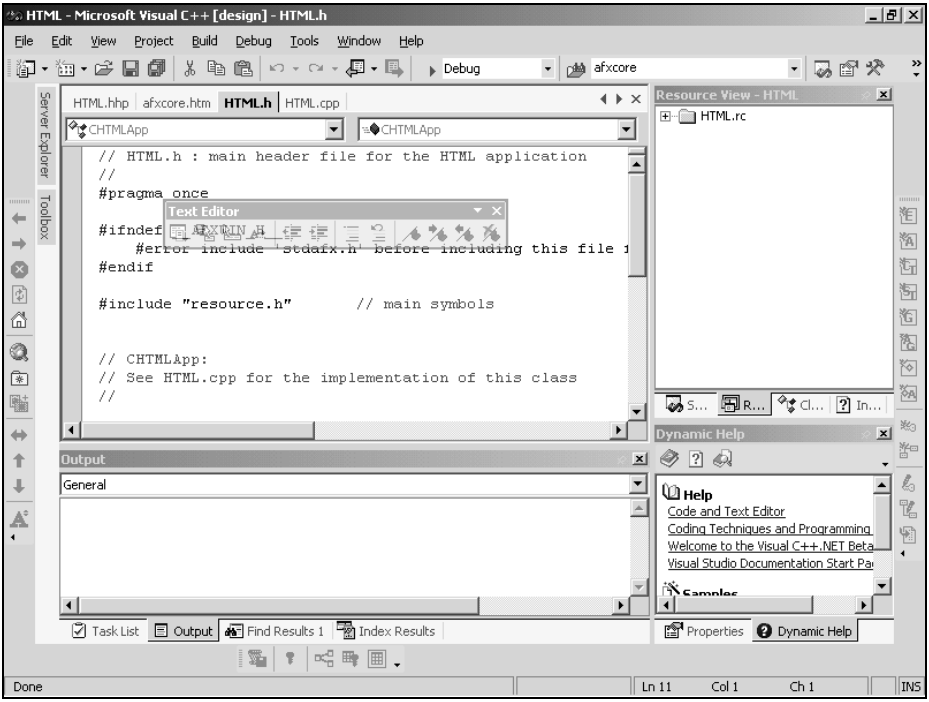


Рис. П2.6. Размещение панелей инструментов

Для перемещения зафиксированной панели инструментов необходимо поместить указатель мыши на ее *бордюр* (wrinkles), представляющий собой едва заметную штриховую полоску на одном из ее краев, и, нажав левую кнопку мыши, переместить ее в новое положение. Если панель инструментов находится далеко от границ рабочей области, она становится плавающей, т. е. превращается в обычное окно, как это показано на рис. П2.7 для панели инструментов **Text Editor** (Текстовый редактор). При приближении перемещаемой панели к границе рабочей области изменение формы ее рамки указывает на то, что она будет зафиксирована.

При установке зафиксированных панелей для устранения промежутков между ними рекомендуется "наезжать" устанавливаемой панелью инструментов на уже установленную. При этом Visual Studio сама позаботится об устранении перекрытия панелей и установит их встык.

Наиболее часто пользователю придется работать с панелью инструментов **Standard** (Стандартная). Ниже приведено ее краткое описание.

Панель инструментов *Standard*

Стандартная панель инструментов позволяет пользователю осуществлять вызов, редактирование и сохранение файлов. Кнопки данной панели инструментов, как правило, дублируются командами меню. В табл. П2.1 приведено соответствие кнопок панели инструментов **Standard** (Стандартная) командам меню.

Таблица П2.1. *Соответствие кнопок панели инструментов* ***Standard*** *командам меню*

Кнопка панели инструментов	Команда меню
New Project	File | New | Project
Add New Item	Project | Add New Item
Open File	File | Open
Save	File | Save
Save All	File | Save All
Cut	Edit | Cut
Copy	Edit | Copy
Paste	Edit | Paste
Undo	Edit | Undo
Redo	Edit | Redo
Navigate Backward	View | Navigate Backward
Navigate Forward	View | Navigate Forward
Start	Debug | Start
Find in Files	Edit | Find and Replace | Find in Files
Solution Explorer	View | Solution Explorer
Properties Window	View | Properties Window
Toolbox	View | Toolbox
Class View	View | Class View

Следует обратить внимание на то, что кнопки **New Project** (Новый проект), **Add New Item** (Добавить новый элемент) и **Class View** (Просмотр классов) представляют собой раскрывающиеся меню, первая команда которых соответствует их названию и сопоставленной им команде основного меню, а остальные команды позволяют производить некоторые дополнительные операции.

Кнопка **Navigate Backward** (Возвратиться назад) также представляет собой раскрывающееся меню, в котором перечислены последние операции перемещения курсора в файлах проекта, что позволяет непосредственно выбрать отменяемую операцию. Если текущее положение курсора было установлено при переключении из другого окна, то при нажатии кнопки **Navigate Backward** (Возвратиться назад) будет осуществлен переход в это окно.

Помимо кнопок панель инструментов **Standard** (Стандартная) содержит два раскрывающихся списка. Раскрывающийся список **Solution Configuration** (Конфигурация решения) позволяет установить конфигурацию создаваемого приложения (отладочную или распространяемую), а раскрывающийся список **Find** (Найти) позволяет задать искомый фрагмент текста при использовании команды меню **Edit | Find and Replace | Find** (Правка | Найти и заменить | Найти).

Система меню

Среда разработки Visual C++ имеет достаточно сложную систему меню. Некоторые команды находятся на третьем или, даже, четвертом уровне иерархии. Обычно этим командам соответствует некоторая комбинация клавиш, позволяющая вызвать их значительно быстрее, чем с использованием меню. Однако у пользователя Visual C++ есть более важные дела, чем запоминать несколько десятков комбинаций клавиш, бóльшую часть из которых он будет использовать не чаще, чем раз в месяц.

Однако разработчики Visual C++ были убеждены в обратном и установили такое количество комбинаций клавиш, что практически любое ошибочное нажатие клавиш вызывает какое-нибудь диалоговое окно, что не дает пользователям данной среды программирования забыть хорошую русскую пословицу: "Заставь дурака Богу молиться, он и лоб разобьет."

Панель меню Visual C++ содержит девять заголовков меню, которые перечислены ниже:

- **File** (Файл) — меню операций, связанных с файлами, включая их создание, открытие, закрытие и печать;
- **Edit** (Правка) — меню операций редактирования, включая копирование, удаление, восстановление измененной информации, поиск и перемещение по файлу;
- **View** (Вид) — меню управления способом отображения информации на экране, включая управление отображением панелей инструментов и окон среды разработки **Output** (Окно вывода) и **Workspace** (Рабочая область);
- **Project** (Проект) — меню команд управления разрабатываемым проектом в целом;
- **Build** (Создать) — меню команд компиляции и компоновки приложения;
- **Debug** (Отладка) — меню команд отладки приложения и запуска его на исполнение;
- **Tools** (Сервис) — меню команд настройки Visual C++ и доступа к автономным утилитам;
- **Window** (Окно) — меню выбора окон и их автоматического упорядочения на экране;
- **Help** (Справка) — меню команд обращения к справочной системе Visual C++.

Ниже будут рассмотрены все перечисленные выше меню и описаны их основные команды.

Меню *File*

Меню **File** (Файл), приведенное на рис. П2.7, содержит команды, осуществляющие операции, связанные с файлами, включая их создание, открытие, закрытие и печать.

Команда *File | New*

Данная команда не является терминальной: при ее выборе появляется контекстное меню, содержащее три команды, как это показано на рис. П2.8.

При выборе команды меню **File | New | Project** (Файл | Создать | Проект) появляется диалоговое окно **New Project** (Новый проект), изображенное на рис. П2.9.

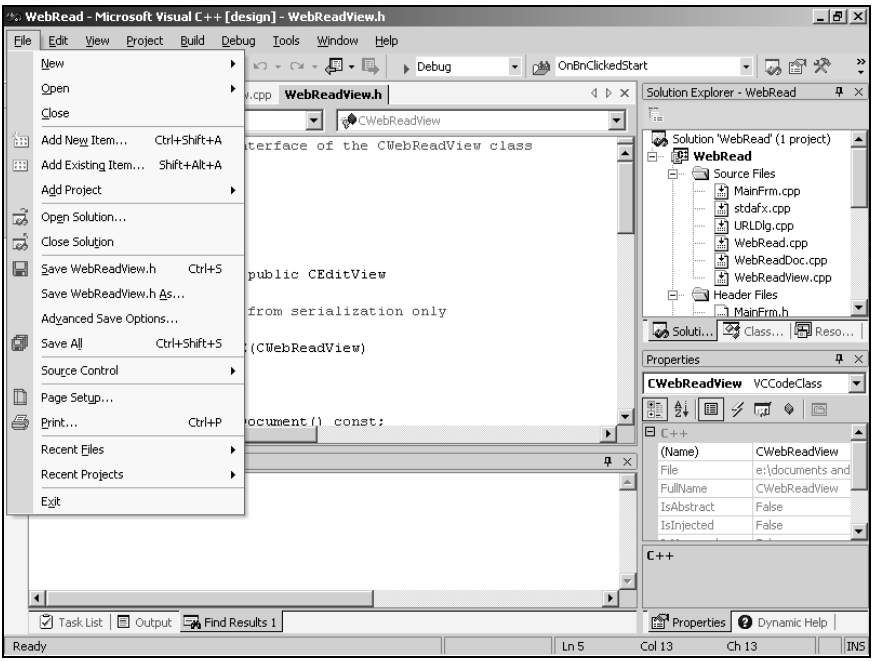


Рис. П2.7. Меню **File**

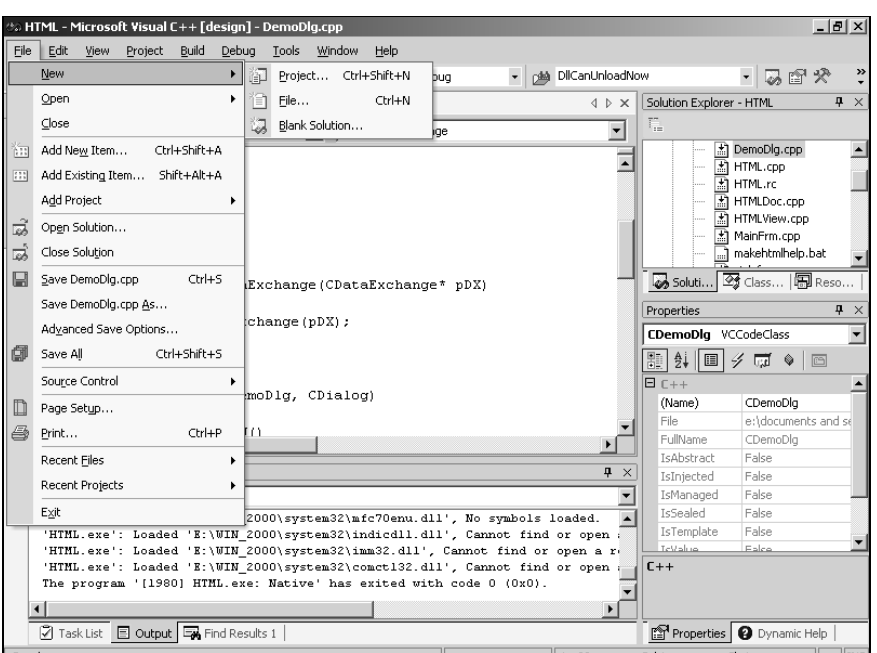


Рис. П2.8. Контекстное меню **New**

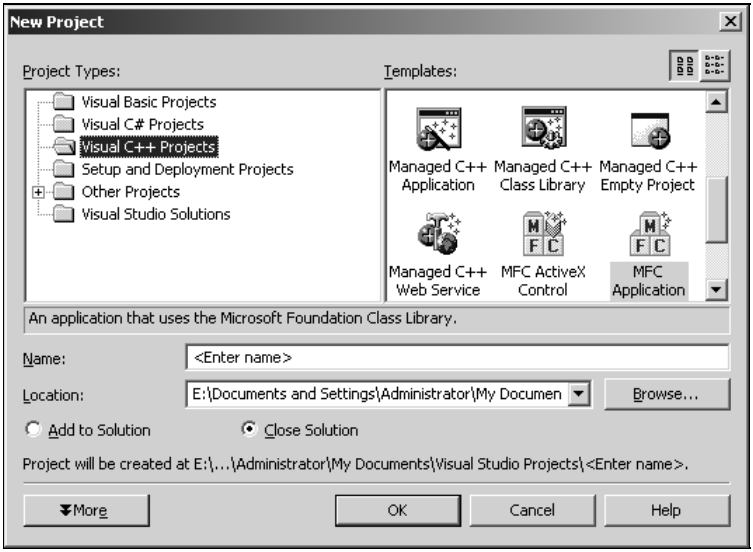


Рис. П2.9. Диалоговое окно **New Project**

Это диалоговое окно используется для создания новых проектов. Его использование подробно описано в *главе 1*. Вместо данной команды меню можно использовать комбинацию клавиш <Ctrl>+<Shift>+<N>.

При выборе команды меню **File | New | File** (Файл | Создать | Файл). Появляется диалоговое окно **New File** (Новый файл), изображенное на рис. П2.10.

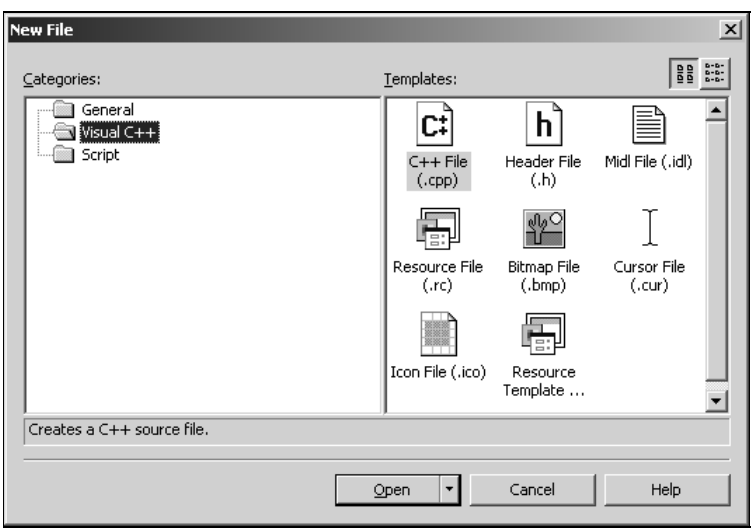


Рис. П2.10. Диалоговое окно **New File**

Это диалоговое окно используется для создания в проекте пустых файлов программ, описаний рабочей среды или других документов. Вместо данной команды меню можно использовать комбинацию клавиш <Ctrl>+<N>.

При выборе команды меню **File | New | Blank Solution** (Файл | Создать | Пустое приложение) появляется диалоговое окно **New Project** (Новый проект). В окне иерархического списка **Project Types** (Типы проектов) данного диалогового окна открыта папка **Visual Studio Solutions** (Решения Visual Studio), позволяющая создавать пустые решения Visual Studio.NET.

Команда *File | Open*

Данная команда меню также, не является терминальной, а при ее выборе появляется контекстное меню, содержащее четыре команды.

При выборе команды меню **File | Open | Project** (Файл | Открыть | Проект), появляется диалоговое окно **Open Project** (Открыть проект), изображенное на рис. П2.11.

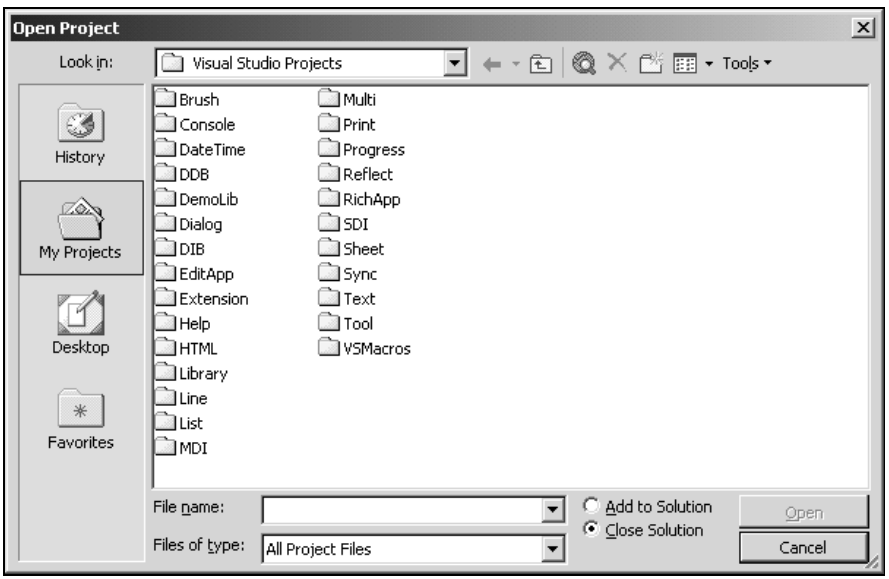


Рис. П2.11. Диалоговое окно **Open Project**

Данное диалоговое окно содержит переключатель, установленный по умолчанию в положение **Close Solution** (Закрыть решение). При таком положении переключателя при открытии нового проекта старое решение (solution) закрывается и проект размещается в новом решении. При переключателе, установленном в положение **Add to Solution** (Добавить к решению) новый проект помещается в старое решение. Вместо данной команды меню можно использовать комбинацию клавиш <Ctrl>+<Shift>+<O>.

При выборе команды меню **File | Open | Project From Web** (Файл | Открыть | Проект в Сети) появляется диалоговое окно **Open Project From Web** (Открыть проект в Сети), изображенного на рис. П2.12.

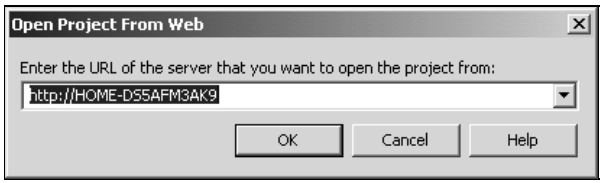


Рис. П2.12. Диалоговое окно **Open Project From Web**

Это диалоговое окно содержит раскрывающийся список, в текстовом поле которого следует указать адрес сервера, на котором нужно открыть проект.

При выборе команды меню **File | Open | File** (Файл | Открыть | Файл). Появляется диалоговое окно **Open File** (Открыть файл), изображенное на рис. П2.13.

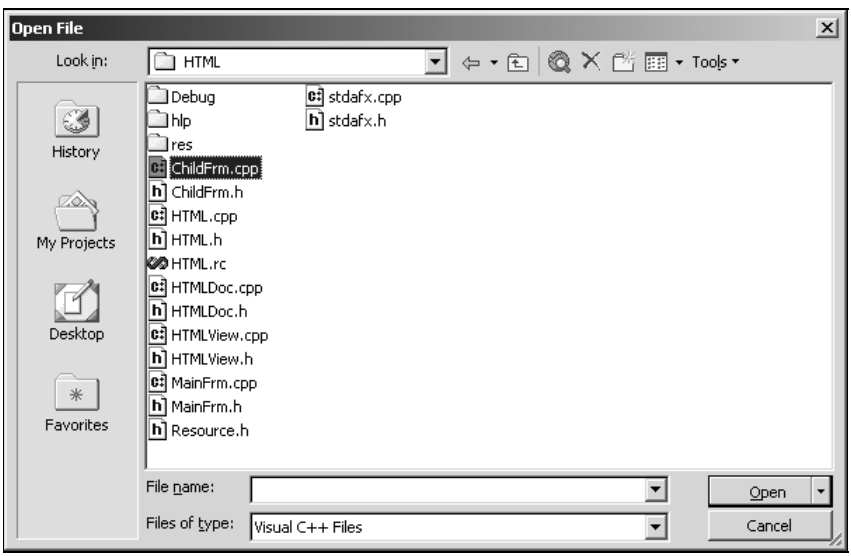


Рис. П2.13. Диалоговое окно **Open File**

Это диалоговое окно используется для включения в проект файлов программ, описаний рабочей среды или других документов и представляет собой стандартное диалоговое окно Windows **Open File** (Открыть), используемое практически во всех приложениях Windows. В раскрывающемся списке **Files of type** (Типы файлов) указываются типы файлов, отображаемых в окне списка данного диалогового окна. Если вам необходимо открыть файл, не включенный в данный список, раскройте этот список и выберите нужный вам тип файлов. При этом произойдет изменение содержимого окна списка данного диалогового окна.

Раскрывающийся список кнопки **Open** (Открыть) позволяет изменить режим открытия файла. Чтобы выбрать режим открытия файла, необходимо выбрать в этом раскрывающемся списке команду **Open With** (Открыть с помощью). Появится диалоговое окно **Open With - HTML.rc** (Открыть как), изображенное на рис. П2.14.

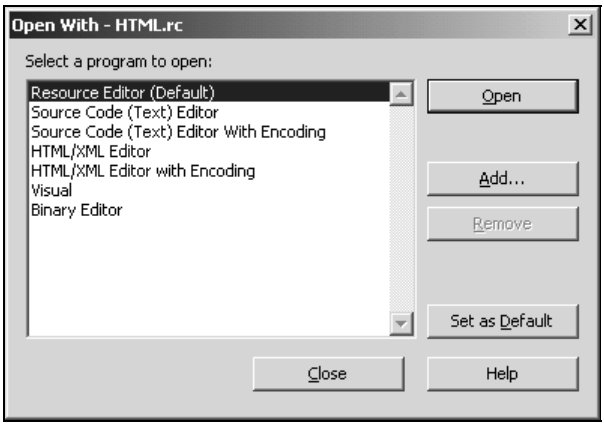


Рис. П2.14. Диалоговое окно **Open With**

Практически единственным случаем, когда следует воспользоваться данным окном, является открытие файла ресурсов в текстовом режиме. В этом случае в окне списка **Select a program to open** (Выберите программу для работы с файлом) следует выделить строку **Source Code (Text) Editor** (Текстовый редактор исходного кода).

Вместо данной команды меню можно использовать комбинацию клавиш <Ctrl>+<O>.

При выборе команды меню **File | Open | File From Web** (Файл | Открыть | Файл в Сети) появляется диалоговое окно **Open File From Web** (Открыть файл в Сети), аналогичное окну **Open Project From Web** (Открыть проект в Сети) изображенному на рис. П2.12.

Команда *File | Close*

Выбор данной команды меню приводит к закрытию файла, окно которого имеет на данный момент фокус ввода. Если таких файлов нет, то данная команда становится недоступной. Другим способом закрыть файл и связанное с ним окно является нажатие на кнопку **Закрыть** (кнопка с косым крестиком, расположенная в крайней правой позиции заголовка окна).

Команды *File | Add New Item* и *File | Add Existing Item*

Команды меню **File | Add New Item** (Файл | Добавить новый элемент) и **File | Add Existing Item** (Файл | Добавить существующий элемент) полностью дублируют одноименные команды меню **Project** (Проект), где они и будут рассмотрены.

Команда *File | Add Project*

Эта команда меню не является терминальной, а при ее выборе появляется контекстное меню, содержащее три команды. Команда **File | Add Project | New Project** (Файл | Добавить проект | Новый проект) аналогична описанной выше команде **File | New | Project** (Файл | Создать | Проект), команда **File | Add Project | Existing Project** (Файл | Добавить проект | Существующий проект) аналогична описанной выше команде

File | Open | Project (Файл | Открыть | Проект), а команда **File | Add Project | Existing Project From Web** (Файл | Добавить проект | Существующий проект в Сети) аналогична описанной выше команде **File | Open | Project From Web** (Файл | Открыть | Проект в Сети). Однако в обоих случаях проект будет включен в текущее решение. Поэтому диалоговое окно **Add Existing Project** (Добавить существующий проект), в отличие от окна **Open Project** (Открыть проект), не содержит переключателя, позволяющего закрыть текущее приложение.

Команда *File | Open Solution*

Выбор данной команды меню позволяет открыть существующее решение. При этом появится диалоговое окно **Open Solution** (Открыть решение), изображенное на рис. П2.15.

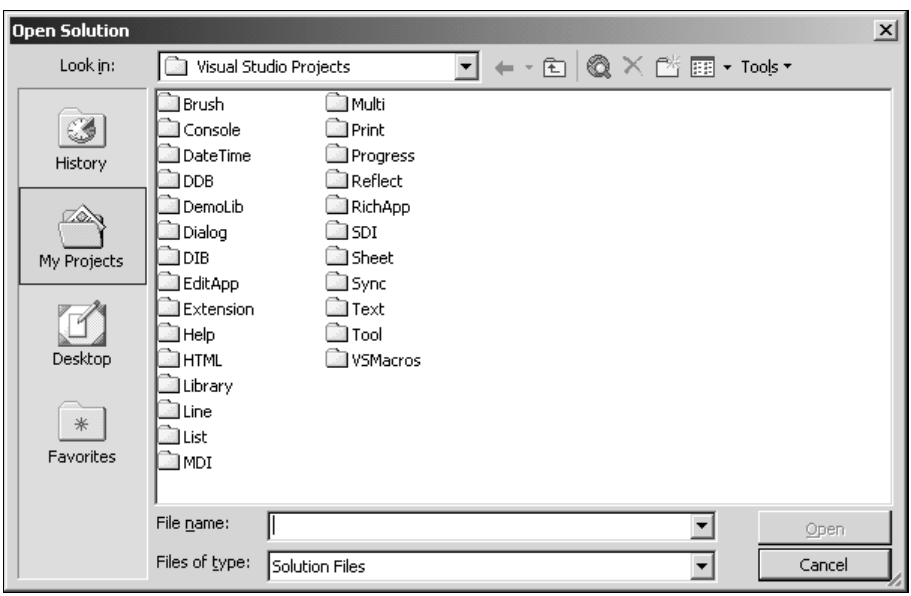


Рис. П2.15. Диалоговое окно **Open Solution**

Чтобы открыть новое решение: раскройте папку, содержащую нужное решение, выделите файл решения и нажмите кнопку **Open** (Открыть). Если текущее решение содержит несохраненную информацию, на дисплее появится соответствующее окно сообщения. После этого в среду программирования Visual C++ будет загружено новое решение.

Команда *File | Close Solution*

Данная команда меню закрывает текущее решение. Если текущий проект содержал несохраненную информацию, на дисплее появится окно сообщения, аналогичное изображенному на рис. П2.16.

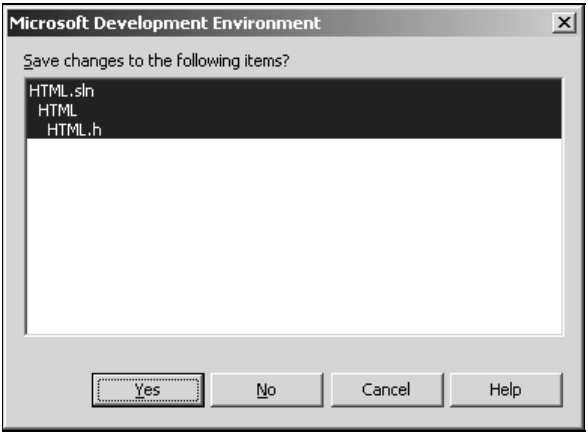


Рис. П2.16. Окно сообщения о несохраненных файлах

Команда *File | Save*

Выбор данной команды меню позволяет сохранить в файле изменения, внесенные в окно, которое имеет на настоящий момент фокус ввода. Если с окном, имеющим в настоящий момент фокус ввода, не связан никакой файл, то вместо данной команды автоматически вызывается команда **File | Save As** (Файл | Сохранить как). Если в данный момент ни одно из окон редактирования файлов не имеет фокус ввода, то данная команда меню и соответствующая ей кнопка панели инструментов становятся недоступными.

Команда *File | Save As*

Выбор данной команды меню позволяет сохранить изменения, внесенные в окно, которое имеет на настоящий момент фокус ввода, в новом файле, сохранив содержимое файла, с которым было связано данное окно, без изменений. При выборе этой команды появляется диалоговое окно **Save File As** (Сохранить файл как), изображенное на рис.П2.17.

Если в данный момент ни одно из окон редактирования файлов не имеет фокуса ввода, то данная команда меню и соответствующая ей кнопка панели инструментов становятся недоступными.

Команда *File | Advanced Save Options*

При выборе данной команды меню появляется диалоговое окно **Advanced Save Options** (Выбор дополнительных опций сохранения), изображенное на рис. П2.18.

В раскрывающемся списке **Encoding** (Кодирование) данного диалогового окна можно выбрать кодировку, используемую при сохранении данного файла, а в раскрывающемся списке **Line endings** (Добавление в конец строки) — набор символов, помещаемый в конец каждой строки текста (зависит от используемой операционной системы).

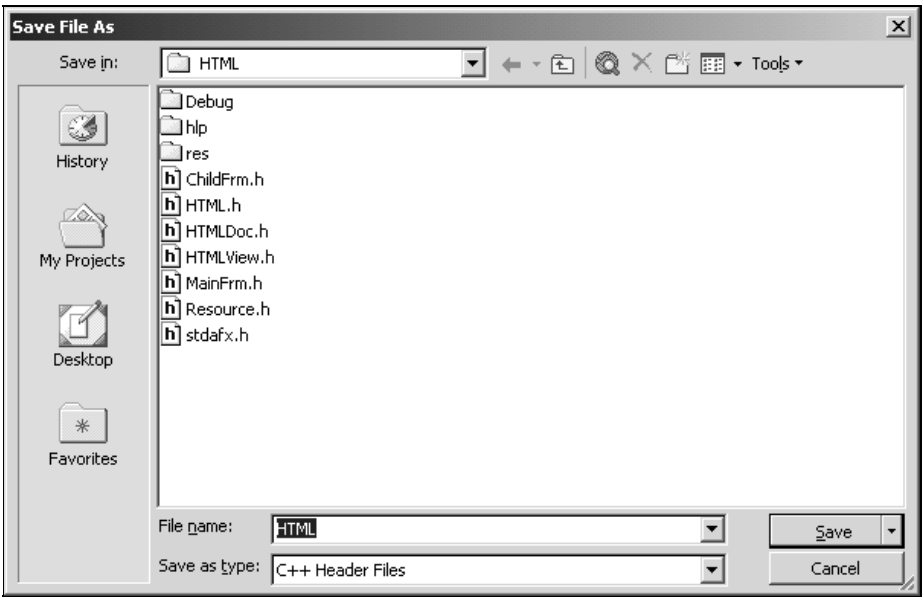


Рис. П2.17. Диалоговое окно **Save File As**

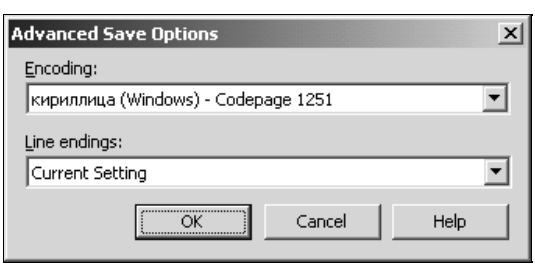


Рис. П2.18. Диалоговое окно **Advanced Save Options**

Команда *File | Save All*

Выбор данной команды меню позволяет сохранить изменения, внесенные во все открытые файлы данного проекта. Рекомендуется выбирать эту команду каждые 10—15 минут, если содержимое этих файлов не сохранялось другим способом. По умолчанию все файлы проекта сохраняются при его компиляции.

Команда *File | Source Control*

Данная команда меню используется для контроля исходного кода приложения. Контроль исходного кода приложения применяется при совместной разработке прило-

жения несколькими разработчиками. Поскольку в данной книге этот вопрос не рассматривался, то и эта команда описываться не будет.

Команда *File | Page Setup*

Данная команда меню выводит на экран диалоговое окно **Page Setup** (Параметры страницы), изображенное на рис. П2.19.

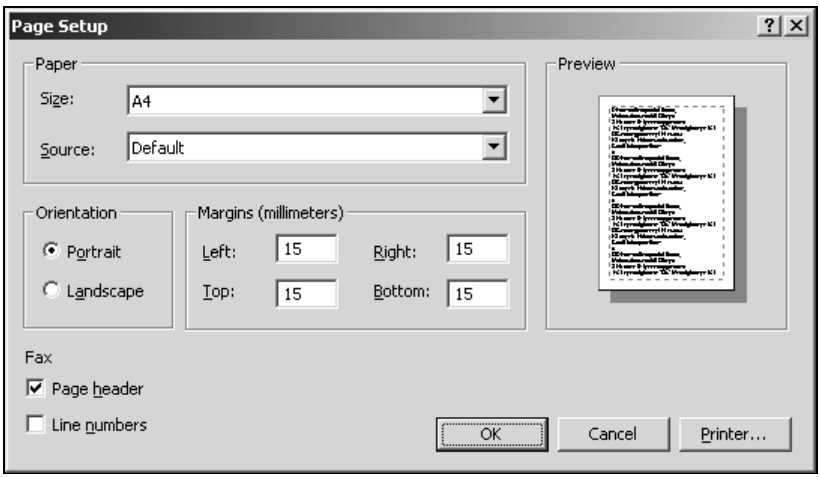


Рис. П2.19. Диалоговое окно **Page Setup**

Это диалоговое окно позволяет задавать размер листа бумаги, на котором будет производиться распечатка, источник бумаги, размещение текста на нем и отступы текста от краев. Результат установок можно оценить в специальном окне предварительного просмотра текста.

Нажатие на кнопку **Printer** (Принтер) позволит выбрать принтер, на котором будет произведена распечатка. В данном случае распечатка будет послана по факсу.

Команда *File | Print*

Выбор данной команды меню приводит к инициализации процесса печати содержимого окна, которое на данный момент имеет фокус ввода. Если в данный момент ни одно из окон редактирования файлов не имеет фокус ввода, то данная команда становится недоступной.

Непосредственно после выбора данной команды меню появляется диалоговое окно **Print** (Печать), изображенное на рис. П2.20.

В раскрывающемся списке **Name** (Имя) может быть выбран принтер, на котором будет произведена распечатка (если в системе установлен более чем один принтер).

Группа переключателей **Print range** (Диапазон печати) позволяет определить диапазон печати в том случае, если в активном окне выделен блок. В этом случае при

установке переключателя в положение **All** (Все) производится печать всего файла. При установке данного переключателя в положение **Selection** (Выбор) печатается только выделенный текст.

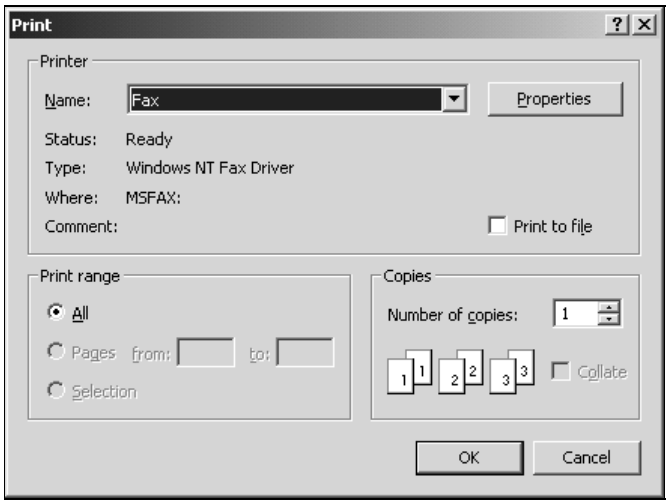


Рис. П2.20. Диалоговое окно **Print**

Нажатие кнопки **Properties** (Свойства) вызывает стандартное диалоговое окно **Properties** (Свойства), позволяющее настроить используемый для печати принтер. Для печати документа нажмите кнопку **OK**.

Команда *File | Recent Files*

При выборе данной команды меню появляется контекстное меню, содержащее список последних закрытых файлов. Выбор имени файла в этом меню позволяет открыть его окно редактирования.

Команда *File | Recent Projects*

При выборе этой команды меню появляется контекстное меню, содержащее список последних открытых в данной системе программирования проектов (хотя, судя по расширениям, здесь речь идет не о проектах, а о решениях). Под номером **1** всегда стоит активный проект, если он содержался в данном списке до своего открытия. В противном случае под номером **1** стоит предыдущий активный проект. Выбор имени проекта в этом меню позволяет открыть его для работы вместо активного в данный момент проекта.

Команда *File | Exit*

Выбор этой команды меню завершает работу в среде программирования Visual C++.

Меню *Edit*

Меню **Edit** (Правка), приведенное на рис. П2.21, содержит команды, осуществляющие операции редактирования, включая копирование, удаление, восстановление измененной информации, поиск и перемещение по файлу.

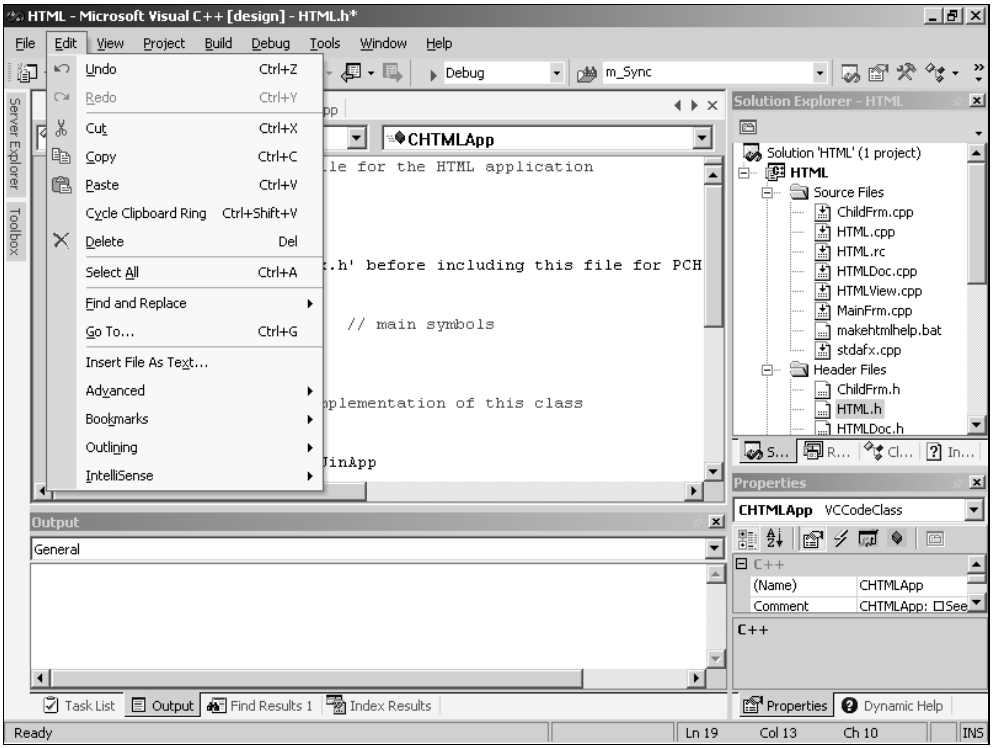


Рис. П2.21. Меню **Edit**

Команда *Edit | Undo*

Выбор данной команды меню отменяет операции редактирования, выполненные ранее. Большая часть операций, таких как редактирование и удаление текста, могут отменяться. Если эта команда меню недоступна, это означает, что или пользователь не произвел ни одной операции редактирования с момента последнего сохранения файла, или все операции, которые могут быть отменены, уже отменены, или что последняя операция редактирования не может быть отменена.

В панели инструментов **Standard** (Стандартная) около кнопки **Undo** (Отменить) имеется треугольная кнопка, раскрывающая стек операций (список в обратном порядке от самых свежих до самых поздних операций редактирования, которые могут быть отменены). Раскрытый стек операций показан на рис. П2.22. При отмене операций с использованием стека необходимо выделить последнюю отменяемую операцию в

списке (расположенные выше операции отмечаются автоматически). Все операции, расположенные в стеке выше выделенной операции и сама выделенная операция, будут отменены.

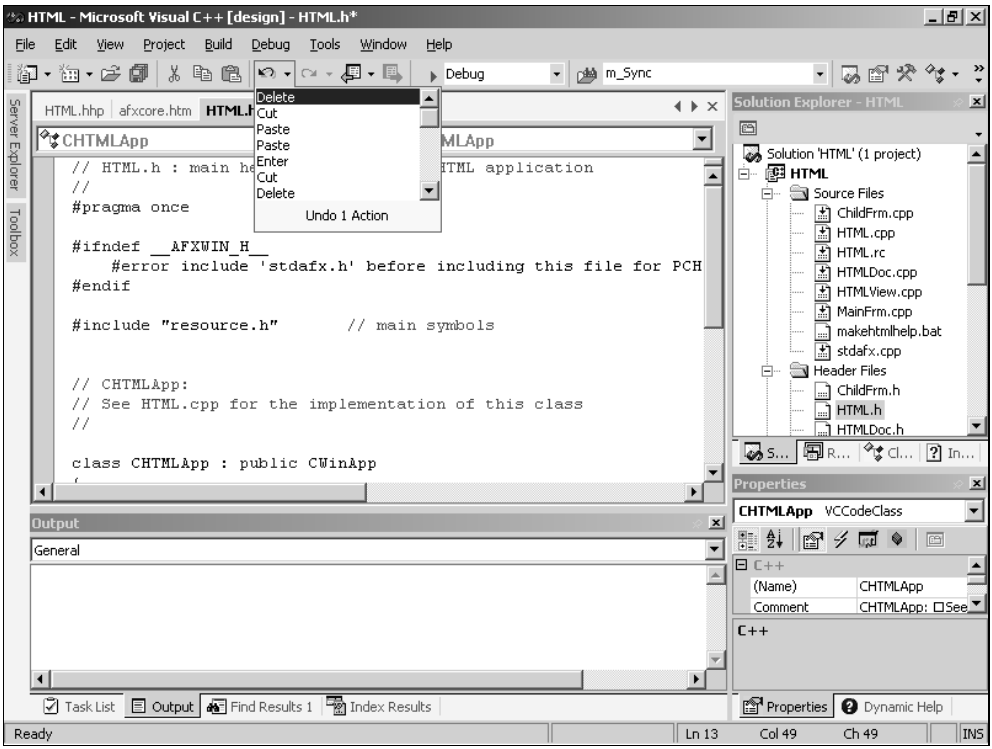


Рис. П2.22. Раскрытый стек операций

Команда *Edit | Redo*

Как только пользователь отменил некоторую операцию с использованием команды **Undo** (Отменить), становится доступной команда **Redo** (Вернуть). Эта команда служит для отмены действия команды **Undo** (Отменить). Около кнопки **Redo** (Вернуть) на панели инструментов **Standard** (Стандартная) также имеется треугольная кнопка, раскрывающая стек операций, в который записаны все операции, отмененные командой **Undo** (Отменить).

Команда *Edit | Cut*

Выбор данной команды меню приводит к копированию содержимого выделенного фрагмента в буфер обмена (Clipboard) и удаления этого фрагмента из исходного текста. Если в данный момент в активном окне отсутствует выделенный фрагмент, то эта команда меню и соответствующая ей кнопка панели инструментов становятся недоступными.

Команда *Edit | Copy*

Выбор данной команды меню приводит к копированию содержимого выделенного фрагмента в буфер обмена (Clipboard). В исходном файле не производится никаких изменений. Если в данный момент в активном окне отсутствует выделенный фрагмент, то эта команда меню и соответствующая ей кнопка панели инструментов становятся недоступными.

Команда *Edit | Paste*

Выбор данной команды меню приводит к копированию содержимого буфера обмена (Clipboard) в окно, которому принадлежит в настоящее время фокус ввода в позицию, определяемую текстовым курсором. Если в окне имеется выделенный фрагмент, то его содержимое заменяется содержимым буфера обмена и выделение снимается. Если в данный момент буфер обмена не содержит никакой информации, то эта команда меню и соответствующая ей кнопка панели инструментов становятся недоступными.

Команда *Edit | Delete*

Выбор команды меню **Edit | Delete** удаляет выделенный текст или элемент. Если операция удаления восстановима, то эта операция добавляется в стек операций кнопки **Undo** (Отменить) на панели инструментов **Standard** (Стандартная).

Команда *Edit | Select All*

Выбор этой команды меню выделяет весь текст файла, связанного с окном, имеющим в настоящее время фокус ввода. В окне редактирования текста программы выделяется весь текст содержащегося в нем файла. В окне редактирования ресурса диалогового окна выделяются все элементы управления данного диалогового окна.

Для выделения элементов управления диалогового окна можно, удерживая нажатой клавишу <Ctrl>, щелкнуть левой кнопкой мыши по всем элементам управления, которые нужно выделить, или выбрать команду меню **Edit | Select All** (Правка | Выделить все) и, удерживая нажатой клавишу <Ctrl>, щелкнуть левой кнопкой мыши по всем элементам управления, которые нужно исключить из выделения.

Команда *Edit | Find and Replace*

Данная команда меню не является терминальной, а при ее выборе появляется контекстное меню, содержащее команды поиска и замены. Это контекстное меню показано на рис. П2.23.

При выборе команды **Edit | Find and Replace | Find** (Правка | Найти и заменить | Найти) или нажатии комбинации клавиш <Ctrl>+<F> появляется диалоговое окно **Find** (Найти), изображенное на рис. П2.24.

Это диалоговое окно используется для поиска текста в проекте. Поиск может быть произведен:

- только в текущем документе (положение переключателя **Current document** (Текущий документ));

- во всех открытых документах (положение переключателя **All open documents** (Все открытые документы));
- только в выделенном фрагменте (положение переключателя **Selection only** (Только выбранное));
- в некоторой области, определяемой контекстом (классе, функции и т. д.) (положение переключателя **Only** (Только)).

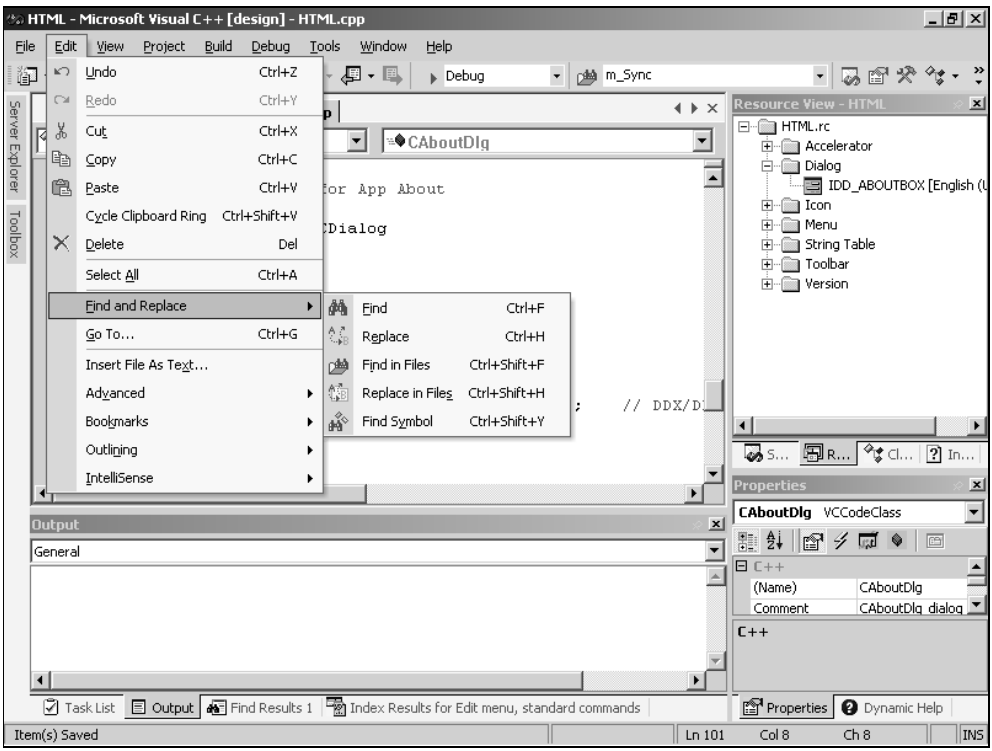


Рис. П2.23. Контекстное меню **Find and Replace**

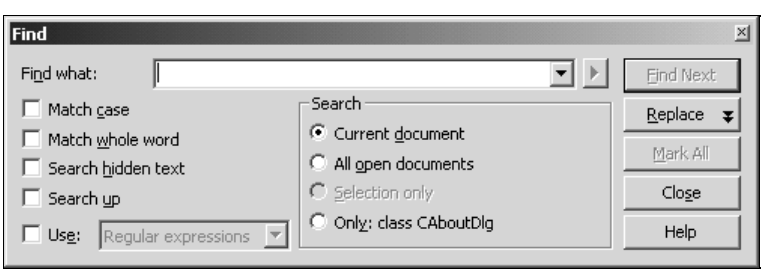


Рис. П2.24. Диалоговое окно **Find**

Искомый текст помещается в текстовое поле раскрывающегося списка **Find what** (Найти). В данном раскрывающемся списке хранятся фрагменты текста, по которым производился поиск в последнее время. Если искомый фрагмент текста содержится в раскрывающемся списке, его можно выбрать, а не вводить заново.

В диалоговом окне **Find** (Найти) могут быть установлены следующие флажки:

- **Match case** (Учитывать регистр) — при сравнении учитывается регистр символов, используемый при написании текста в текстовом поле раскрывающегося списка **Find what** (Найти). Если этот флажок сброшен, то при сравнении регистр символов не учитывается, что удобно при поиске идентификаторов, содержащих заглавные буквы в середине слова;
- **Match whole word** (Только слово целиком) — устанавливает режим, при котором текст в текстовом поле раскрывающегося списка **Find what** (Найти) рассматривается при сравнении как целое слово и при поиске будут пропущены слова, в которых данный текст является составной частью другого слова;
- **Search hidden text** (Поиск скрытого текста) — производится поиск скрытого текста;
- **Search up** (Поиск к началу) — поиск производится по направлению к началу текста;
- **Use** (Использовать) — делает доступным связанный с ним раскрывающийся список. При выделении в этом списке строки **Regular expressions** (Регулярные выражения) в процессе поиска содержимое текстового поля раскрывающегося списка **Find what** (Найти) рассматривается как регулярное выражение. Под регулярным выражением понимается некоторый текст, включающий в себя специальные символы, такие как конец строки, любое число или другие символы. Для ввода специального символа достаточно нажать на кнопку со стрелкой, расположенную справа от текстового поля раскрывающегося списка **Find what** (Найти), и выбрать соответствующую команду в появившемся контекстном меню, изображенном на рис. П2.25.

  При выделении в этом списке строки **Wildcards** (Подстановочные знаки) некоторые символы и последовательности символов будут рассматриваться как указатели на класс символов или последовательность символов. Так, например, вопросительный знак (?) будет обозначать любой символ, а звездочка (*) будет обозначать один или несколько символов.

Обычно при работе с диалоговым окном **Find** (Найти) в текстовое поле раскрывающегося списка **Find what** (Найти) вводится некоторый текст и нажимается кнопка **Find Next** (Найти следующий). Диалоговое окно **Find** (Найти) закрывается и в окне редактирования файла выделяется следующее вхождение данного текста. Для поиска нужной строки в тексте нужно последовательно открывать диалоговое окно **Find** (Найти) и нажимать в нем кнопку **Find Next** (Найти следующий). Эту процедуру нельзя назвать очень удобной.

Другой возможностью осуществить поиск текста в файле является использование кнопки **Mark All** (Закладки). В этом случае у каждой строки, содержащей искомый текст, будет поставлена закладка. После этого можно просмотреть все строки, нажимая клавишу <F2>.

После первого же поиска, осуществленного с использованием диалогового окна **Find** (Найти), в текстовом поле раскрывающегося списка **Find** (Найти), расположенного

на панели инструментов **Standard** (Стандартная), появится текст, введенный в текстовое поле раскрывающегося списка **Find what** (Найти). При выделении данного текста и нажатии клавиши <Enter> будет производиться поиск в направлении к концу файла.

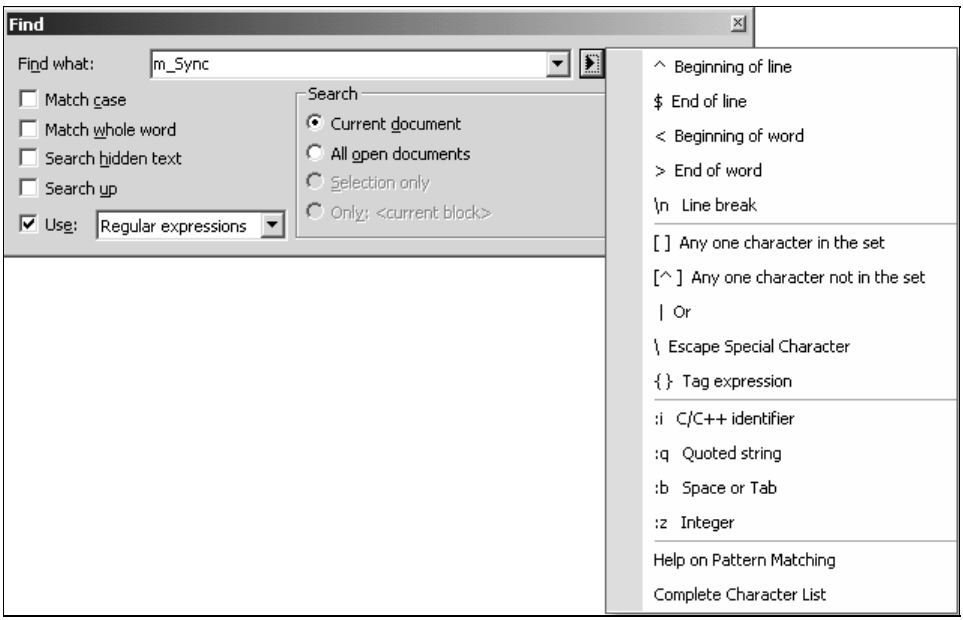


Рис. П2.25. Контекстное меню для ввода специальных символов

При выборе команды **Edit | Find and Replace | Replace** (Правка | Найти и заменить | Заменить) или нажатии комбинации клавиш <Ctrl>+<H> появляется диалоговое окно **Replace** (Заменить), изображенное на рис. П2.26.

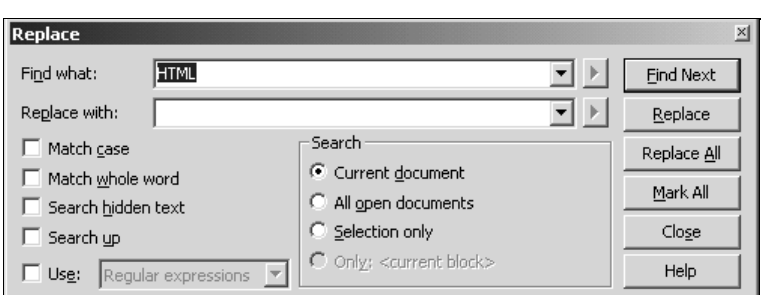


Рис. П2.26. Диалоговое окно **Replace**

Это диалоговое окно, во многом аналогичное диалоговому окну **Find** (Найти), используется для замены найденного текста новым. Применение текстового поля рас-

крывающегося списка **Find what** (Найти) и флажков **Match case** (Учитывать регистр), **Match whole word** (Только слово целиком), **Search hidden text** (Поиск скрытого текста), **Search up** (Поиск по направлению к началу текста), **Use** (Использовать) и связанного с ним раскрывающегося списка, а также переключателей группы **Search** (Поиск) в диалоговом окне **Replace** (Заменить) ничем не отличается от использования этих элементов управления в диалоговом окне **Find** (Найти). Основным преимуществом диалогового окна **Replace** (Заменить) перед диалоговым окном **Find** (Найти) является то, что при нажатии кнопки **Find Next** (Найти далее) данное окно не закрывается, что позволяет существенно упростить процедуру поиска информации в файле.

В текстовом поле раскрывающегося списка **Replace with** (Заменить на) помещается текст, на который следует заменить искомый текст. Так же, как и в случае с раскрывающимся списком **Find what** (Найти), в данном раскрывающемся списке хранятся фрагменты текста, использовавшиеся в предыдущих заменах. Если требуемый текст содержится в раскрывающемся списке, то его можно выбрать из него, а не вводить вручную. Текст в текстовом поле раскрывающегося списка **Replace with** (Заменить на) может представлять собой регулярное выражение.

После того как в тексте файла с использованием кнопки **Find Next** (Найти далее) будет найден требуемый фрагмент текста, пользователь может нажать кнопку **Replace** (Заменить). В результате найденный фрагмент текста будет заменен и будет произведен поиск следующего фрагмента текста. Если пользователь нажал кнопку **Replace All** (Заменить все), операция поиска и замены текста по всему файлу будет произведена автоматически без участия пользователя. Кнопку **Replace All** (Заменить все) следует использовать с особой осторожностью, поскольку при неправильном выборе установок в окне результаты быстрой замены придется исправлять в течение нескольких часов, поскольку стек операций команды **Undo** (Отменить) имеет ограниченные размеры. Если вы все-таки решились использовать эту кнопку, то настоятельно рекомендуется, по крайней мере, для первых замен использовать кнопку **Replace** (Заменить) и убедиться, что при замене не возникает никаких проблем.

При выборе команды **Edit | Find and Replace | Find in Files** (Правка | Найти и заменить | Найти в файлах) или нажатии комбинации клавиш <Ctrl>+<Shift>+<F> появляется диалоговое окно **Find in Files** (Найти в файлах), изображенное на рис. П2.27.

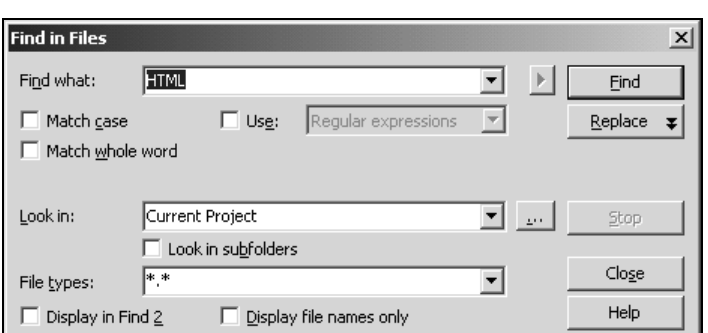


Рис. П2.27. Диалоговое окно **Find in Files**

Это диалоговое окно используется для поиска текста во всех файлах проекта, расширения которых указаны в текстовом поле раскрывающегося списка **File types** (Типы файлов). Так же, как и в диалоговом окне **Find** (Найти), искомый текст помещается в текстовое поле раскрывающегося списка **Find what** (Найти). Работа с ним аналогична работе с данным раскрывающимся списком в диалоговом окне **Find** (Найти).

В текстовом поле раскрывающегося списка **Look in** (Искать в) указывается область поиска. Это может быть текущий документ, выделенный фрагмент, текущий проект и другие области. Область поиска может быть настроена при нажатии на кнопку с тремя точками, расположенную справа от данного раскрывающегося списка. При нажатии на эту кнопку появляется диалоговое окно **Look In** (Искать в), изображенное на рис. П2.28.

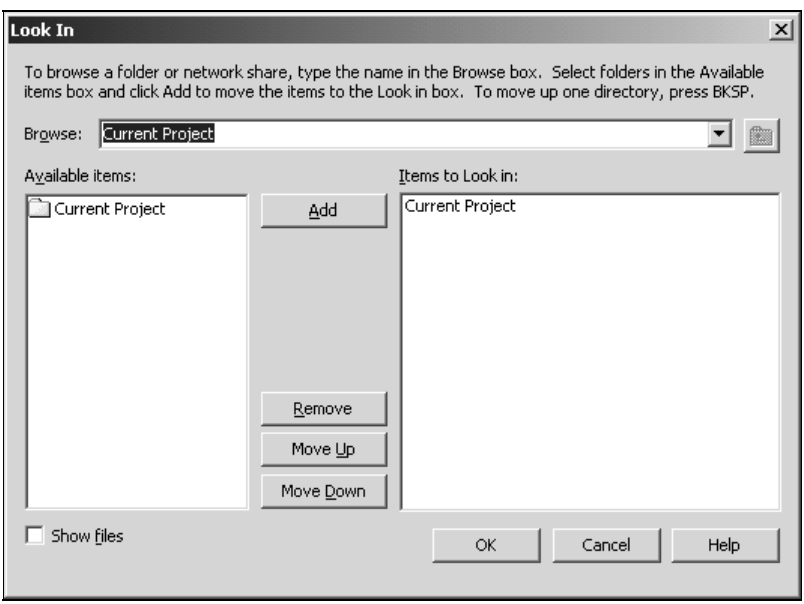


Рис. П2.28. Диалоговое окно **Look In**

В диалоговом окне **Find In Files** (Найти в файлах) могут быть установлены флажки **Match case** (Учитывать регистр), **Match whole word** (Только слово целиком) и **Use** (Использовать), назначение которых совпадает с назначением одноименных флажков в диалоговых окнах **Find** (Найти) и **Replace** (Заменить). Кроме того, в данном диалоговом окне могут быть установлены следующие флажки:

- **Display in Find 2** (Отобразить в найденном 2) — собранная информация отображается в окне **Find Results 2** (Результаты поиска 2);
- **Display file names only** (Отображать только имя файла) — в окне **Find Results** (Результаты поиска) будут выводиться только имена файлов, в которых найден указанный текст (то есть не будет выводиться номер и содержимое строки, в которой найден этот текст).

После ввода всей необходимой информации и нажатия кнопки **Find** (Найти) среда программирования Visual C++ производит поиск в файлах данного проекта и выводит его результаты в окно **Find Results** (Результаты поиска), как это показано на рис. П2.29. Если окно **Find Results** (Результаты поиска) было до этого скрыто, то оно будет выведено на экран. Номер окна, в которое будут выведены результаты поиска, зависит от состояния флажка **Display in Find 2** (Отобразить в найденном 2).

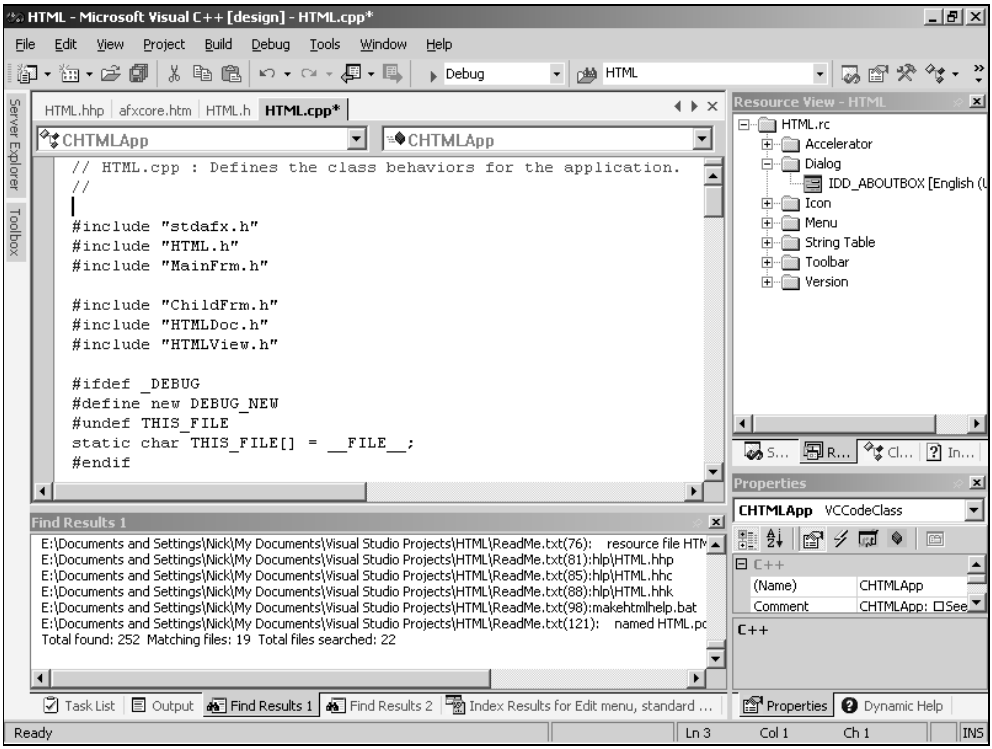


Рис. П2.29. Результат поиска в файлах проекта

Чтобы перейти к найденной строке в файле, достаточно дважды щелкнуть по соответствующей ей строке в окне **Find Results** (Результаты поиска). При этом окно, содержащее данную строку, получит фокус ввода и текстовый курсор переместится в начало этой строки в окне редактирования. Если соответствующее окно редактирования не было открыто, оно автоматически открывается в текстовом режиме. Поэтому поиск в файлах может использоваться для открытия файла ресурсов в текстовом режиме. Для этого нужно произвести поиск по одному из идентификаторов ресурсов, а затем дважды щелкнуть левой кнопкой мыши по строке, расположенной в файле ресурсов.

При нажатии кнопки **Replace** (Заменить) в диалоговом окне **Find in Files** (Найти в файлах) оно превращается в диалоговое окно **Replace in Files** (Заменить в файлах), изображенное на рис. П2.30. Это диалоговое окно можно также вызвать командой

меню **Edit | Find and Replace | Replace in Files** (Правка | Найти и заменить | Заменить в файлах) или нажатием комбинации клавиш <Ctrl>+<Shift>+<H>.

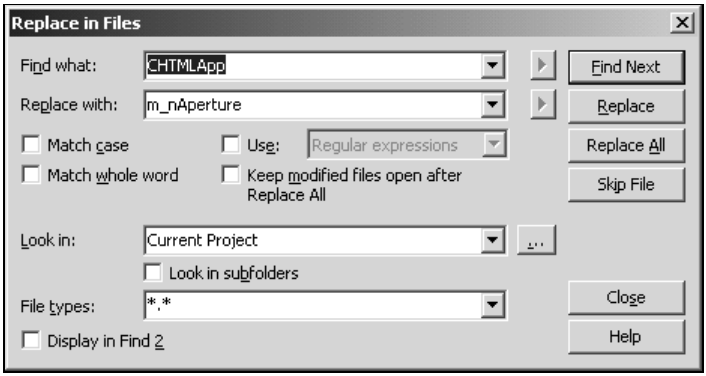


Рис. П2.30. Диалоговое окно **Replace in Files**

Это диалоговое окно работает аналогично диалоговому окну **Replace** (Заменить), но позволяет произвести замену текста не только в окне, имеющем в настоящее время фокус ввода, но и во всех файлах, определяемых содержимым текстовых полей раскрывающихся списков **Look in** (Искать в) и **File types** (Типы файлов).

При выборе команды **Edit | Find and Replace | Find Symbol** (Правка | Найти и заменить | Найти символ) или нажатии комбинации клавиш <Ctrl>+<Shift>+<Y> появляется диалоговое окно **Find Symbol** (Найти символ), изображенное на рис. П2.31.

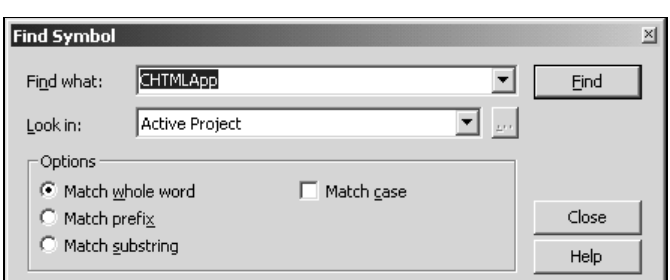


Рис. П2.31. Диалоговое окно **Find Symbol**

Это окно позволяет производить поиск символьных объектов, под которыми понимаются именованные области, классы, структуры, интерфейсы, типы и их члены, такие как свойства, методы, события, переменные и константы. Результаты поиска выводятся в окно **Find Symbol Results** (Результаты поиска символа), изображенное на рис. П2.32.

Это диалоговое окно имеет следующие элементы управления:

- **Find what** (Найти) — текстовое поле раскрывающегося списка, в котором указывается полное или частичное имя искомого символа. В раскрывающемся списке содержатся символы, поиск по которым уже производился;

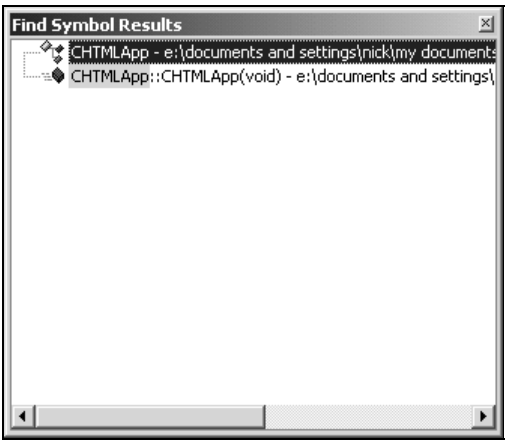


Рис. П2.32. Окно **Find Symbol Results**

- **Look in** (Искать в) — текстовое поле раскрывающегося списка, в котором указывается область, в которой следует производить поиск. В этом раскрывающемся списке может быть выбрана одна из следующих областей поиска:
  - **Active Project** (Активный проект) — ограничивает область поиска активным проектом и его компонентами, на которые имеются ссылки;
  - **Selected Components** (Выбранные компоненты) — позволяет пользователю самому определить область поиска, которая может включать в себя проекты, их компоненты, на которые имеются ссылки, и другие внешние компоненты;

    Для выбора области поиска пользователь должен нажать кнопку **Browse** (Просмотр). При этом появится диалоговое окно **Selected Components** (Выбранные компоненты), изображенное на рис. П2.33. Для включения компонента в область поиска необходимо раскрыть папку, в которой он содержится, и установить флажок у его имени;
- **Match whole word** (Только целое слово) — при установке переключателя в это положение будет произведен поиск символов, полностью совпадающих с указанной строкой. Например, при поиске по строке "Demo" будет найден символ "Demo", но не будут найдены символы "DemoObject" и "CDemo";
- **Match prefix** (Учитывать предшествующие символы) — при установке переключателя в это положение будет произведен поиск символов, начинающихся с указанной строки. То есть не будут найдены символы, в которых перед указанной строкой стоит некоторый префикс. Например, при поиске по строке "Demo" будут найдены символы "Demo" и "DemoObject", но не будет найден символ "CDemo";
- **Match substring** (Не учитывать окружающие символы) — при установке переключателя в это положение будет произведен поиск символов, включающих указанную строку. Например, при поиске по строке "Demo" будут найдены символы "Demo", "DemoObject" и "CDemo";

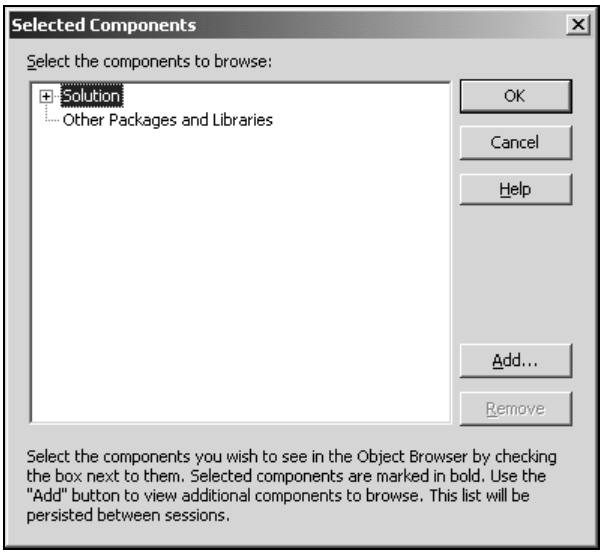


Рис. П2.33. Диалоговое окно **Selected Components**

- **Match case** (Учитывать регистр) — полностью аналогичен одноименному флажку в диалоговых окнах **Find** (Найти) и **Replace** (Заменить). При поиске учитывается регистр символов.

Команда *Edit | Go To*

Данная команда меню выводит на экран диалоговое окно **Go To Line** (Перейти), изображенное на рис. П2.34.

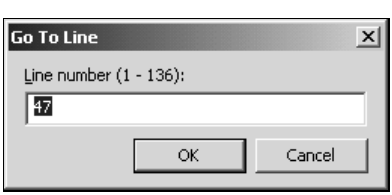


Рис. П2.34. Диалоговое окно **Go To Line**

Это диалоговое окно позволяет перейти к строке с конкретным номером, который указывается в текстовом поле данного диалогового окна. Для того чтобы пользователь не ввел в него недопустимый номер строки, в заголовке текстового поля указан диапазон допустимых значений.

Команда *Edit | Insert File As Text*

Данная команда меню используется для вставки содержимого одного файла в другой файл в текстовом виде. Вставка производится в текущую позицию текстового курсо-

ра, а для выбора вставляемого файла на экран выводится диалоговое окно **Insert File** (Вставить файл), изображенное на рис. П2.35.

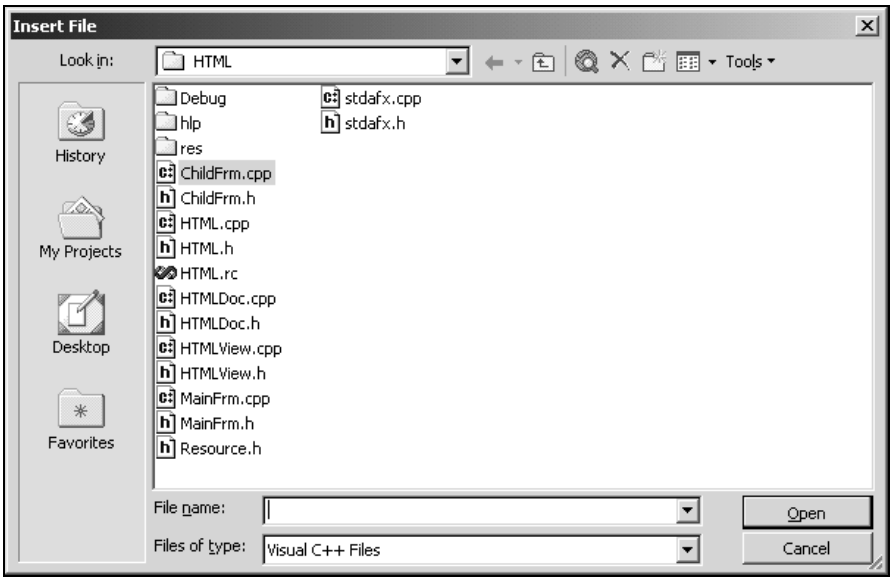


Рис. П2.35. Диалоговое окно **Insert File**

Команда *Edit | Advanced*

Данная команда меню не является терминальной. При ее вызове на экране появляется контекстное меню, содержащее набор команд форматирования текста, как это показано на рис. П2.36.

Это контекстное меню содержит следующие команды:

- **Format Selection** (Выбор формата) — выбор данной команды производит форматирование выделенного текста в соответствии с требованиями, предъявляемыми при написании программ;
- **Tabify Selection** (Табулирование) — преобразует последовательность пробелов в выделенном фрагменте текста в символы табуляции;
- **Untabify Selection** (Растабулирование) — преобразует в выделенном фрагменте текста символы табуляции в последовательность пробелов;
- **Make Uppercase** (Перевод в верхний регистр) — преобразует символы выделенного текста в заглавные;
- **Make Lowercase** (Перевод в нижний регистр) — преобразует символы выделенного текста в строчные;
- **Delete Horizontal White Space** (Удалить горизонтальные пробелы) — уничтожает множественные пробелы и символы табуляции в позиции текстового курсора. В результате выполнения этой операции на месте нескольких пробелов остается

один пробел. На месте символов табуляции или комбинации этих символов с пробелами не остается ничего;

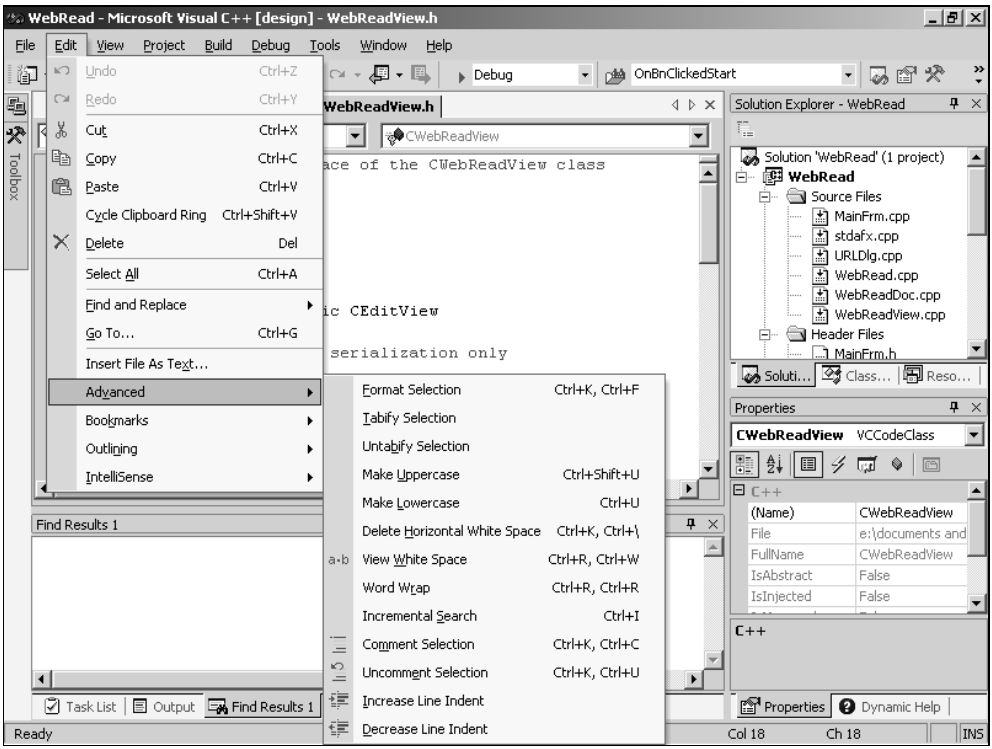


Рис. П2.36. Контекстное меню **Advanced**

- **View White Space** (Визуализация пробелов) — вставляет символы-заполнители (".” для пробела и ">>" для символа табуляции);
- **Word Wrap** (Автоматический переход на новую строку) — при выборе этой команды меню в окне редактирования текста отменяется режим горизонтальной прокрутки. При достижении текстом правой границы окна он автоматически переносится на новую строку;
- **Incremental Search** (Поиск по возрастанию) — эта команда меню позволяет осуществить более быстрый поиск, чем диалоговое окно **Find** (Найти). При выборе этой команды внешне ничего не происходит, однако при вводе с клавиатуры символов в тексте начинает выделяться первое включение данной комбинации символов начиная с текущей позиции курсора. По мере ввода новых символов выделенный фрагмент изменяется и, возможно, перемещается по тексту. Эта команда удобна для поиска первого вхождения идентификатора;
- **Comment Selection** (Превращение выделенного текста в комментарии) — помещает перед каждой строкой выделенного текста символы строкового комментария

(//). Если в окне редактирования отсутствует выделение, комментируется текущая строка;

- **Uncomment Selection** (Убрать комментарии в выделенном фрагменте) — отменяет действие команды **Comment Selection** (Превращение выделенного текста в комментарии);
- **Increase Line Indent** (Увеличить отступ) — эта команда перемещает текущую строку на одну позицию табуляции вправо;
- **Decrease Line Indent** (Уменьшить отступ) — эта команда перемещает текущую строку на одну позицию табуляции влево.

Как следует из рис. П2.35, многим командами этого меню соответствуют двойные нажатия комбинаций клавиш. То есть для вызова соответствующей команды необходимо последовательно нажать две различные (или одинаковые) комбинации клавиш. Надо полагать, что разработчиков Visual Studio.NET замучила ностальгия по Turbo Pascal и другим продуктам десятилетней давности.

Команда *Edit | Bookmarks*

Данная команда меню не является терминальной. При ее вызове на экране появляется контекстное меню, содержащее набор команд для работы с закладками, как это показано на рис. П2.37.

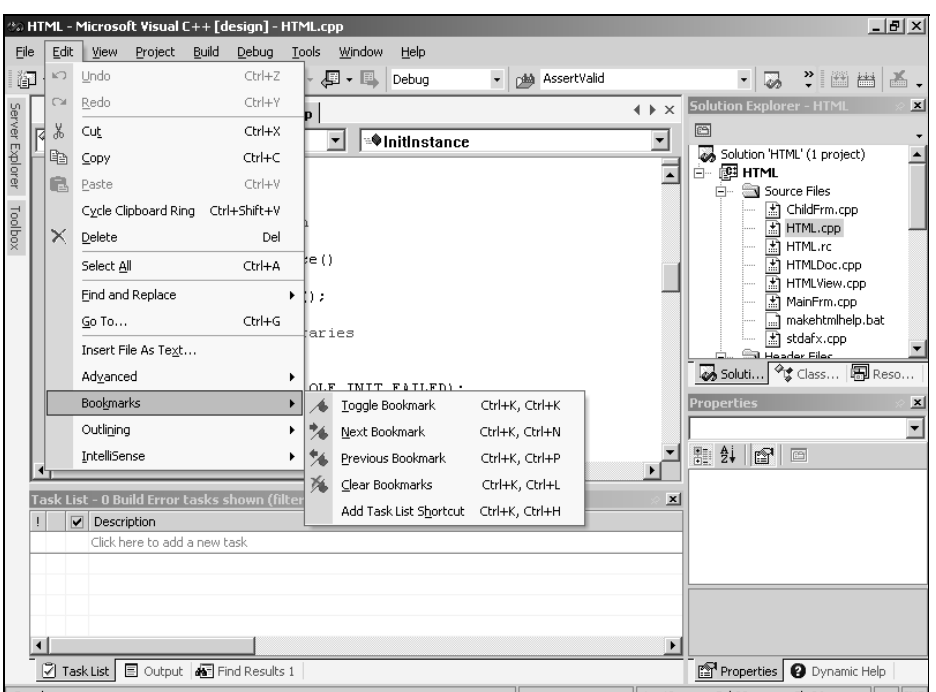


Рис. П2.37. Контекстное меню **Bookmarks**

Данное контекстное меню содержит следующие команды:

- **Toggle Bookmark** (Переключить закладку) — выбор данной команды позволяет установить закладку в строке, если она в ней отсутствует, и удалить эту закладку, если она присутствует;
- **Next Bookmark** (Следующая закладка) — выбор данной команды позволяет перейти к следующей закладке в данном окне;
- **Previous Bookmark** (Предыдущая закладка) — выбор данной команды позволяет перейти к предыдущей закладке в данном окне;
- **Clear Bookmarks** (Удалить закладки) — выбор данной команды позволяет удалить все закладки в данном окне;
- **Add Task List Shortcut** (Добавить метку списка задач) — строка, помеченная такой меткой, выводится в окно **Task List** (Список задач) (если в контекстном меню данного окна была выбрана команда **Show Tasks | Shortcut** (Показать задачи | Метка) или **Show Tasks | All** (Показать задачи | Все)). Двойной щелчок левой кнопкой мыши на этой строке в окне **Task List** (Список задач) позволяет перейти к ней в окне редактирования. Если в текущей строке уже установлена эта метка, данная команда в меню заменяется командой **Remove Task List Shortcut** (Удалить метку списка задач).

Команда *Edit | Outlining*

Данная команда меню не является терминальной. При ее вызове на экране появляется контекстное меню, содержащее набор команд для работы со структурой проекта, как это показано на рис. П2.38.

Это контекстное меню разрабатывалось исходя из структуры приложений языка C#. Visual C++ имеет другую структуру и поэтому команды данного меню выглядят для данного языка несколько натянутыми и работают далеко не во всех случаях.

Контекстное меню **Outlining** (Режим структуры) содержит следующие команды:

- **Hide Selection** (Скрыть выделение) — выбор данной команды позволяет скрыть выделенный фрагмент программы. При этом весь выделенный фрагмент заменяется рамкой с тремя точками, как показано на рис. П2.39. Обратите внимание на то, что выделенный фрагмент может не являться законченной конструкцией;
- **Toggle Outlining Expansion** (Переключить режим отображения) — если текущая строка программы расположена в открытом узле структурной иерархии, данная команда скрывает текущий уровень структурной иерархии, как это показано на рис. П2.40. Если текущая строка содержит скрытый уровень структурной иерархии, вызов данной команды отменит скрытие и выведет программный текст данного уровня. Эта команда меню работает только в том случае, если в текущем окне существует разметка структурной иерархии;
- **Toggle All Outlining** (Переключить режим структуры) — если в окне программы содержатся раскрытые узлы структурной иерархии, данная команда закрывает все эти узлы, как это показано на рис. П2.41. В противном случае данная команда отменяет скрытие всех элементов в текущем окне. Данная команда меню работает только в том случае, если в текущем окне существует разметка структурной иерархии;

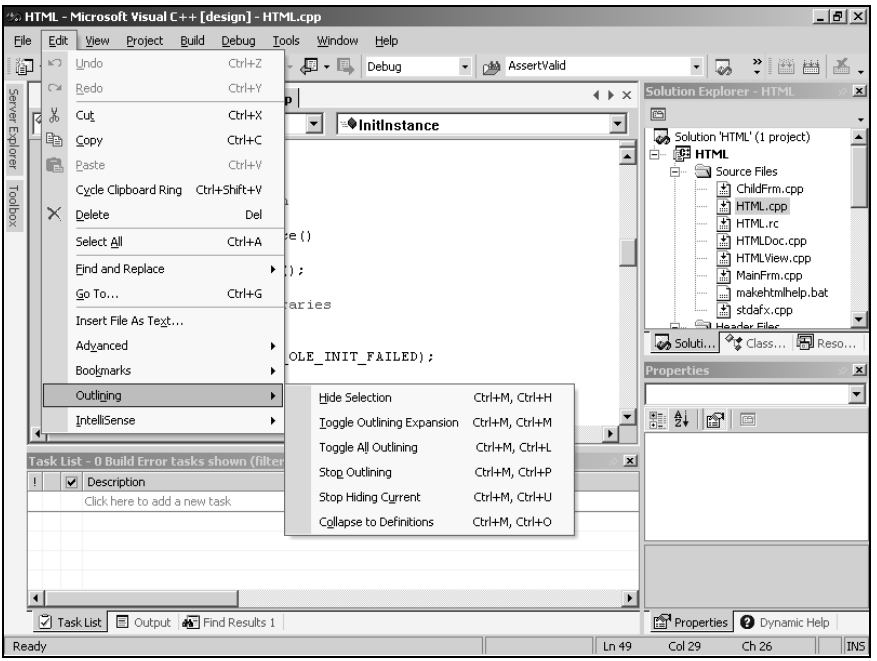


Рис. П2.38. Контекстное меню **Outlining**

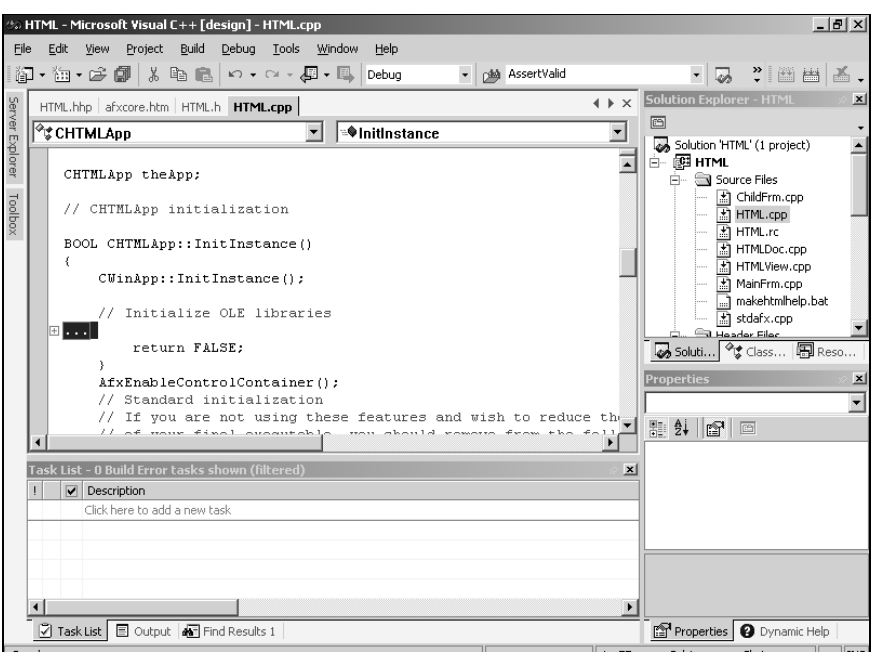


Рис. П2.39. Скрытие выделенного фрагмента

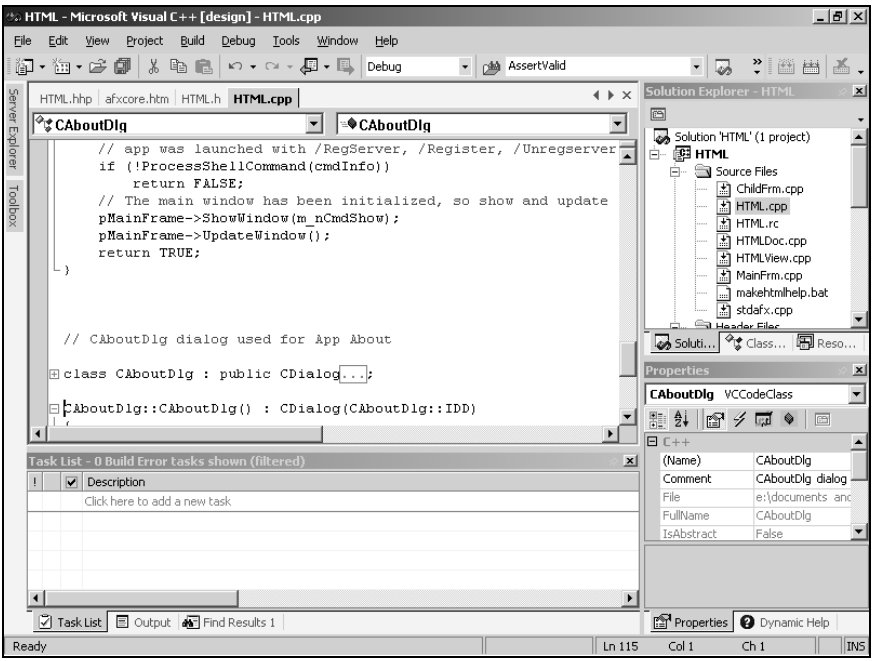


Рис. П2.40. Скрытие элемента программы

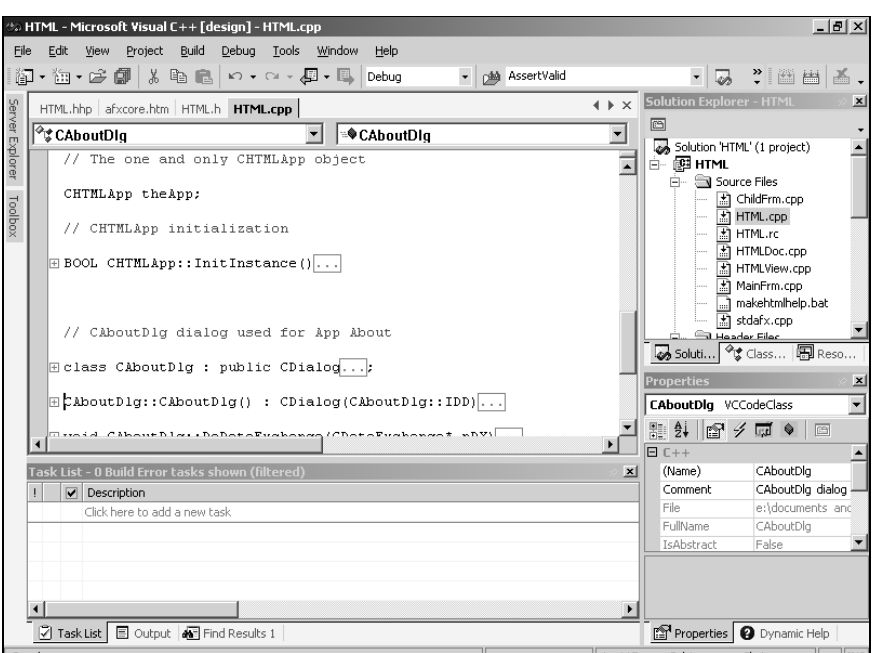


Рис. П2.41. Полное скрытие элементов программы

- **Stop Outlining** (Отмена режима структуры) — отменяет скрытие всех элементов в окне и отменяет действие всех перечисленных выше команд данного контекстного меню (команды остаются доступными, но их вызов не приводит ни к каким последствиям);
- **Stop Hiding Current** (Отменить текущее скрытие) — отменяет действие команды **Hide Selection** (Скрыть выделение);
- **Collapse to Definitions** (Сжать до объявлений) — если в окне программы содержатся раскрытые узлы структурной иерархии, данная команда аналогична команде **Toggle All Outlining** (Переключить режим структуры), поэтому на рис. П2.42 действие этой команда показано не в окне редактирования файла реализации, а в окне редактирования файла заголовка. Если же в окне программы отсутствуют раскрытые узлы структурной иерархии, вызов данной команды не приводит ни к каким последствиям. Поскольку по умолчанию в окнах редактирования файла Visual C++ не производится выделение структурной иерархии, эта команда, наряду с командой **Hide Selection** (Скрыть выделение), является единственным способом ее создать. При этом следует помнить, что команда **Hide Selection** (Скрыть выделение) создает узел для выделенного фрагмента, а данная команда выводит структуру всего окна.

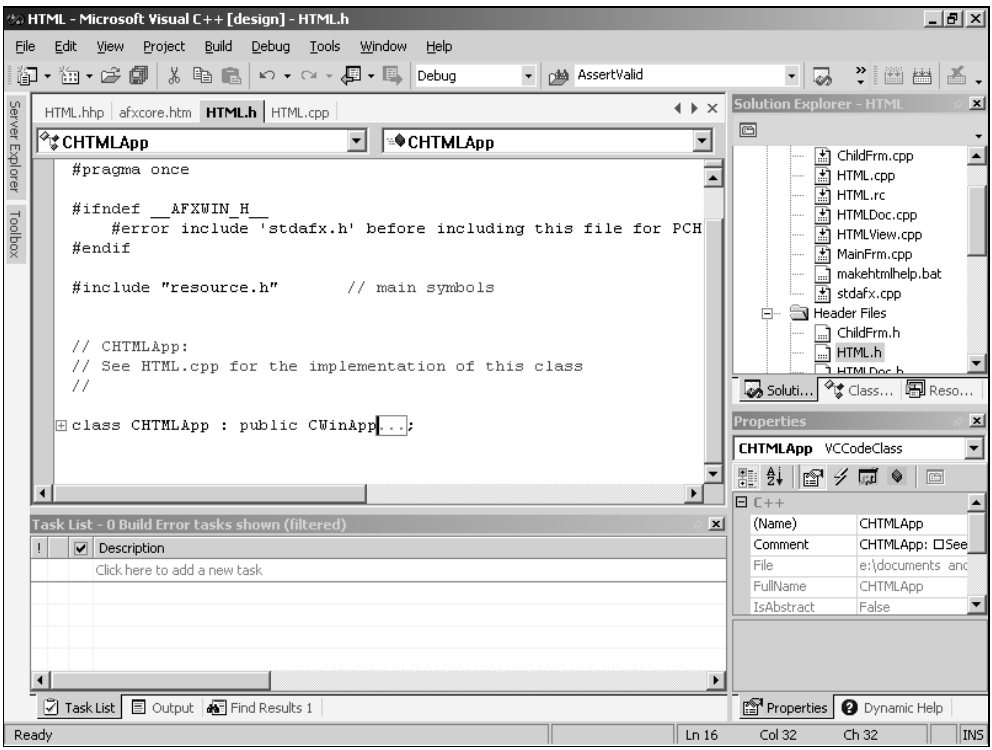


Рис. П2.42. Структура программы

Команда *Edit | IntelliSense*

Эта команда меню не является терминальной. При ее вызове на экране появляется контекстное меню, содержащее набор команд, позволяющих пользователю автоматизировать процесс написания программы, как это показано на рис. П2.43. Большинство возможностей, реализованных в данном меню, выполняются автоматически, поэтому его следует вызывать только в крайних случаях.

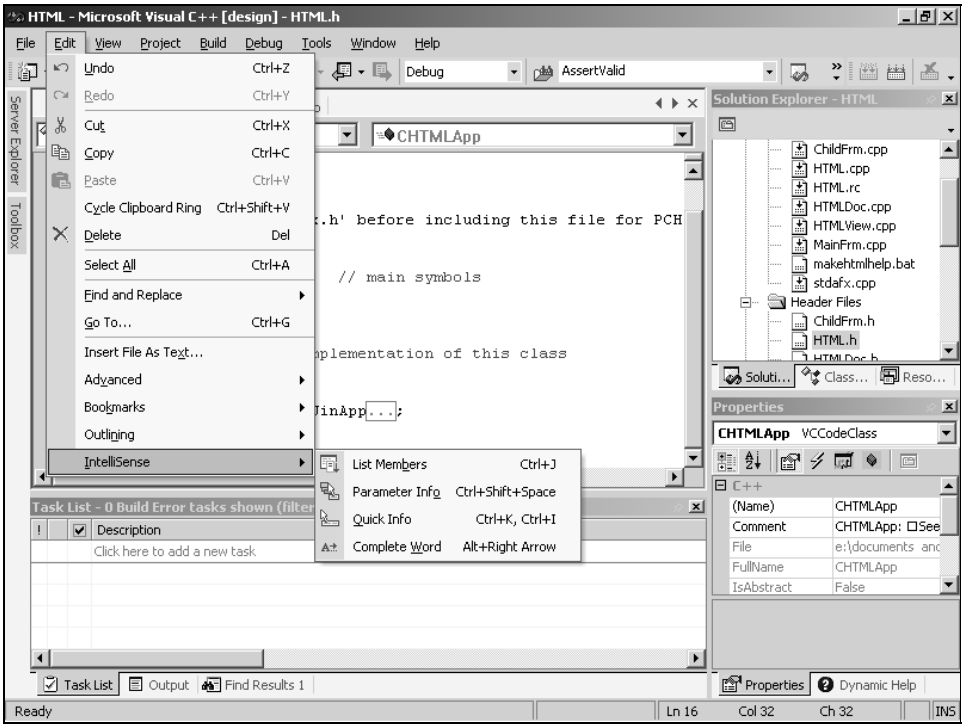


Рис. П2.43. Контекстное меню **IntelliSense**

Данное контекстное меню содержит следующие команды:

- **List Members** (Список членов) — выводит список членов типа для ближайшего к текстовому курсору объекта, как это показано на рис. П2.44;
- **Parameter Info** (Сведения о формальных параметрах) — при нахождении текстового курсора в списке аргументов метода выводит информацию о данном формальном параметре и о самом методе, как это показано на рис. П2.45;
- **Quick Info** (Быстрая справка) — позволяет получить краткую справку по элементу программы, как это показано на рис. П2.46;
- **Complete Word** (Завершить слово) — позволяет автоматически завершить ввод идентификатора, если введенной информации достаточно для его однозначного определения.

Рис. П2.44. Список членов объекта

Рис. П2.45. Информация о формальных параметрах метода

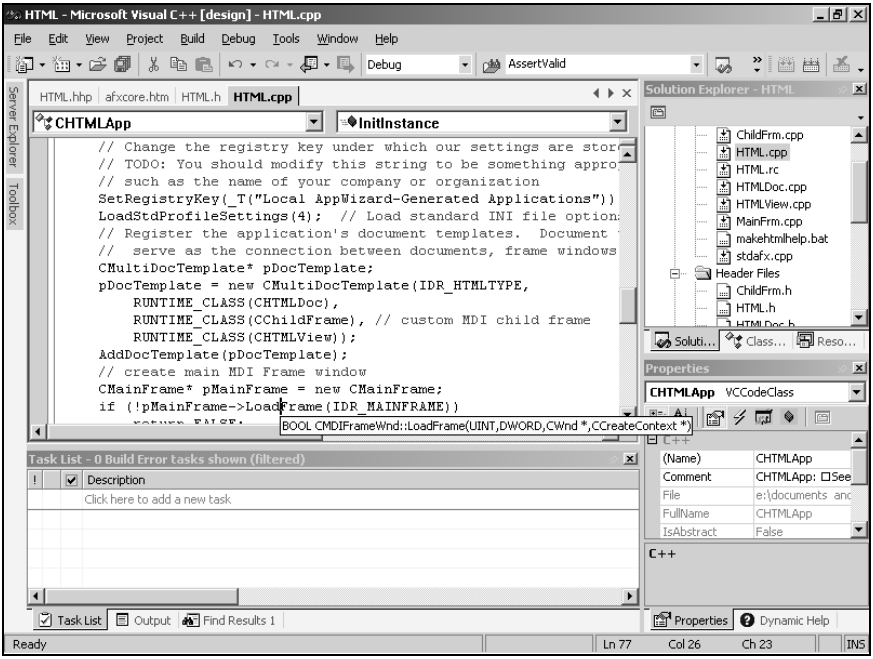


Рис. П2.46. Краткая справка по переменной

Меню *View*

Меню **View** (Вид), показанное на рис. П2.47, содержит команды, осуществляющие управление способом отображения информации на экране, включая управление отображением панелей инструментов и окон среды разработки.

Команда *View | Open*

Данная команда меню позволяет открыть окно редактирования файла или класса, выделенного в окне **Solution Explorer** (Проводник приложения) или **Class View** (Просмотр класса).

Команда *View | Open With*

Данная команда позволяет выбрать редактор, в котором будет открыт указанный файл. Эта же команда позволяет изменить редактор, используемый в текущем окне. В последнем случае появится окно сообщения, предлагающее закрыть текущее окно. При выборе этой команды на экран выводится диалоговое окно **Open With** (Открыть с помощью), изображенное на рис. П2.14.

Команда *View | Solution Explorer*

Данная команда открывает окно **Solution Explorer - HTML** (Проводник решения), содержащее список файлов проекта. Это окно, как правило, выводится в виде вкладки, но на рис. П2.48 оно представлено как самостоятельное окно.

Рис. П2.47. Меню **View**

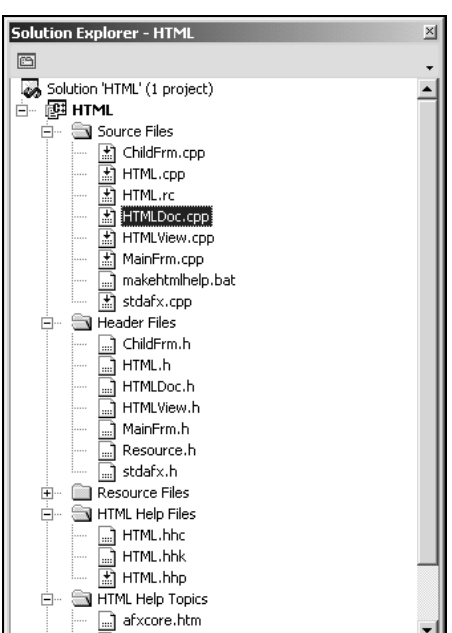


Рис. П2.48. Окно **Solution Explorer - HTML**

Команда *View | Class View*

Данная команда открывает окно **Class View - HTML** (Просмотр класса), содержащее иерархическую структуру типов и именованных областей проекта. Это окно, как правило, выводится в виде вкладки, но на рис. П2.49 оно представлено как самостоятельное окно.

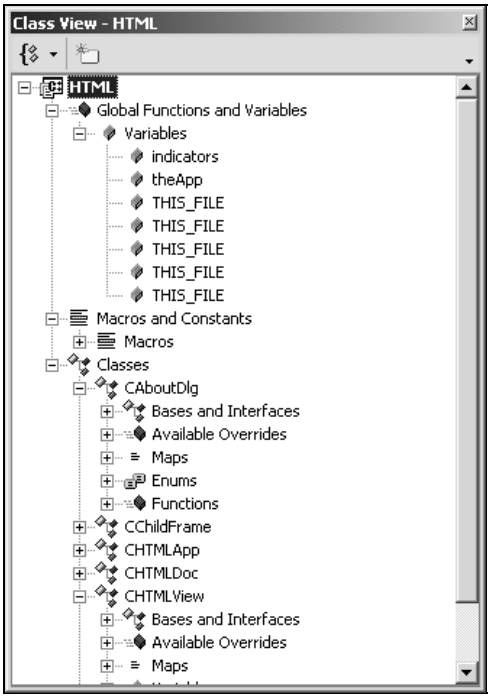


Рис. П2.49. Окно **Class View - HTML**

Команда *View | Server Explorer*

Данная команда открывает вкладку **Server Explorer** (Проводник сервера), как это показано на рис. П2.50. Эта вкладка позволяет задать связь приложения с базами данных и серверами.

Команда *View | Resource View*

Данная команда открывает окно **Resource View - WebRead** (Просмотр ресурсов), содержащее иерархический список ресурсов приложения. Это окно, как правило, выводится в виде вкладки, но на рис. П2.51 оно представлено как самостоятельное окно.

Команда *View | Properties Window*

Данная команда открывает окно **Properties** (Свойства), содержащее набор свойств того или иного программного элемента. Данное окно позволяет разработчику вно-

сить изменения в эти свойства. Это окно, как правило, выводится в виде вкладки, но на рис. П2.52 оно представлено как самостоятельное окно.

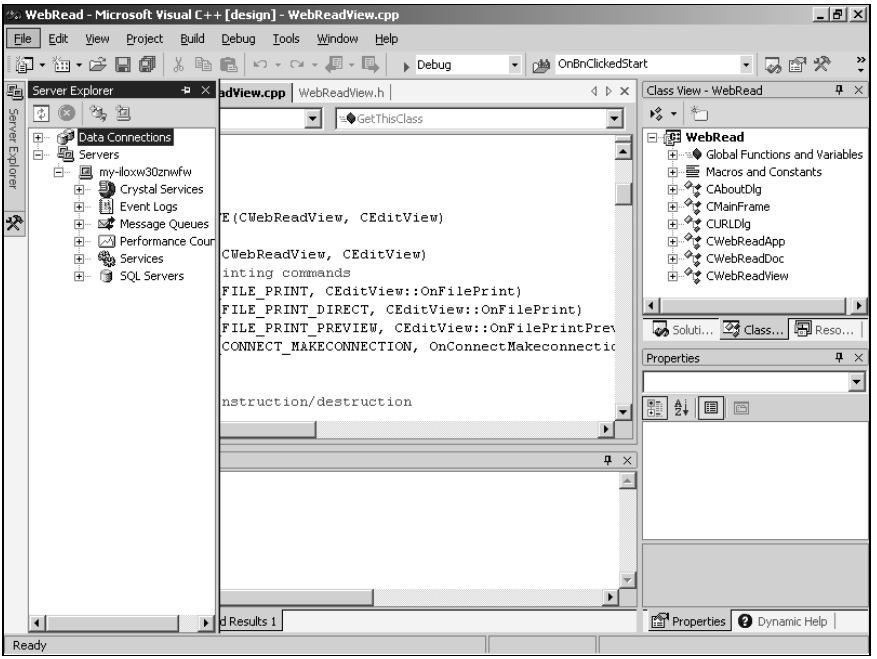


Рис. П2.50. Вкладка **Server Explorer**

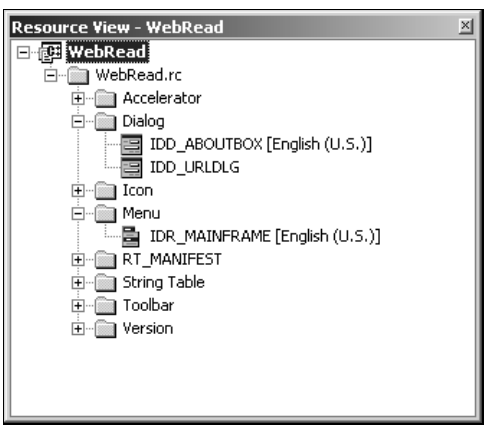


Рис. П2.51. Окно **Resource View - WebRead**

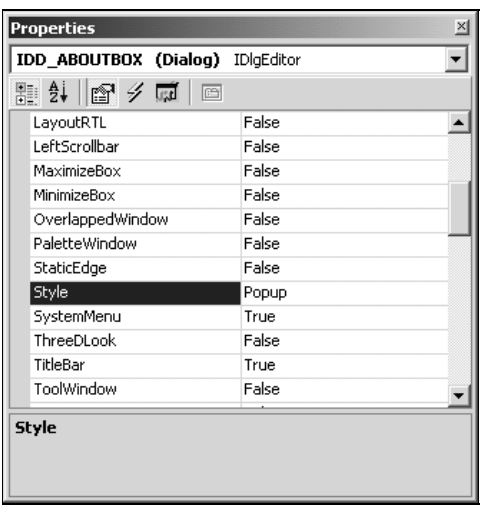


Рис. П2.52. Окно **Properties**

Команда *View | Toolbox*

Данная команда открывает вкладку **Toolbox** (Инструментарий), как это показано на рис. П2.53. Эта вкладка позволяет автоматически создавать в приложении некоторые объекты. В основном, она используется для автоматического добавления элементов управления в диалоговое окно.

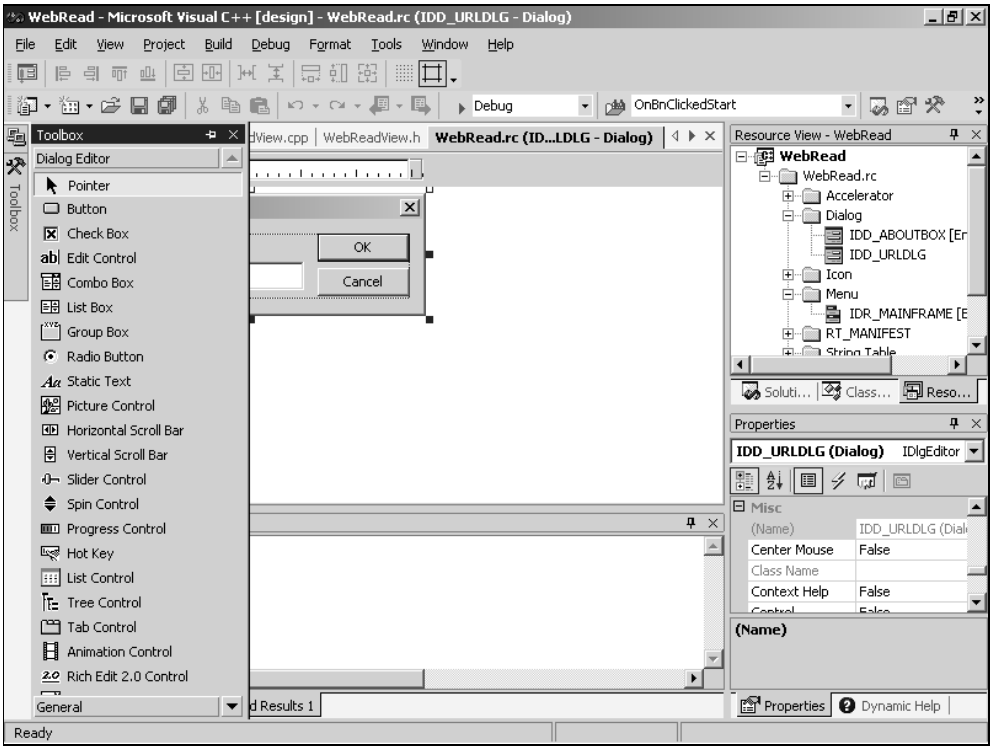


Рис. П2.53. Вкладка **Toolbox**

Команда *View | Web Browser*

Данная команда открывает окно Web-браузера. В данном случае это рассмотренная выше первая страница Visual Studio.NET.

Команда *View | Other Windows*

Данная команда меню не является терминальной и при ее выборе открывается контекстное меню, изображенное на рис. П2.54.

Открываемые в нем окна используются для вывода различной информации в процессе разработки и отладки приложения. Как правило, они открываются автоматически, если в них возникает необходимость.

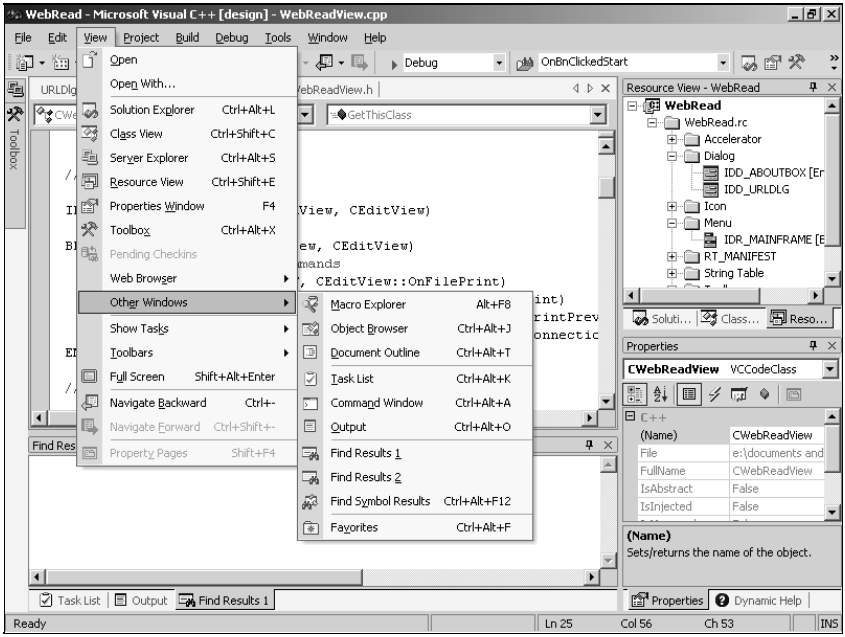


Рис. П2.54. Контекстное меню **Other Windows**

Команда *View | Show Tasks*

Данная команда меню не является терминальной и при ее выборе открывается контекстное меню, изображенное на рис. П2.55.

Команды этого меню определяют информацию, выводимую в окно **Task List** (Список задач), позволяют выполнять некоторые операции, необходимые при работе с несколькими заданиями.

Команда *View | Toolbars*

Данная команда меню не является терминальной и при ее выборе открывается контекстное меню, изображенное на рис. П2.56.

Команды данного контекстного меню являются, по сути дела, флажками, определяющими набор выводимых в среде разработки панелей инструментов.

Команда *View | Full Screen*

Выбор данной команды меню скрывает все панели инструментов, меню, окна вывода и панель задач **Windows**, предоставляя пользователю возможность работать в полноэкранном режиме, как это показано на рис. П2.57.

Для восстановления нормального режима работы в окне остается панель инструментов **Full Screen** (Во весь экран), содержащая единственную кнопку **Full Screen** (Во весь экран), на которую и нужно нажать для выхода из полноэкранного режима отображения.

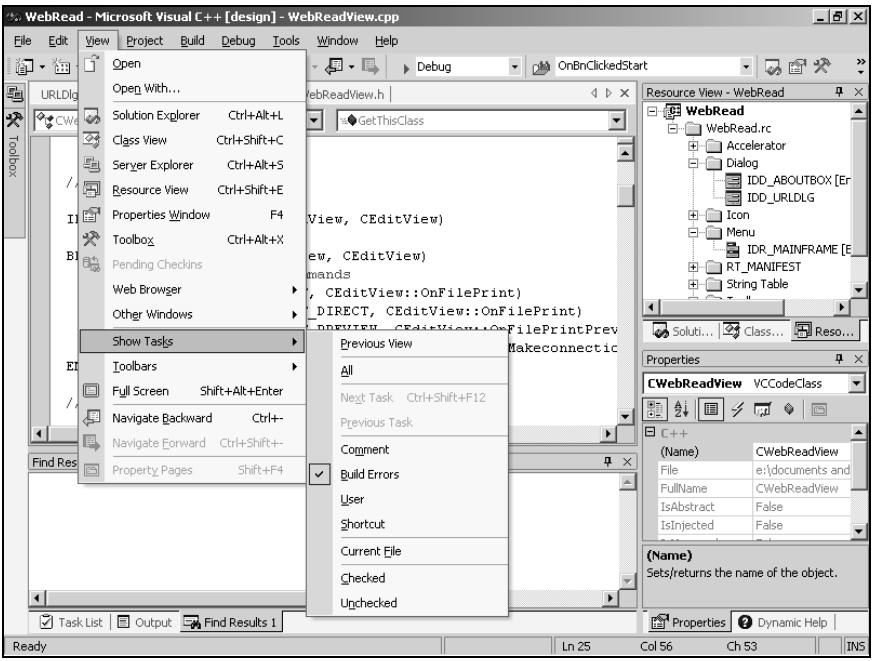


Рис. П2.55. Контекстное меню **Show Tasks**

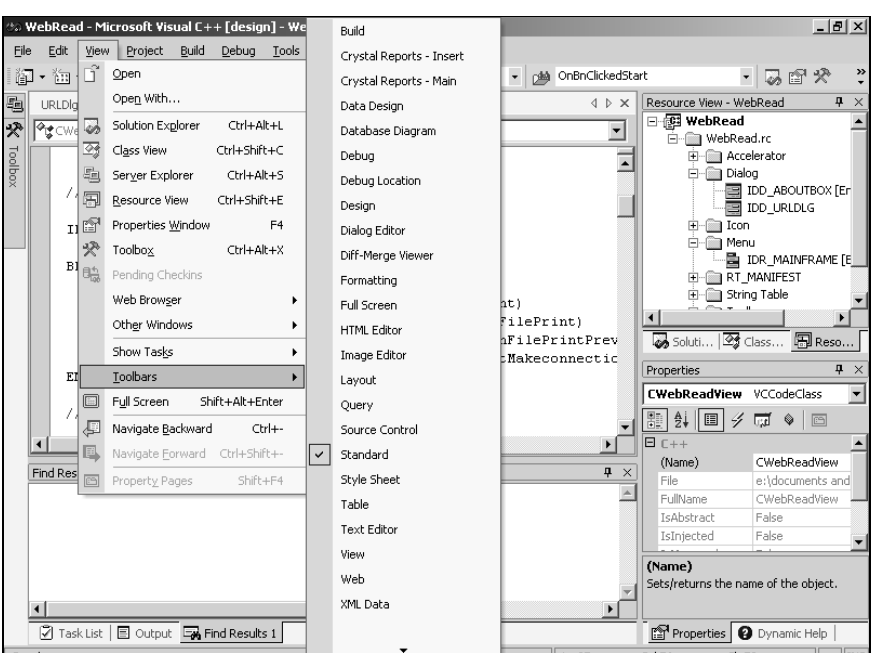


Рис. П2.56. Контекстное меню **Toolbars**

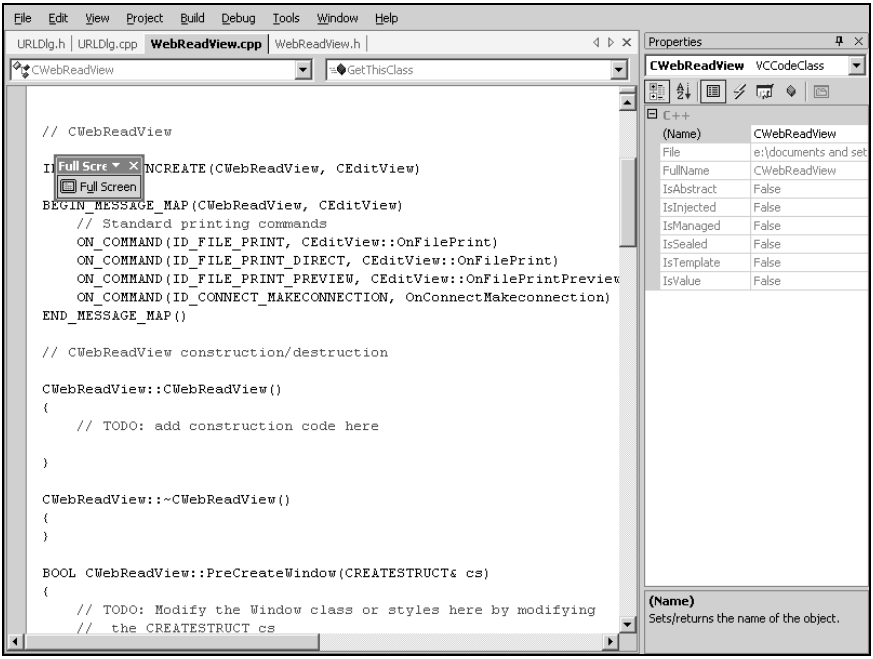


Рис. П2.57. Работа в полноэкранном режиме

Команда *View* | *Navigate Backward*

Как уже говорилось при рассмотрении кнопок панели инструментов **Standard** (Стандартная), в которой дублирована данная команда меню, ее выбор позволяет отменять команды перемещения текстового курсора в файлах проекта.

Команда *View* | *Navigate Forward*

Данная команда меню отменяет однократное действие команды **Navigate Backward** (Возвратиться назад).

Команда *View* | *Property Pages*

Данная команда меню позволяет открыть диалоговое окно **WebRead Property Pages** (Страницы свойств), изображенное на рис. П2.58. В этом окне могут быть изменены установки текущего проекта. Чтобы эта команда меню стала доступной, следует выделить имя проекта в окне **Solution Explorer** (Проводник приложения) или **Class View** (Просмотр класса).

Меню *Project*

Меню **Project** (Проект), приведенное на рис. П2.59, содержит команды, осуществляющие управление разрабатываемым проектом в целом.

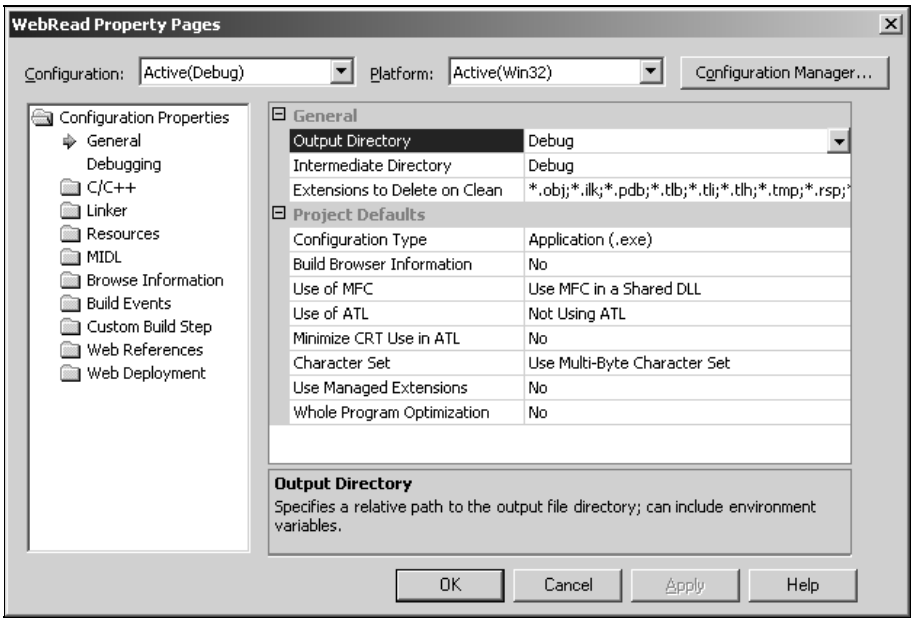


Рис. П2.58. Диалоговое окно **WebRead Property Pages**

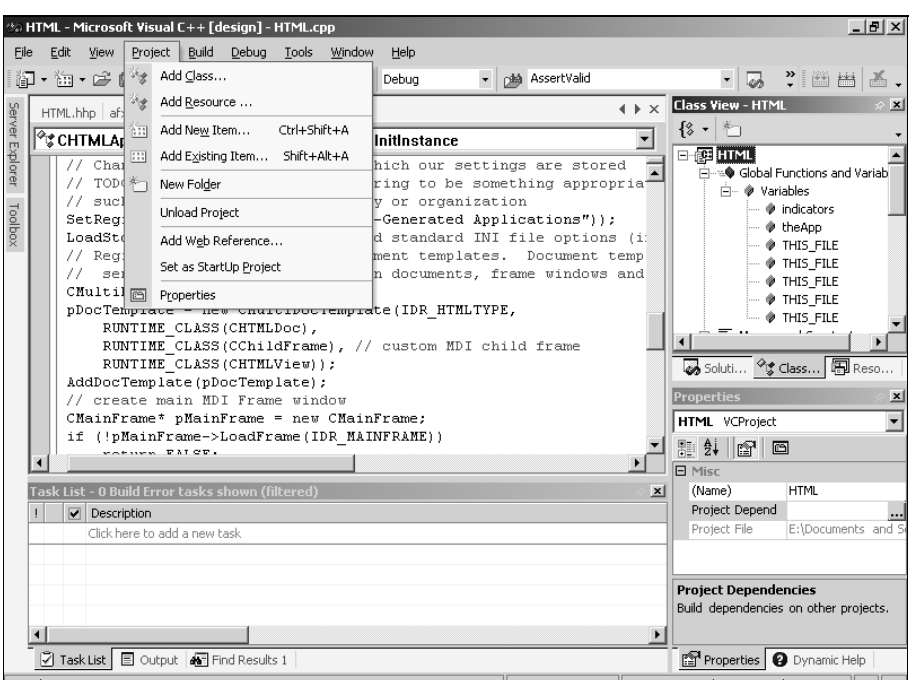


Рис. П2.59. Меню **Project**

Команда *Project | Add Class*

При выборе данной команды меню появляется диалоговое окно **Add Class - HTML** (Добавить класс), изображенное на рис. П2.60. Данное диалоговое окно позволяет выбрать шаблон создаваемого класса. После выбора шаблона класса и нажатия кнопки **Open** (Открыть) появляется диалоговое окно **MFC Class Wizard** (Мастер создания класса MFC), позволяющее создать новый класс, выбрать для него базовый класс и связать его с идентификатором ресурса диалогового окна (если базовым классом является `CDialog` или подобные ему классы).

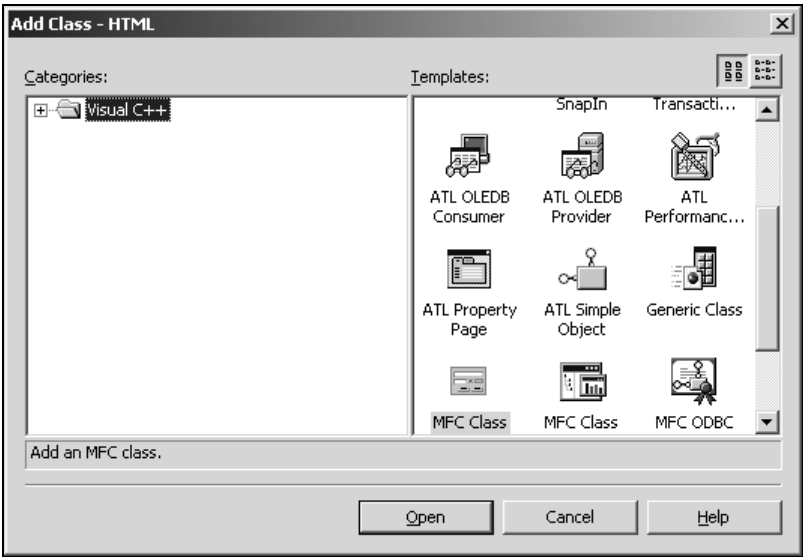


Рис. П2.60. Диалоговое окно **Add Class - HTML**

Команда *Project | Add Function*

Данная команда появляется в меню, если в окне **Class View** (Просмотр класса) выделено имя класса. При ее выборе появляется диалоговое окно **Add Member Function Wizard - HTML** (Мастер добавления функции-члена), изображенное на рис. П2.61. Данное диалоговое окно позволяет полностью задать шаблон новой функции — члена выделенного класса, включая ее имя, режим доступа к ней, тип возвращаемого значения и полный набор ее формальных параметров.

Команда *Project | Add Variable*

Данная команда появляется в меню, если в окне **Class View** (Просмотр класса) выделено имя класса. При ее выборе появляется диалоговое окно **Add Member Variable Wizard - HTML** (Мастер добавления переменной в класс), изображенное на рис. П2.62. Данное диалоговое окно позволяет полностью задать параметры новой переменной — члена выделенного класса, включая ее имя, тип и режим доступа к ней, а также связать ее с идентификатором ресурса, если данный класс является классом диалогового окна.

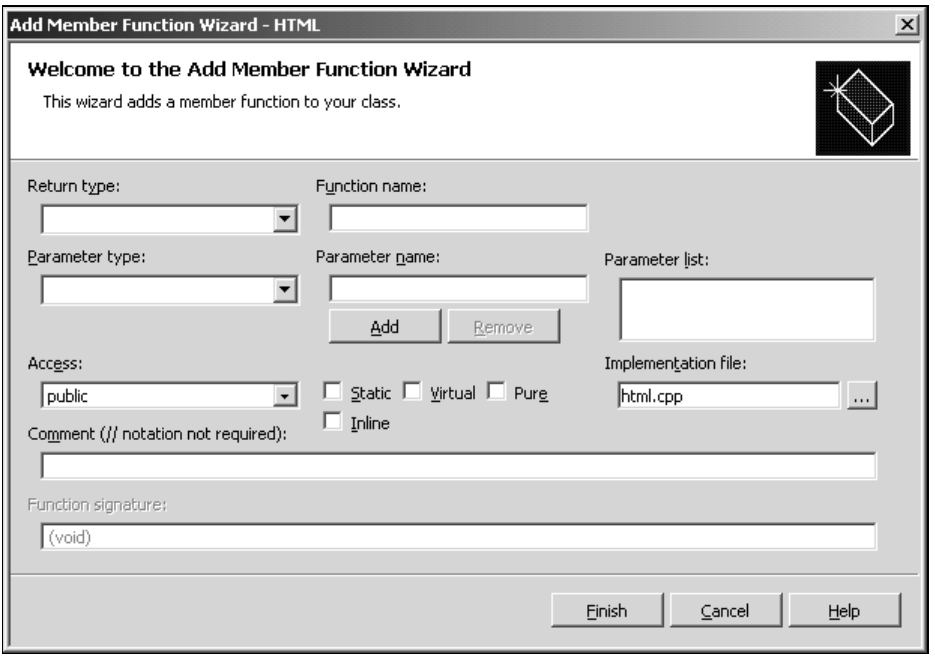


Рис. П2.61. Диалоговое окно **Add Member Function Wizard - HTML**

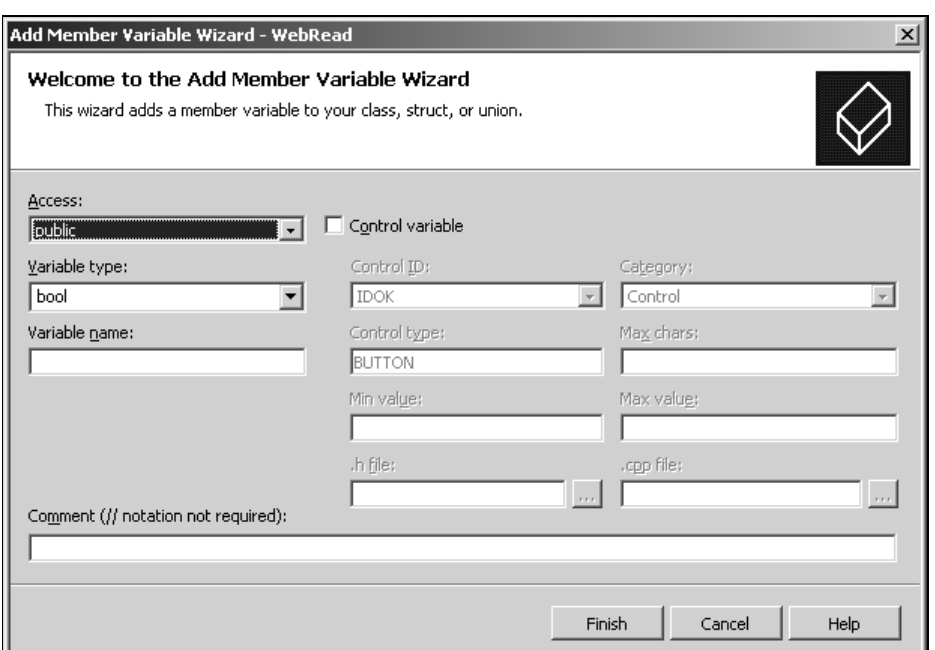


Рис. П2.62. Диалоговое окно **Add Member Variable Wizard - WebRead**

Команда *Project | Add Resource*

При выборе данной команды меню появляется диалоговое окно **Add Resource** (Добавить ресурс), изображенное на рис. П2.63. Данное диалоговое окно позволяет выбрать тип нового ресурса, включаемого в приложение, импортировать в него существующий ресурс или же включить в приложение пользовательский ресурс.

Рис. П2.63. Диалоговое окно **Add Resource**

Команда *Project | Add New Item*

При выборе данной команды меню появляется диалоговое окно **Add New Item - WebRead** (Добавить новый элемент), изображенное на рис. П2.64. Это диалоговое окно позволяет включать в проект пустые файлы соответствующего формата.

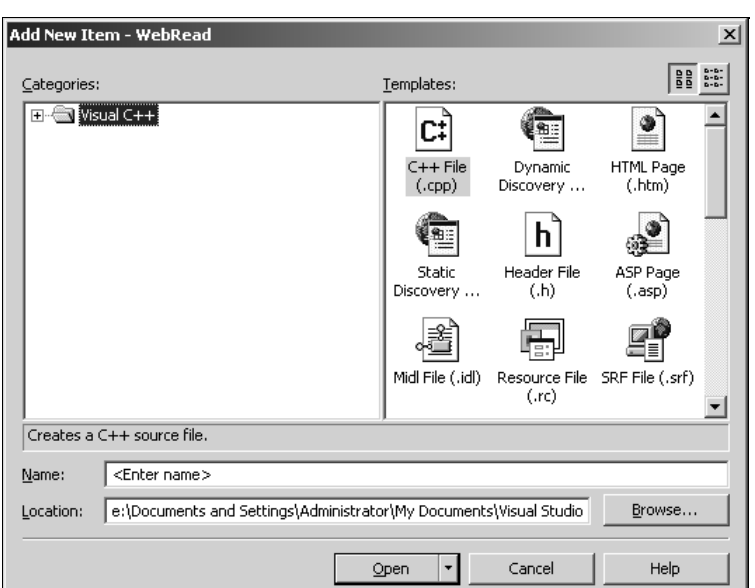


Рис. П2.64. Диалоговое окно **Add New Item - WebRead**

Команда *Project | Add Existing Item*

При выборе данной команды меню появляется диалоговое окно **Add Existing Item - HTML** (Добавить существующий элемент), изображенное на рис. П2.65.

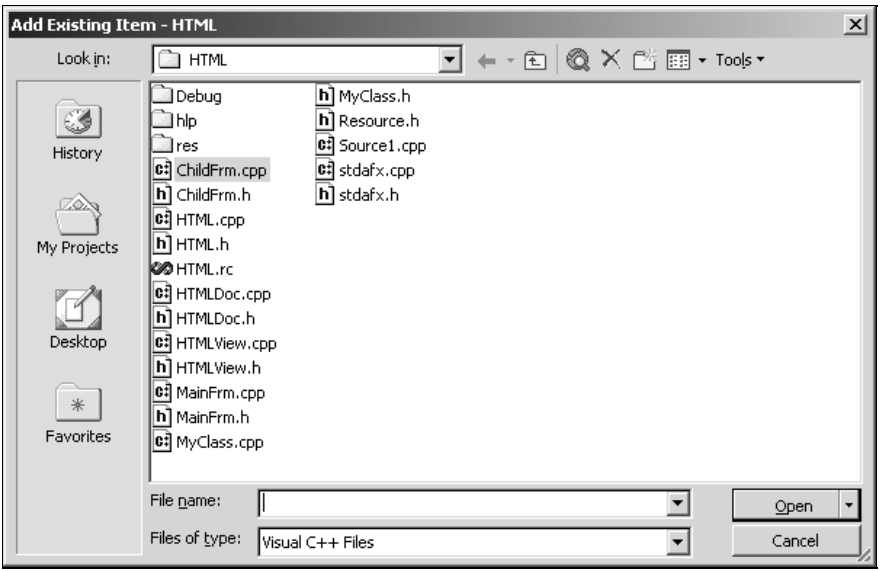


Рис. П2.65. Диалоговое окно **Add Existing Item - HTML**

Данное диалоговое окно позволяет добавить в проект все классы и ресурсы, содержащиеся в выделенном файле. Подобный подход заставляет разработчика семь раз отмерить прежде, чем поместить описание двух типов в один файл. Кроме того, это затрудняет использование в проектах имен файлов, создаваемых Visual Studio.NET. Поскольку эти имена не содержат имен приложения и формируются единообразно, появляется большая вероятность того, что в данном приложении уже существует файл с тем же именем, что и добавляемый. В этом случае Visual Studio.NET не находит лучшего выхода, чем заменить существующий файл новым.

Команда *Project | New Folder*

Данная команда меню позволяет создать в окнах **Class View** (Просмотр класса) и **Solution Explorer** (Проводник решения) новые папки. В окне **Solution Explorer** (Проводник решения) папка создается в текущем приложении (наряду с папками файлов заголовков, файлов реализации, файлов ресурсов и т. д.), а в окне **Class View** (Просмотр класса) папка создается на верхнем уровне иерархии (наряду с папкой проекта).

Команда *Project | Unload Project*

Данная команда появляется в меню, когда в окне **Solution Explorer** (Проводник решения) выделено имя проекта. Она позволяет выгрузить проект из решения. Если

данный проект содержал несохраненные файлы, выводится диалоговое окно, предлагающее их сохранить. Если данный проект был единственным в решении, это решение закрывается.

Для отмены действия этой команды достаточно выбрать появившуюся на ее месте команду **Reload Project** (Перезагрузить проект).

Команда *Project | Add Web Reference*

При выборе данной команды меню появляется диалоговое окно **Add Web Reference** (Добавить Web-ссылку), изображенное на рис. П2.66.

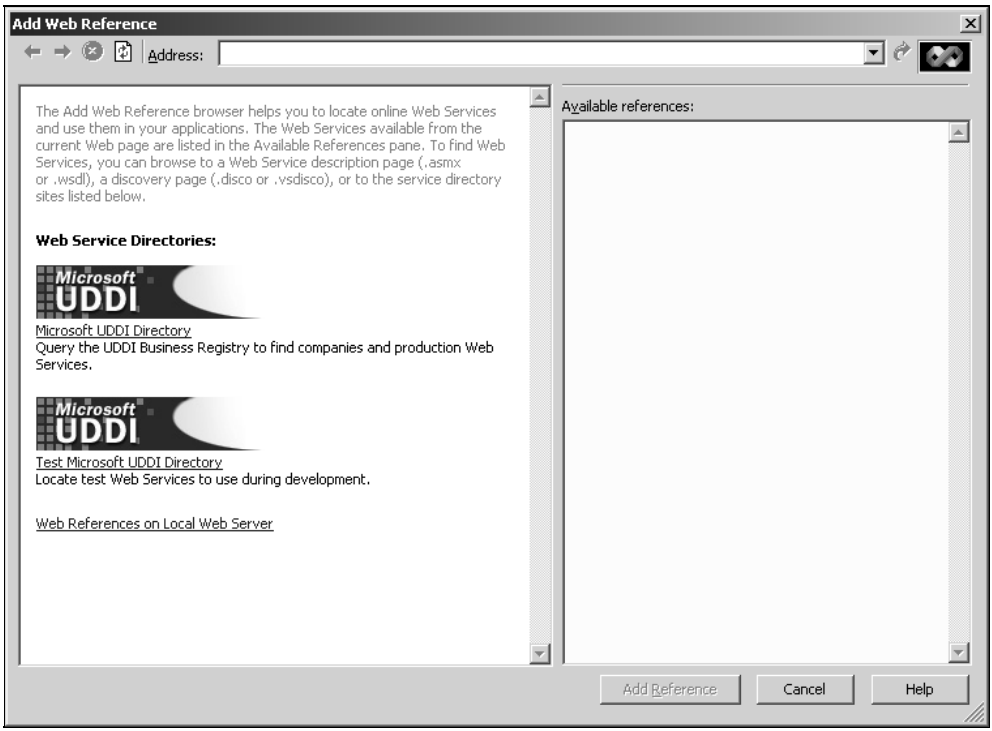


Рис. П2.66. Диалоговое окно **Add Web Reference**

Данное диалоговое окно позволяет добавить в проект ссылки на компоненты, доступные в Сети. При нахождении браузера на Web-странице, содержащей сервисы и наборы данных, ссылки на них отображаются в окне списка **Available references** (Доступные ссылки).

Команда *Project | Set as StartUp Project*

Выбор данной команды меню приводит к тому, что текущий проект будет автоматически загружаться при старте среды Visual Studio.NET.

Команда *Project | Properties*

Данная команда меню аналогична команде меню **View | Property Pages** (Вид | Страницы свойств) и так же, как и она, выводит на экран диалоговое окно **WebRead Property Pages** (Страницы свойств), изображенное на рис. П2.58. Это диалоговое окно позволяет изменить текущие установки проекта или файла, выделенного в окне **Solution Explorer** (Проводник решения) или **Class View** (Просмотр класса).

Меню *Build*

Меню **Build** (Компоновка), приведенное на рис. П2.67, содержит команды компиляции, компоновки и отладки.

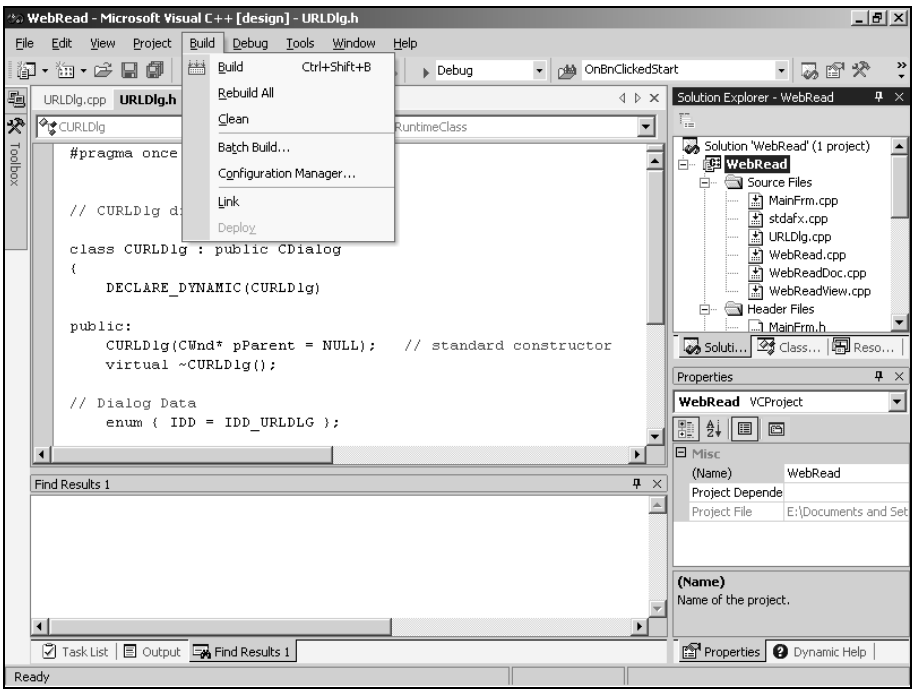


Рис. П2.67. Меню **Build**

Команда *Build | Build*

Данная команда позволяет перекомпилировать все файлы, в которые были внесены изменения. На панели инструментов **Build** (Компоновка) имеется кнопка, соответствующая данной команде.

Команда *Build | Rebuild All*

Данная команда позволяет перекомпилировать все файлы проекта независимо от того, были в них внесены изменения после последней компиляции или нет. Иногда

встречаются случаи, когда в процессе компиляции пропускается файл, который должен быть перекомпилирован. Обычно это имеет место в том случае, если в данный файл были внесены изменения в обход среды программирования Visual Studio.NET (например, вместо одного из файлов была записана его старая версия, взятая из архива).

Команда *Build | Clean*

Данная команда меню уничтожает в проекте все файлы, создаваемые при его трансляции и компоновке для данной конфигурации, оставляя в нем только исходные файлы. Эта команда используется в том случае, когда есть подозрение в том, что команда **Rebuild** (Перекомпоновать) пропускает файл, в который были внесены изменения.

Команда *Build | Batch Build*

Выбор данной команды меню выводит на экран диалоговое окно **Batch Build** (Пакетная компоновка), изображенное на рис. П2.68.

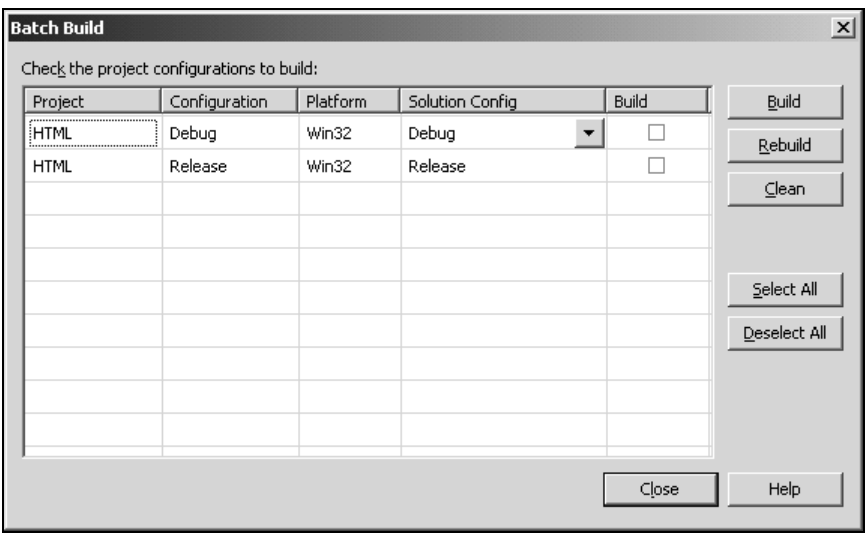


Рис. П2.68. Диалоговое окно **Batch Build**

Данное диалоговое окно позволяет произвести одновременную компиляцию нескольких конфигураций приложения (например, отладочной и распространяемой). Компилируемые конфигурации отмечаются в столбце **Build** (Компоновка) окна списка **Check the project configurations to build** (Отметьте компилируемые конфигурации).

Для компиляции измененных файлов нажимается кнопка **Build** (Компоновка), а для компиляции всех файлов проекта — кнопка **Rebuild** (Перекомпоновка). После успешной компиляции производится компоновка файлов.

Команда *Build | Configuration Manager*

При выборе данной команды меню появляется диалоговое окно **Configuration Manager** (Конфигуратор), изображенное на рис. П2.69.

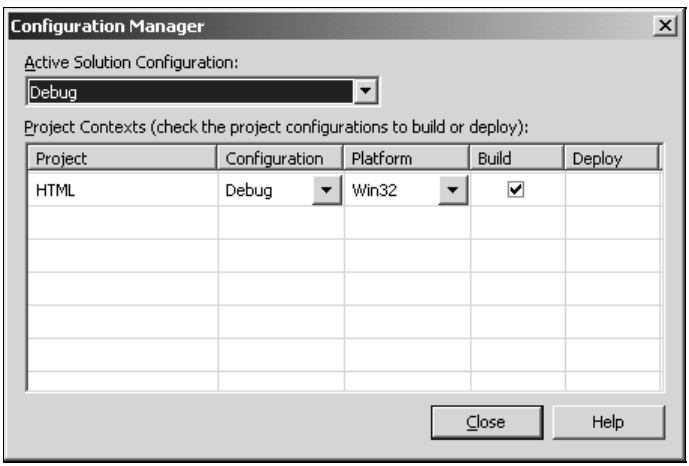


Рис. П2.69. Диалоговое окно **Configuration Manager**

Данное диалоговое окно позволяет выбрать компилируемые и распространяемые конфигурации создаваемого проекта, а также используемую в нем платформу.

Команда *Build | Compile*

Данная команда меню становится доступной, если активное окно содержит текст файла реализации. Выбор этой команды позволяет откомпилировать этот файл.

Команда *Build | Deploy*

Данная команда меню используется для установки создаваемого компонента на удаленном компьютере. Для установки компонента необходимо создать специальный проект установки, в котором определяется состав комплекта поставки данного компонента и другие параметры его распространения.

Меню *Debug*

Меню **Debug** (Отладка), приведенное на рис. П2.70, содержит команды запуска приложения на исполнение и команды, используемые при его отладке.

Команда *Debug | Windows*

Данная команда меню не является терминальной и при ее выборе появляется контекстное меню, изображенное на рис. П2.71. При запуске приложения на исполнение в этом меню появляются дополнительные команды.

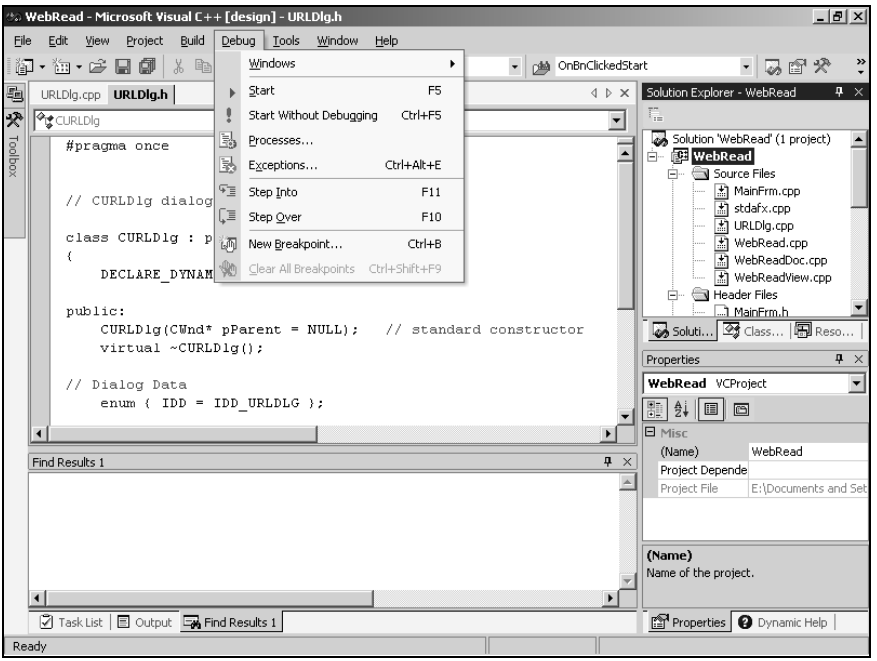


Рис. П2.70. Меню **Debug**

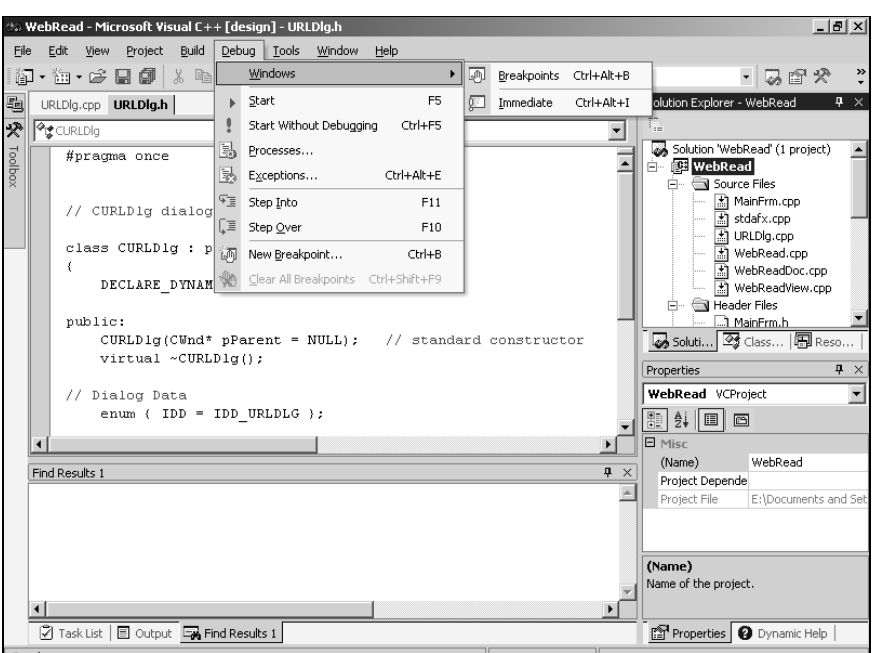


Рис. П2.71. Контекстное меню **Windows**

Команды данного контекстного меню используются для вывода на экран окон, в которые будет выводиться различная отладочная информация.

Команда *Debug | Start*

Выбор данной команды меню запускает приложение на исполнение в режиме отладки. После запуска приложения на исполнение на месте этой команды меню появляется команда **Continue** (Продолжить), возобновляющая выполнение программы в режиме отладки до следующей точки останова или до конца программы, если при ее дальнейшем выполнении не встретятся точки останова.

Команда *Debug | Break All*

Данная команда появляется в меню только в процессе отладки приложения. Ее выбор приостанавливает выполнение текущего приложения, оставляя возможность его старта с текущей точки.

Команда *Debug | Stop Debugging*

Данная команда появляется в меню только в процессе отладки приложения. Ее выбор завершает процесс выполнения отлаживаемого приложения.

Команда *Debug | Detach All*

Данная команда появляется в меню только в процессе отладки приложения. Ее выбор завершает процесс отладки приложения, не прекращая его работы. То есть от приложения отключаются все отладочные процессы.

Команда *Debug | Restart*

Данная команда появляется в меню только в процессе отладки приложения. Ее выбор перезапускает исполняемое приложение. Выполнение перезапущенного приложения останавливается на его первом исполняемом выражении.

Команда *Debug | Apply Code Changes*

Данная команда появляется в меню только в процессе отладки приложения, и становится доступной после того, как в приложение будут внесены какие-либо изменения. Выбор данной команды меню для продолжения работы после внесения изменений в текст исполняемого приложения позволяет избежать появления окна сообщения, запрашивающего информацию о том, следует ли перекомпилировать проект прежде, чем продолжить его исполнение с текущей точки останова.

Команда *Debug | Processes*

При выборе данной команды меню появляется диалоговое окно **Processes** (Процессы), изображенное на рис. П2.72.

Данное диалоговое окно позволяет получить информацию обо всех процессах, запущенных на всех компьютерах сети и осуществить простейшее управление отлаживаемыми процессами (прервать или остановить выполнение процесса или же отключить его).

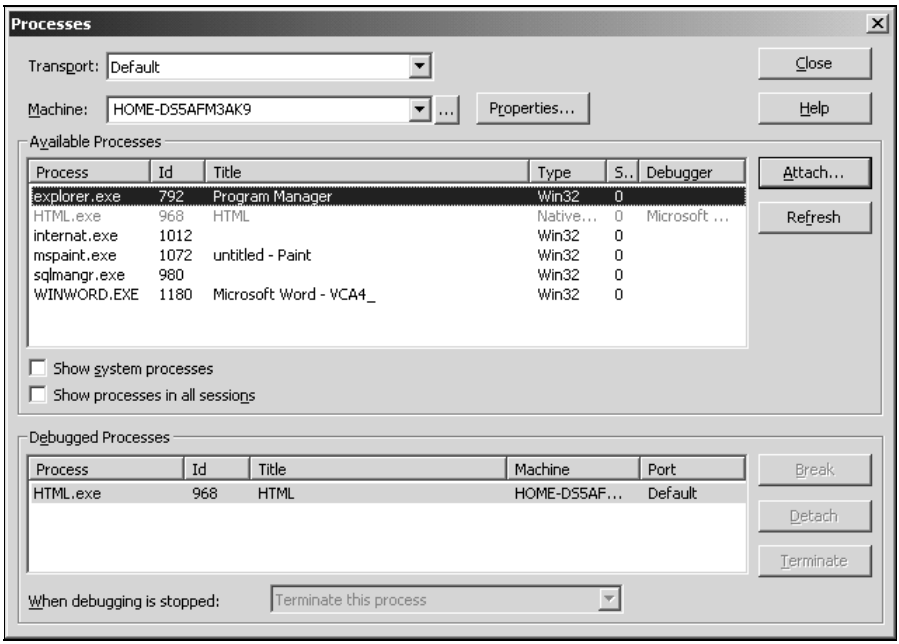


Рис. П2.72. Диалоговое окно **Processes**

Команда *Debug | Exceptions*

При выборе данной команды меню появляется диалоговое окно **Exceptions** (Исключения), изображенное на рис. П2.73.

Данное диалоговое окно позволяет определить действия, предпринимаемые приложением при вызове каждого исключения или в том случае, если это исключение осталось необработанным.

Команда *Debug | Step Into*

Эта команда меню появляется только в процессе отладки приложения. Ее выбор позволяет выполнить один оператор программы и снова остановиться. В том случае, если текущий оператор является функцией, доступной для отладки, следующим оператором программы является первый оператор данной функции (ее заголовок). Если текущий оператор не является функцией, данная команда работает аналогично команде меню **Debug | Step Over** (Отладка | Обойти вызов функции). Если текст данной функции не содержится в каталогах, указанных пользователем при настройке системы, система выводит диалоговое окно, предлагая указать путь к данному файлу.

Команда *Debug | Step Over*

Эта команда меню появляется только в процессе отладки приложения. Ее выбор позволяет выполнить один оператор программы и снова остановиться независимо от того, является ли данный оператор функцией или нет.

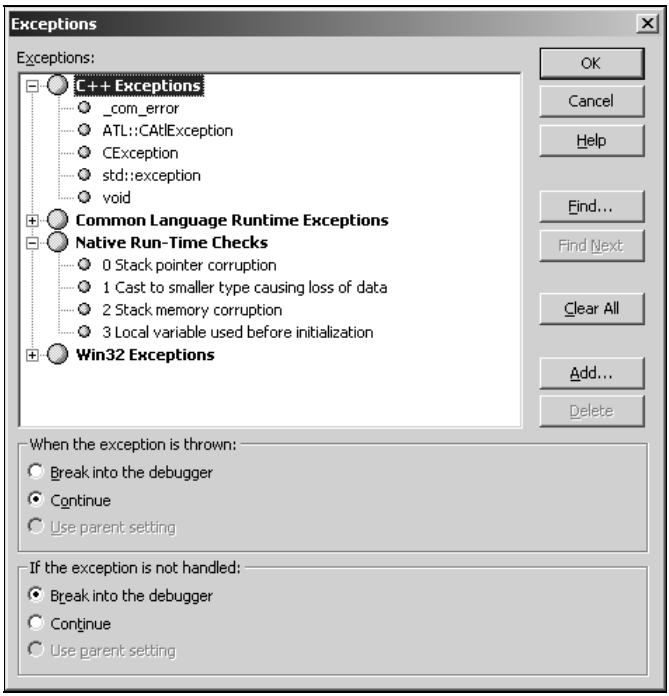


Рис. П2.73. Диалоговое окно **Exceptions**

Команда *Debug | Step Out*

Эта команда меню появляется только в процессе отладки приложения. Ее выбор позволяет выйти из текущей функции и остановиться на первом операторе программы, расположенном за ее вызовом.

Команда *Debug | QuickWatch*

Выбор данной команды меню выводит на экран диалоговое окно **QuickWatch** (Быстрый просмотр), изображенное на рис. П2.74.

Это диалоговое окно позволяет просмотреть значение выражения, которое было выделено в момент вызова данной команды меню или было введено пользователем в текстовое поле раскрывающегося списка **Expression** (Выражение). Нажатие кнопки **Add Watch** (Добавить просмотр) позволяет добавить это выражение в окно **Watch** (Просмотр).

Команда *Debug | New Breakpoint*

Выбор данной команды меню выводит на экран диалоговое окно **New Breakpoint** (Установить точку останова), изображенное на рис. П2.75.

Это диалоговое окно позволяет устанавливать точки останова, указывая при этом не только их местоположение, но и достаточно сложные условия остановки в них программы.

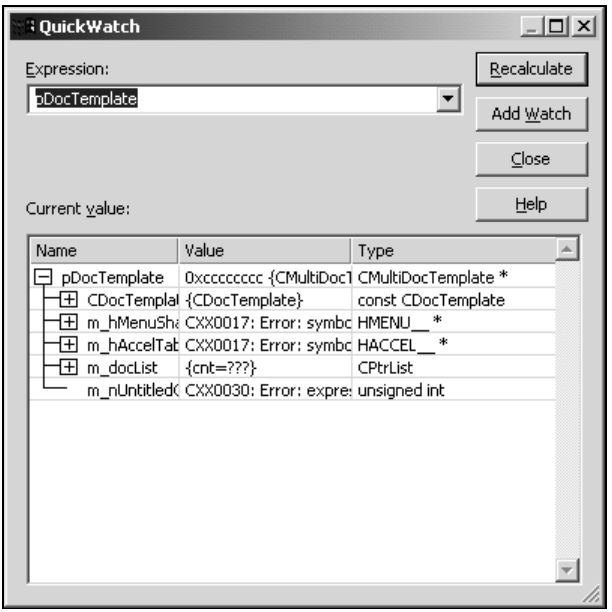


Рис. П2.74. Диалоговое окно **QuickWatch**

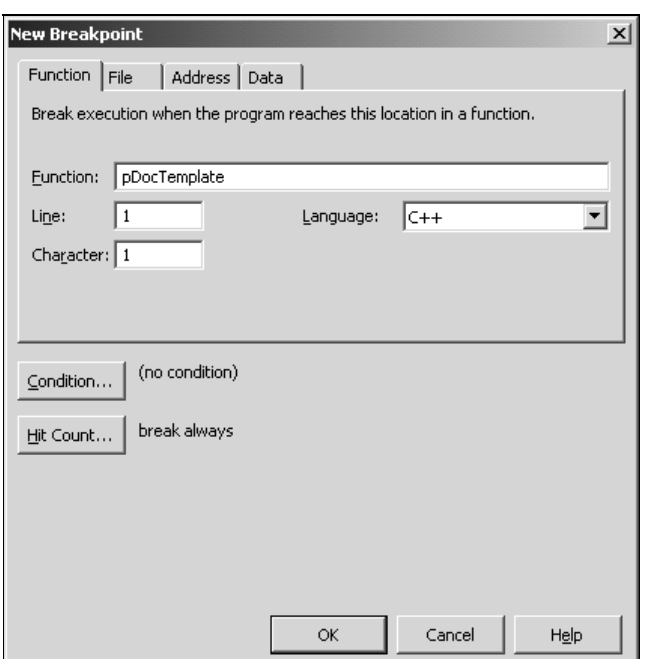


Рис. П2.75. Диалоговое окно **New Breakpoint**

Команда *Debug | Clear All Breakpoints*

Вызов данной команды меню удаляет из приложения все установленные в нем точки останова.

Команда *Debug | Disable Breakpoint*

Вызов данной команды меню отменяет действие точки останова не убирая ее. При этом на месте этой команды появляется команда **Enable Breakpoint** (Восстановить действие точки останова), восстанавливающая действие точки останова. Если в текущей строке не была установлена точка останова, эта команда становится недоступной.

Команда *Debug | Save Dump As*

Выбор данной команды меню выводит на экран диалоговое окно **Save Dump As** (Сохранить дамп как), изображенное на рис. П2.76.

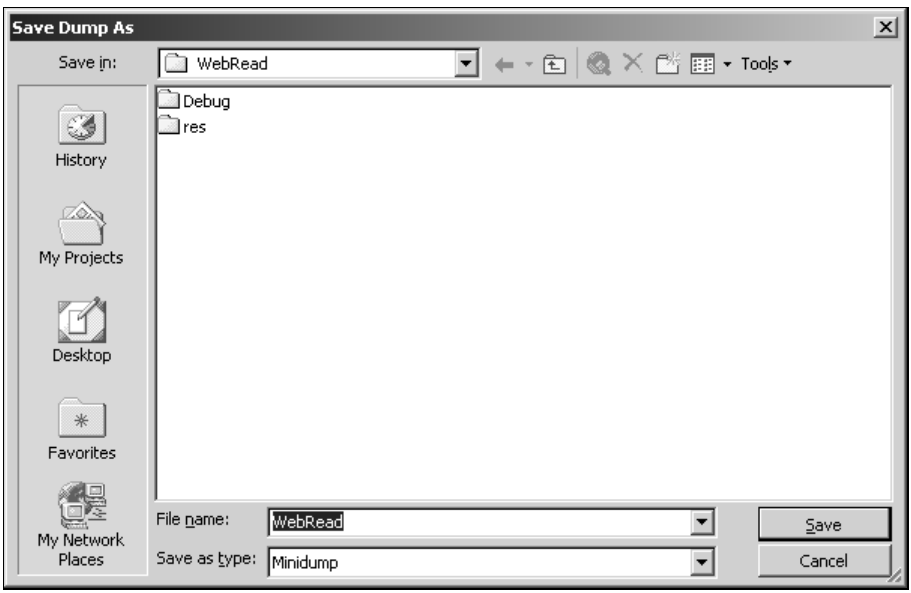


Рис. П2.76. Диалоговое окно **Save Dump As**

Это диалоговое окно позволяет сохранить текущее состояние проекта в формате, используемом для определения причин его краха.

Меню *Tools*

Меню **Tools** (Сервис), приведенное на рис. П2.77, содержит команды настройки среды Visual Studio.NET и команды доступа к автономным утилитам.

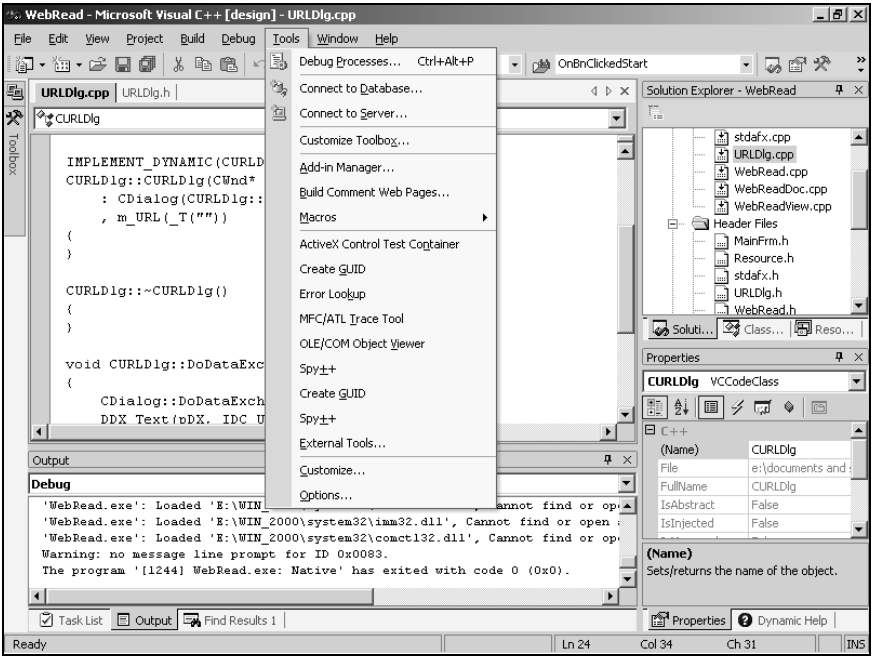


Рис. П2.77. Меню **Tools**

Команда *Tools | Debug Processes*

Данная команда меню эквивалентна рассмотренной выше команде меню **Debug | Processes** (Отладка | Процессы).

Команда *Tools | Connect to Database*

Данная команда меню позволяет подключить приложение к серверу SQL базы данных. При выборе этой команды раскрывается панель **Server Explorer** (Проводник сервера) и появляется диалоговое окно **Data Link Properties** (Свойства связи), как это показано на рис. П2.78.

Команда *Tools | Connect to Server*

Данная команда меню позволяет подключить приложение к новому серверу, который можно указать, используя его имя, адрес HTTP или адрес IP. При выборе этой команды раскрывается панель **Server Explorer** (Проводник сервера) и появляется диалоговое окно **Add Server** (Добавить сервер), приведенное на рис. П2.79.

Команда *Tools | Customize Toolbox*

Данная команда меню позволяет настроить вид панели **Toolbox** (Инструментарий). При ее выборе появляется диалоговое окно **Customize Toolbox** (Настройка инструментария), изображенное на рис. П2.80.

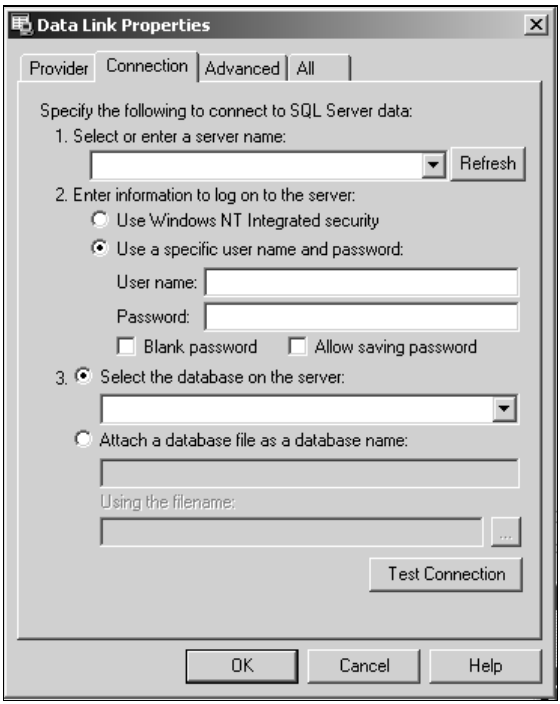


Рис. П2.78. Диалоговое окно **Data Link Properties**

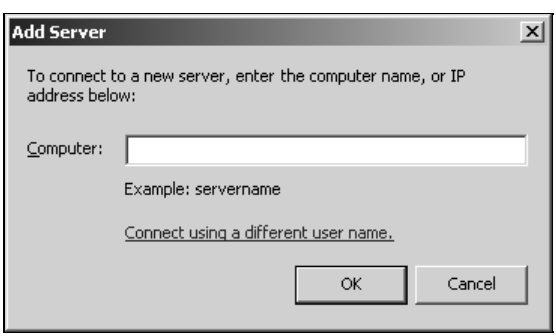


Рис. П2.79. Диалоговое окно **Add Server**

Раскрытая на рис. П2.80 вкладка **COM Controls** позволяет добавить в список ресурсов панели **Toolbox** (Инструментарий) элементы управления COM.

Команда *Tools | Add-in Manager*

Данная команда меню позволяет добавить в среду программирования Visual Studio.NET дополнительные компоненты. При выборе этой команды появляется диалоговое окно **Add-in Manager** (Добавить компонент), изображенное на рис. П2.81.

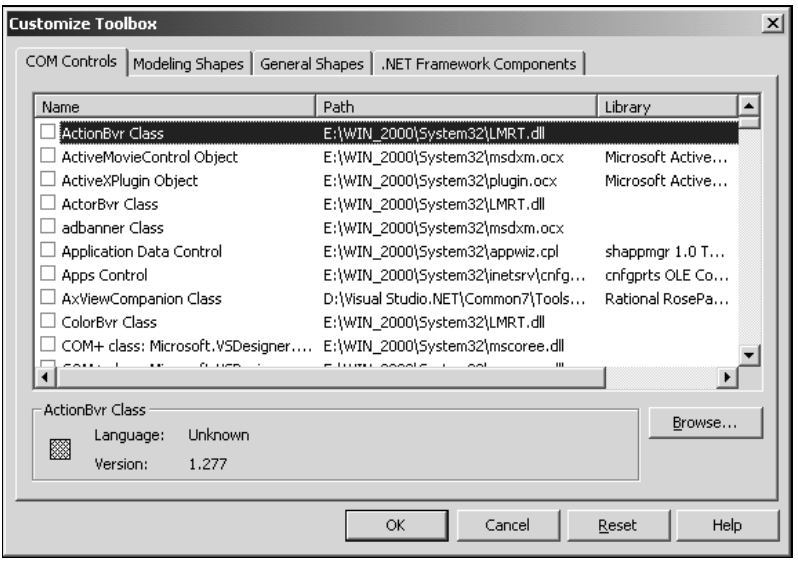


Рис. П2.80. Диалоговое окно **Customize Toolbox**

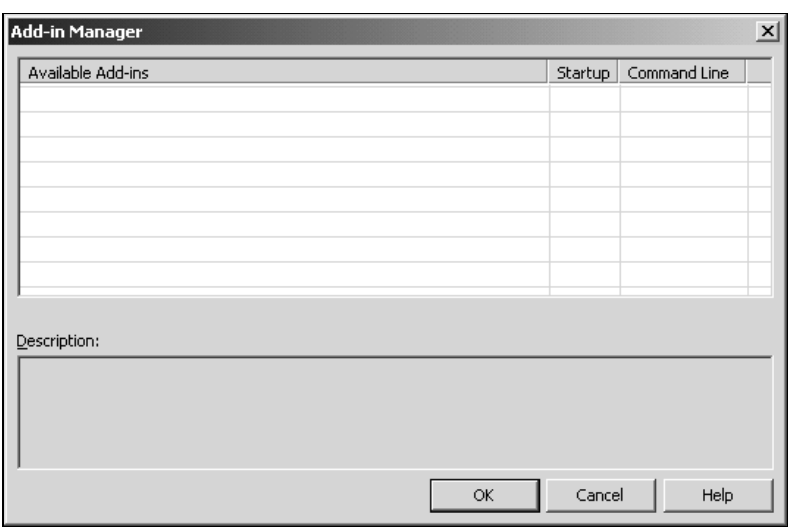


Рис. П2.81. Диалоговое окно **Add-in Manager**

Каждая строка в окне списка данного диалогового окна содержит имя подключенного компонента и определение того, будет ли он подключаться при запуске среды программирования и будет ли он доступен при запуске компилятора из командной строки. Кроме того, это диалоговое окно содержит текстовое поле, в которое будет выводиться краткое описание выделенного компонента, составленное его разработчиком.

Команда *Tools | Build Comment Web Pages*

При выборе данной команды меню появляется диалоговое окно **Build Comment Web Pages** (Создать комментарий Web-страниц), изображенное на рис. П2.82.

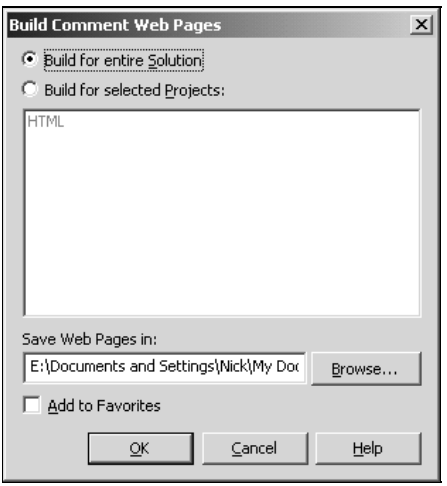


Рис. П2.82. Диалоговое окно **Build Comment Web Pages**

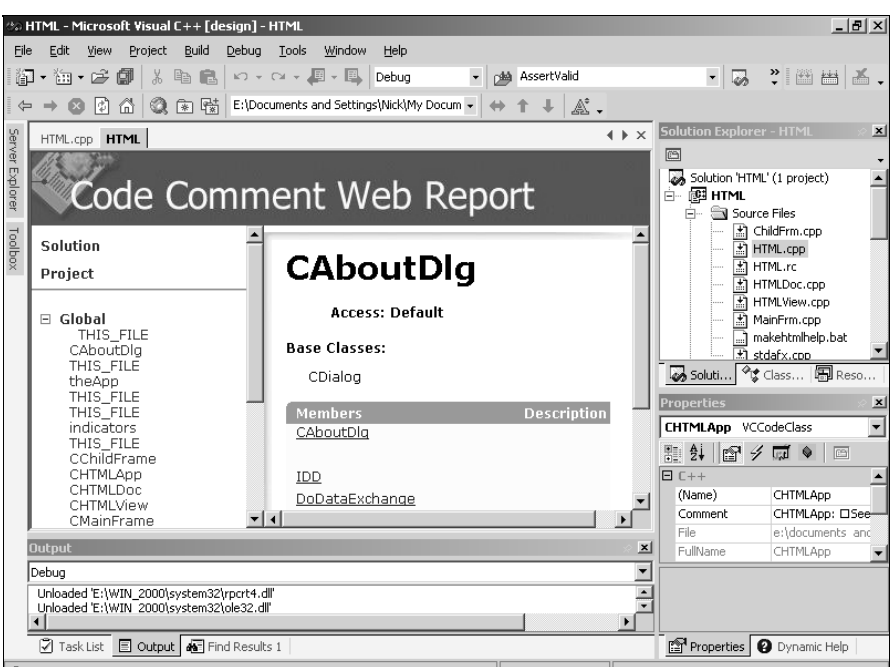


Рис. П2.83. Вывод информации о приложении

Это диалоговое окно позволяет создать набор страниц в формате HTML, содержащих описание всего решения или некоторых составляющих его проектов. Пример такого набора приведен на рис. П2.83.

Команда *Tools | Macros*

Данная команда меню не является терминальной. При ее выборе появляется контекстное меню для работы с макросами, изображенное на рис. П2.84.

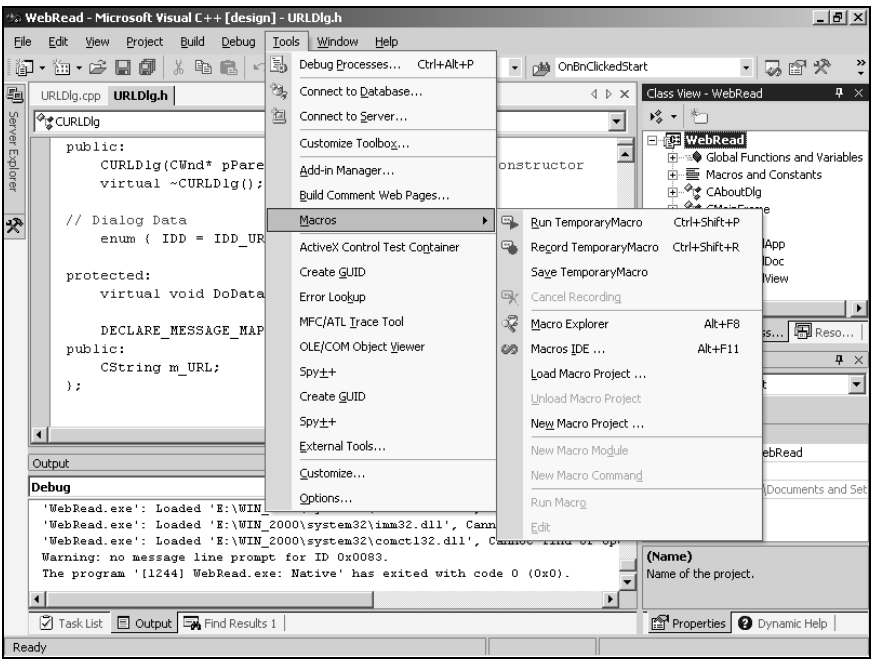


Рис. П2.84. Контекстное меню **Macros**

Данное контекстное меню позволяет записать или воспроизвести простой макрос, или редактировать записанную последовательность нажатия клавиш, добавляя в нее операторы VBScript.

Команда *Tools | ActiveX Control Test Container*

Выбор данной команды меню запускает приложение ActiveX Control Test Container (Проверка контейнера элемента управления ActiveX), используемое для отладки приложений-серверов, создаваемых в среде программирования Visual Studio.NET. Окно этого приложения приведено на рис. П2.85.

Команда *Tools | Create GUID*

Выбор данной команды меню запускает приложение guidgen, окно которого изображено на рис. П2.86.

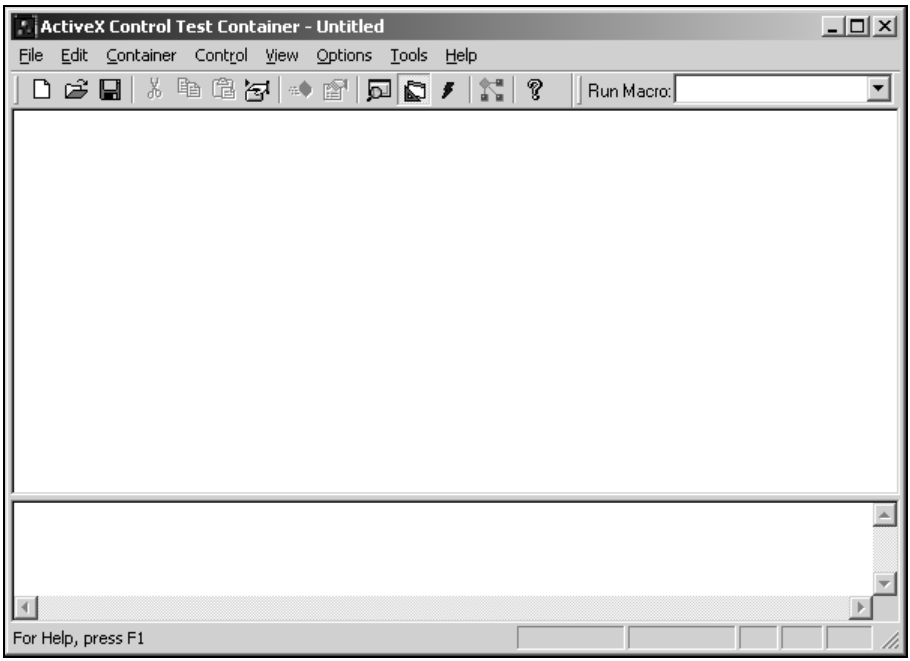


Рис. П2.85. Окно приложения ActiveX Control Test Container

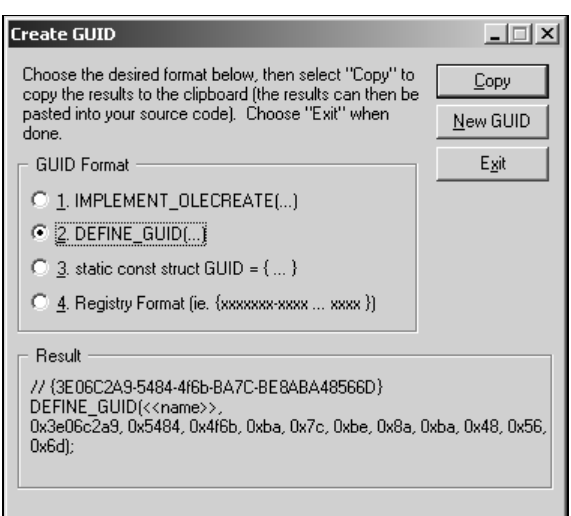


Рис. П2.86. Окно приложения guidgen

Это приложение позволяет создать глобально-уникальный идентификатор, используя для этого различные форматы его представления.

Команда *Tools | Error Lookup*

При выборе данной команды меню появляется диалоговое окно **Error Lookup** (Поиск ошибки), изображенное на рис. П2.87.

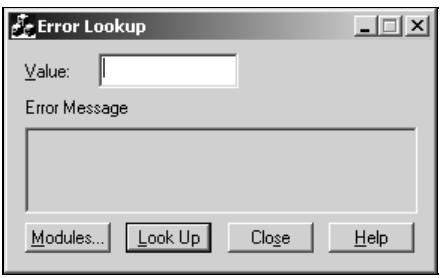


Рис. П2.87. Диалоговое окно **Error Lookup**

Это диалоговое окно позволяет получить текстовое сообщение, связанное с системной ошибкой или ошибкой в модуле, код которой введен в текстовое поле **Value** (Переменная).

Команда *Tools | MFC/ATL Trace Tool*

Выбор данной команды меню запускает приложение MFC/ATL Trace Tool, окно которого изображено на рис. П2.88.

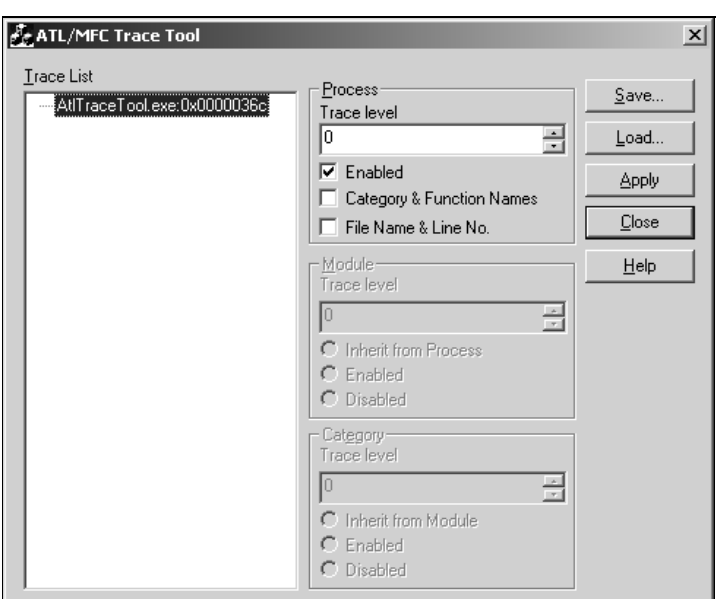


Рис. П2.88. Окно приложения ATL/MFC Trace Tool

Команда *Tools | OLE/COM Object Viewer*

Выбор данной команды меню запускает приложение OLE/COM Object Viewer, окно которого изображено на рис. П2.89.

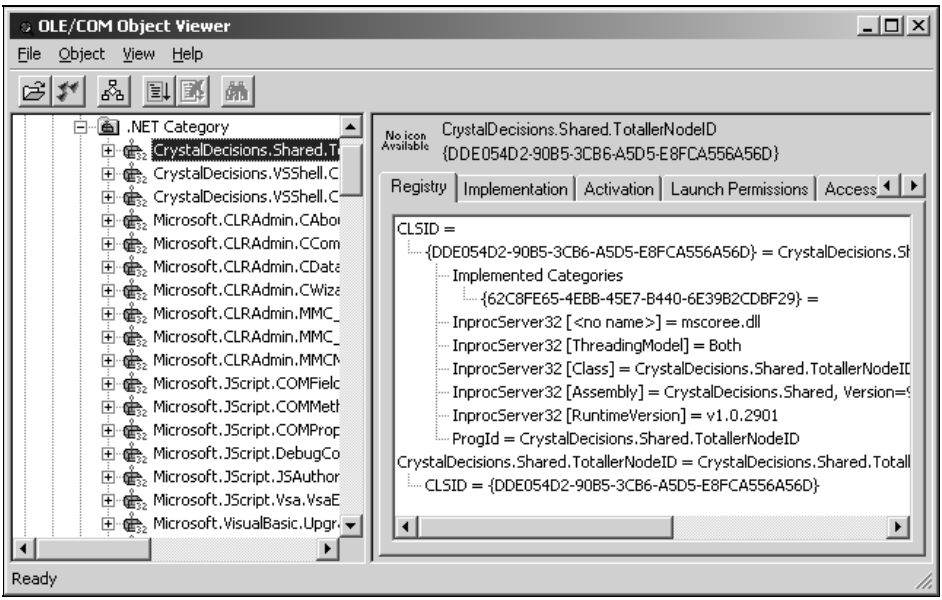


Рис. П2.89. Окно приложения OLE/COM Object Viewer

Это приложение позволяет получить исчерпывающую информацию о зарегистрированных в системе объектах OLE и COM.

Команда *Tools | Spy++*

Выбор данной команды меню запускает приложение Spy++, окно которого изображено на рис. П2.90.

Это приложение позволяет контролировать совместную работу приложений на данном компьютере.

Команда *Tools | External Tools*

При выборе данной команды меню появляется диалоговое окно **External Tools** (Внешний сервис), изображенное на рис. П2.91.

Это диалоговое окно используется для добавления и исключения из меню **Tools** (Сервис) команд вызова внешних приложений, а также для настройки параметров их вызова. Список имеющихся команд для вызова этих приложений выводится в окне списка **Menu Contents** (Содержание меню).

Примечание

Двойной набор команд был задан Microsoft. Поэтому автор не осмелился его изменить.

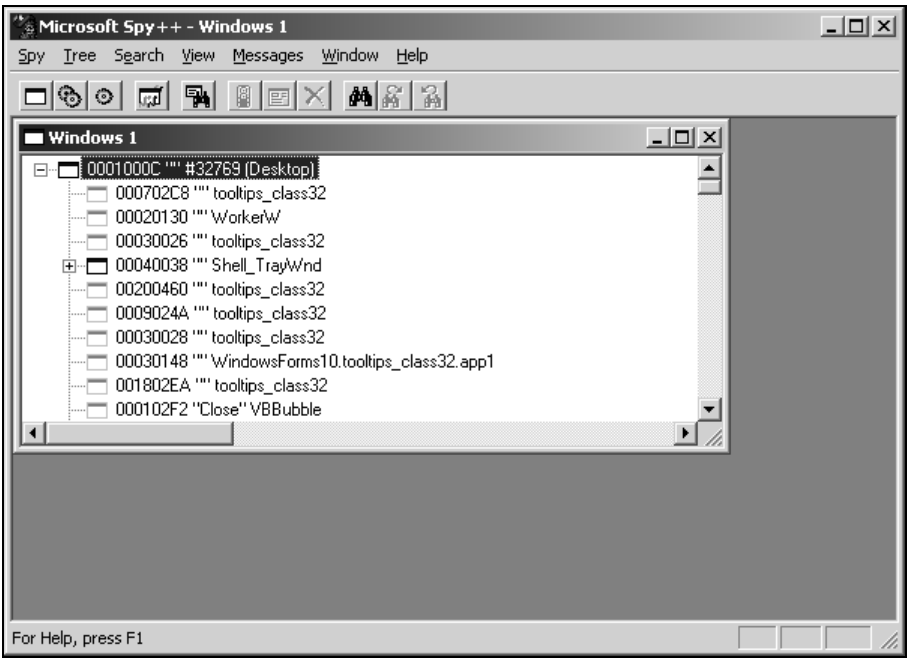


Рис. П2.90. Окно приложения Spy++

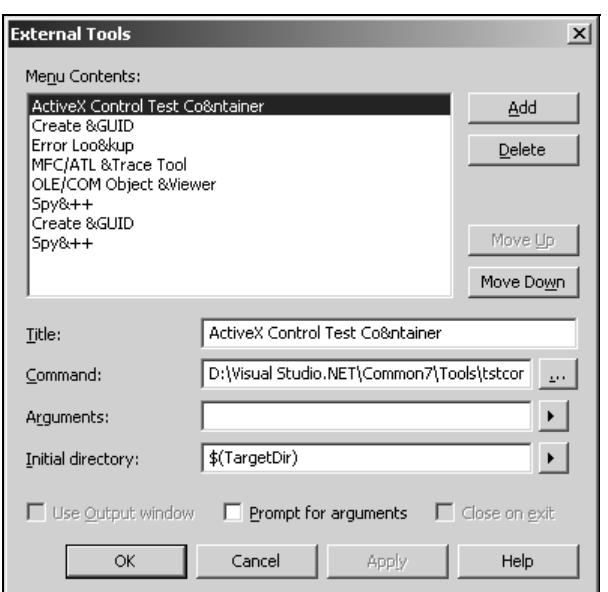


Рис. П2.91. Диалоговое окно **External Tools**

Команда *Tools | Customize*

Выбор данной команды меню выводит на экран диалоговое окно **Customize** (Настройка), изображенное на рис. П2.92.

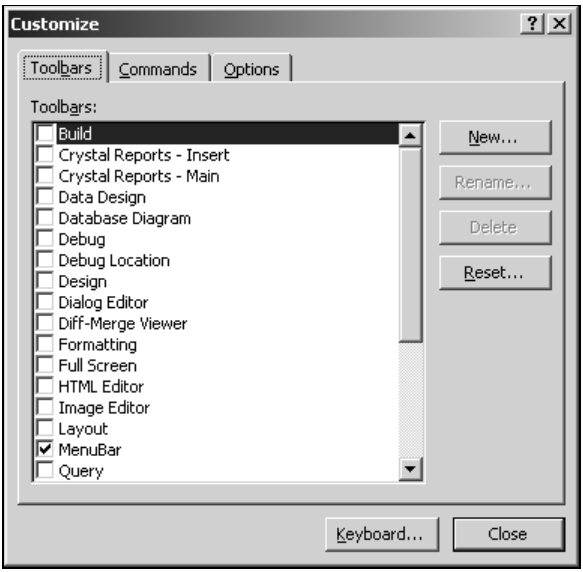


Рис. П2.92. Диалоговое окно **Customize**

Это диалоговое окно позволяет произвести настройку внешнего вида и других параметров среды программирования Visual Studio.NET. Вкладка **Toolbars** (Панели инструментов) данного диалогового окна позволяет определить набор выводимых на экран панелей инструментов. Вкладка **Commands** (Команды) позволяет добавить или удалить кнопки в панели инструментов. Вкладка **Options** (Опции) позволяет настроить режим вывода элементов управления среды программирования: задать размер кнопок панелей инструментов (крупные или нормальные) и режим вывода подсказок к ним, задать прозрачность плавающих панелей инструментов и произвести другие действия. Нажатие кнопки **Keyboard** (Клавиатура) выводит диалоговое окно **Options** (Опции), позволяющее определить режим использования клавиатуры. Среди прочего, оно позволяет задать режим использования функциональных клавиш в соответствии с режимами, используемыми в некоторых популярных средах программирования, и определить комбинации клавиш для выбранных команд.

Команда *Tools | Options*

Выбор данной команды меню выводит на экран диалоговое окно **Options** (Опции), изображенное на рис. П2.93.

Данное диалоговое окно позволяет настраивать различные параметры среды программирования Visual Studio.NET.

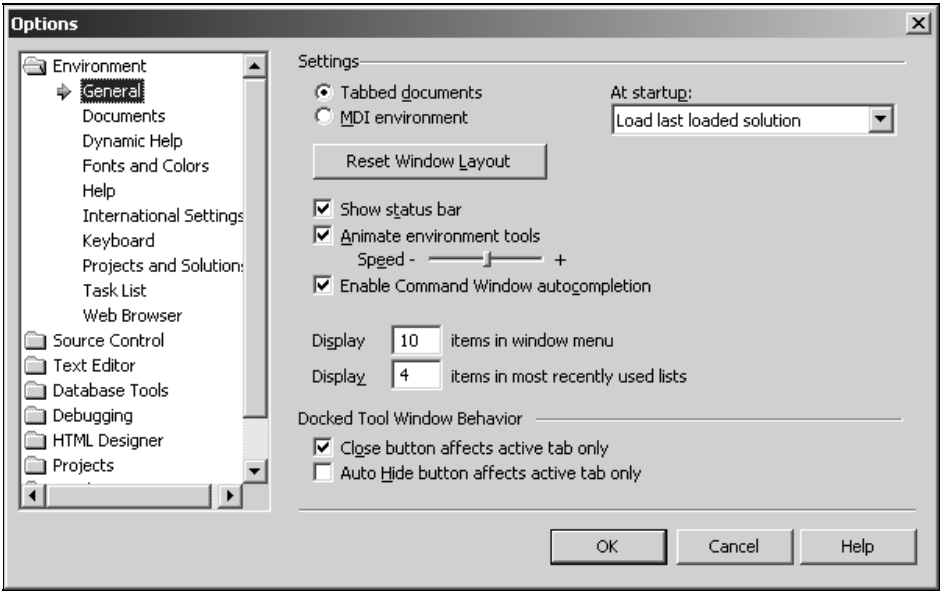


Рис. П2.93. Диалоговое окно **Options**

Меню *Window*

Меню **Window** (Окно), приведенное на рис. П2.94, содержит команды выбора окон и их автоматического упорядочения на экране.

Команда *Window | New Window*

Выбор данной команды меню создает новое окно редактирования, в которое помещается копия файла, которому соответствовало окно редактирования, имевшее фокус ввода в момент выбора данной команды меню. В строке заголовка к имени файла добавляется номер окна (например, **:2**). Если окно, имевшее до этого фокус ввода, не содержало в своем заголовке номер окна, к имени файла в его заголовке добавляется номер **:1**. Изменения, произведенные в одном окне, немедленно отображаются в других окнах.

Команда *Window | Split*

Выбор этой команды меню разделяет по горизонтали окно редактирования текстового файла, имеющее в данный момент фокус ввода, на две панели, как это показано на рис. П2.95. В каждой из этих панелей отображается текст одного и того же файла, но каждая из них имеет независимые друг от друга вертикальную и горизонтальную полосу прокрутки, что позволяет одновременно просматривать различные фрагменты текста этого файла.

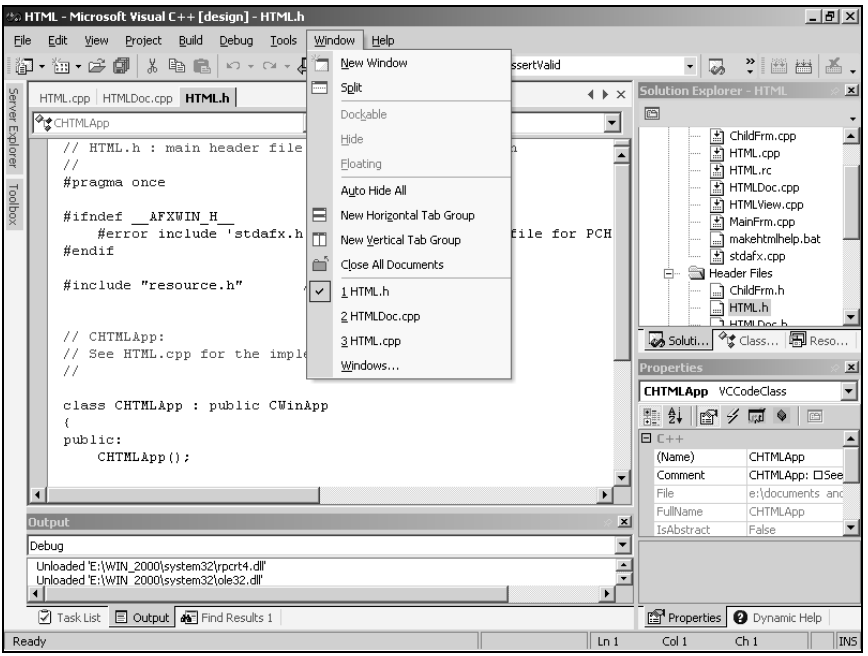


Рис. П2.94. Меню **Window**

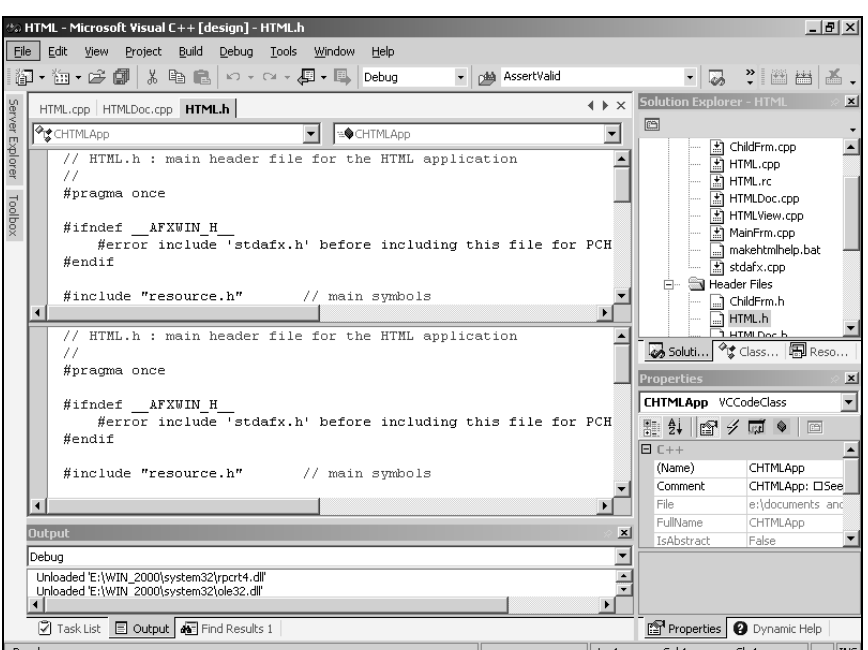


Рис. П2.95. Разделенное окно

Чтобы убрать разделение окна, выберите команду **Window | Remove Split** (Окно | Удалить разделение), появившуюся на месте команды **Window | Split** (Окно | Разделить), или перетащите разделительную полосу между панелями в крайнюю верхнюю позицию.

Команда *Window | Dockable*

Данная команда меню управляет состоянием окон среды программирования Visual Studio.NET, таких как **Workspace** (Рабочая область), **Output** (Окно вывода), **Watch** (Просмотр) и подобных им окон. Если такое окно имеет фокус ввода и данная команда меню отмечена (у нее стоит флажок), то это окно может стать стационарным и отображаться как панель соответствующего окна Visual Studio.NET.

Команда *Window | Hide*

Данная команда меню управляет состоянием окон среды программирования Visual Studio.NET, таких как **Workspace** (Рабочая область), **Output** (Окно вывода), **Watch** (Просмотр) и подобных им окон. При ее выборе окно, имеющее фокус ввода, скрывается (убирается с экрана). После вывода на экран соответствующего окна с использованием меню **View** оно появляется в том месте экрана и с теми установками, с которыми оно было скрыто.

Команда *Window | Floating*

Данная команда меню управляет состоянием окон среды программирования Visual Studio.NET, таких как **Workspace** (Рабочая область), **Output** (Окно вывода), **Watch** (Просмотр) и подобных им окон. При ее выборе окно, имеющее фокус ввода, теряет возможность фиксации (то есть превращения в панель). Если в момент выбора команды это окно было зафиксировано и отображалось как панель другого окна, оно становится плавающим. Если оно уже было плавающим, то оно не претерпевает никаких внешних изменений, но попытки превратить его в панель завершаются безуспешно.

Команда *Window | Auto Hide*

Данная команда меню управляет состоянием окон среды программирования Visual Studio.NET, таких как **Workspace** (Рабочая область), **Output** (Окно вывода), **Watch** (Просмотр) и подобных им окон. При ее выборе окно с вкладками, имеющее фокус ввода, само превращается в панель окна Visual Studio.NET, а ее вкладки становятся панелями, как это показано на рис. П2.96.

Для обратного преобразования нужно сделать активной одну из вкладок бывшего окна и снова выбрать эту команду меню (снять с нее флажок).

Команда *Window | Auto Hide All*

Данная команда меню управляет состоянием окон среды программирования Visual Studio.NET, таких как **Workspace** (Рабочая область), **Output** (Окно вывода), **Watch** (Просмотр) и подобных им окон. При ее выборе все эти окна с вкладками превращаются в панели окна Visual Studio.NET, как это показано на рис. П2.97.

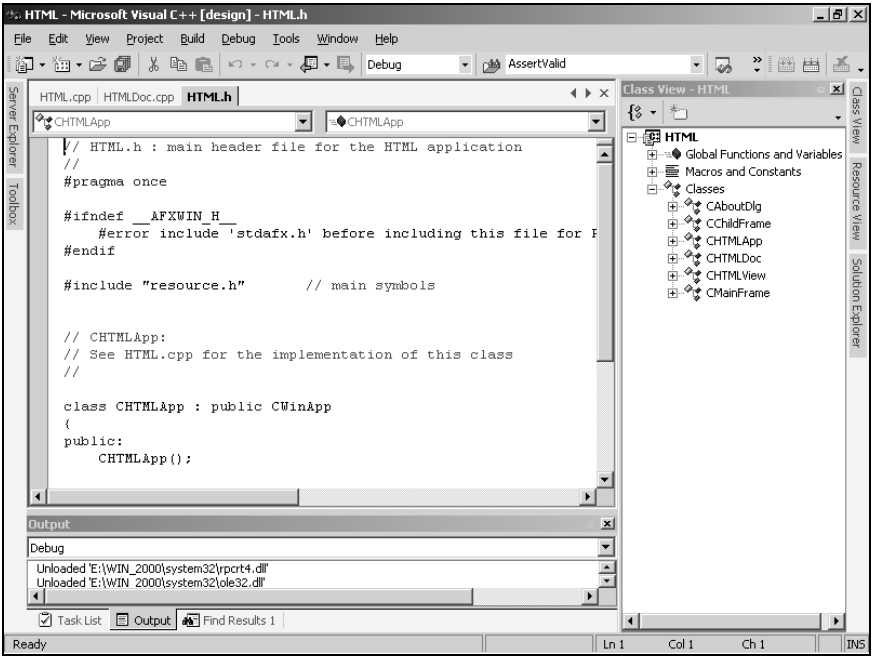


Рис. П2.96. Превращение вкладок в панели

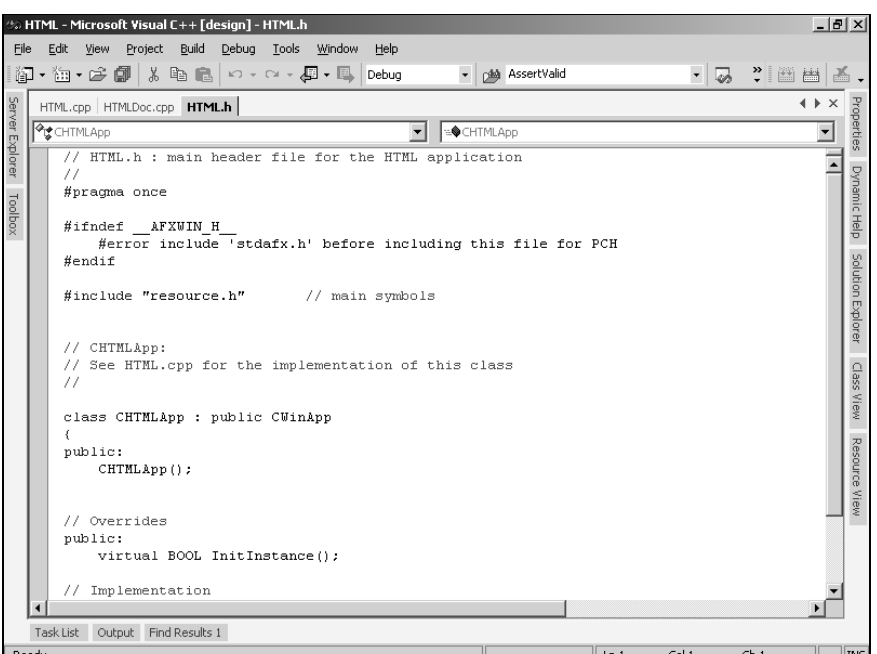


Рис. П2.97. Превращение всех окон в панели

Обратное преобразование каждого окна производится по отдельности, как это было описано в предыдущем разделе.

Команда *Window | New Horizontal Tab Group*

Данная команда меню управляет состоянием окон редактирования среды программирования Visual Studio.NET. При ее выборе активное окно редактирования, содержащее несколько вкладок, разбивается по горизонтали на два независимых окна и в нижнее окно помещается его текущая вкладка, которая при этом исчезает из верхнего окна, как это показано на рис. П2.98.

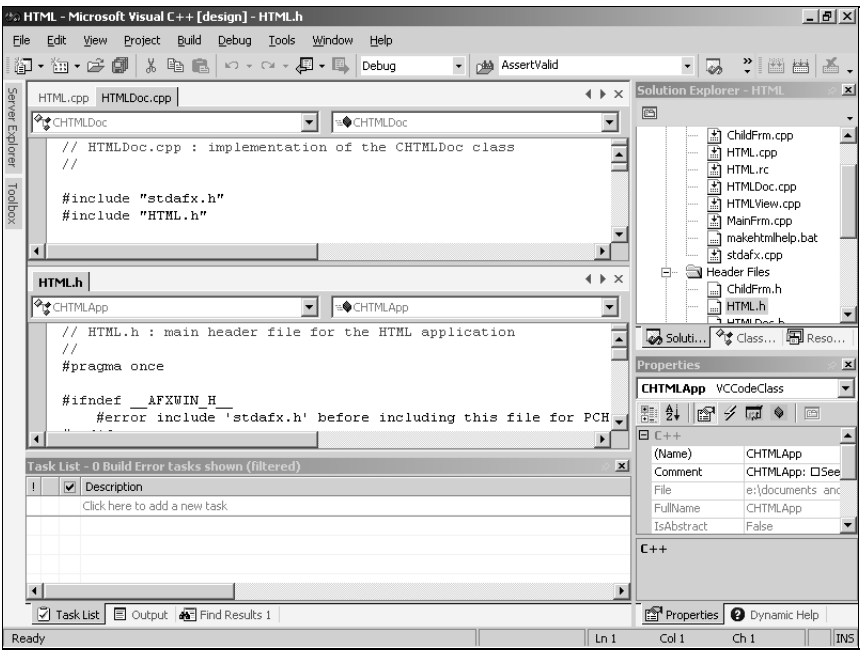


Рис. П2.98. Создание нового горизонтального окна с вкладками

Вкладки можно перетаскивать из одного окна в другое. После того как из окна будет перетащена последняя вкладка, окно уничтожается, а на его место автоматически растягивается соседнее окно.

После создания горизонтального окна с вкладками в меню исчезает команда, позволяющая создать вертикальное окно с вкладками.

Команда *Window | New Vertical Tab Group*

Данная команда меню управляет состоянием окон редактирования среды программирования Visual Studio.NET. При ее выборе активное окно редактирования, содержащее несколько вкладок, разбивается по вертикали на два независимых окна и в правое окно помещается его текущая вкладка, которая при этом исчезает из левого окна, как это показано на рис. П2.99.

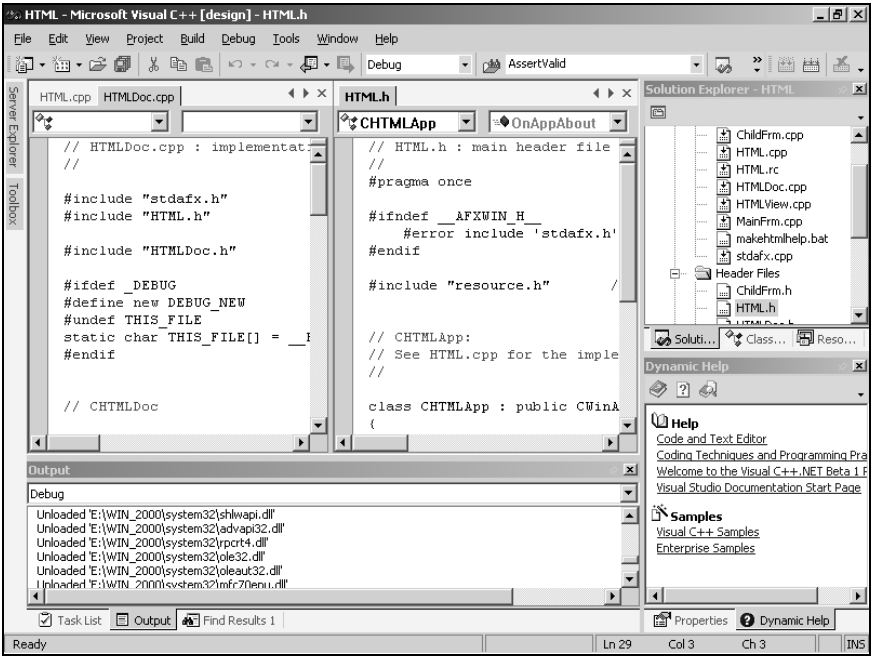


Рис. П2.99. Создание нового вертикального окна с вкладками

Вкладки можно перетаскивать из одного окна в другое. После того как из окна будет перетащена последняя вкладка, окно уничтожается, а на его место автоматически растягивается соседнее окно.

После создания вертикального окна с вкладками в меню исчезает команда, позволяющая создать горизонтальное окно с вкладками.

Команда *Window | Move to Next Tab Group*

Данная команда меню управляет состоянием окон редактирования среды программирования Visual Studio.NET. При ее выборе активная вкладка перемещается в следующее окно. Если это окно является последним, данная команда в меню отсутствует.

Команда *Window | Move to Previous Tab Group*

Данная команда меню управляет состоянием окон редактирования среды программирования Visual Studio.NET. При ее выборе активная вкладка перемещается в предыдущее окно. Если это окно является первым, данная команда в меню отсутствует.

Команда *Window | Close All Documents*

Данная команда меню управляет состоянием окон редактирования среды программирования Visual Studio.NET. Ее выбор приводит к закрытию всех окон документов в приложении.

Список открытых окон

Нижняя часть меню **Window** (Окно) содержит список открытых окон, что позволяет достаточно просто перемещаться между ними. Если количество открытых окон больше десяти, то в данном списке отображаются только десять окон, расположенные в порядке убывания времени последнего обращения к ним пользователя. Для доступа к остальным раскрытым окнам следует выбрать команду меню **Window | Windows** (Окно | Окна).

Примечание

Ценность списка открытых окно меню Windows существенно снижена превращением окон редактирования во вкладки, что позволяет открывать их по ярлыку. Этот список можно рассматривать как рудимент предыдущих версий.

Команда *Window | Windows*

Выбор данной команды меню выводит на экран диалоговое окно **Windows** (Окна), изображенное на рис. П2.100.

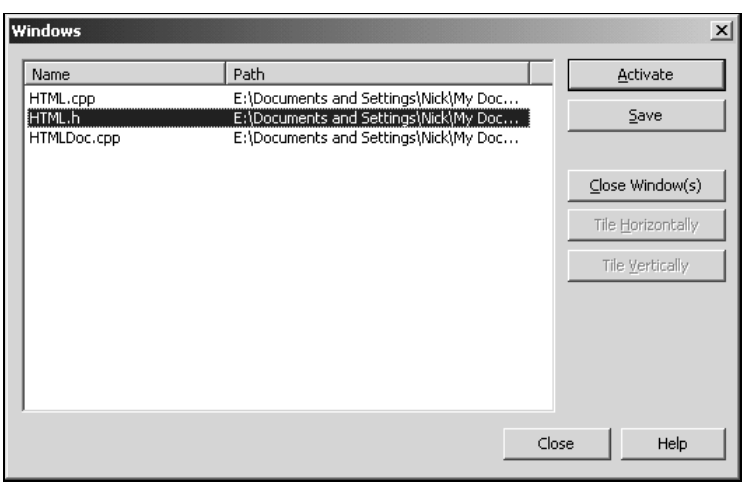


Рис. П2.100. Диалоговое окно **Windows**

Данное диалоговое окно позволяет передавать фокус ввода окну, выделенному в окне списка, а также сохранять содержимое окон, выделенных в данном окне списка, и закрывать эти окна.

Меню *Help*

Меню **Help** (Справка), приведенное на рис. П2.101, содержит команды обращения к справочной системе Visual Studio.NET. Как уже говорилось выше, эта справочная система может выводиться в окне среды программирования Visual Studio.NET или в окнах отдельного приложения справочной системы. В данном случае нами рассматривается установленный по умолчанию режим вывода в окна среды Visual Studio.NET.

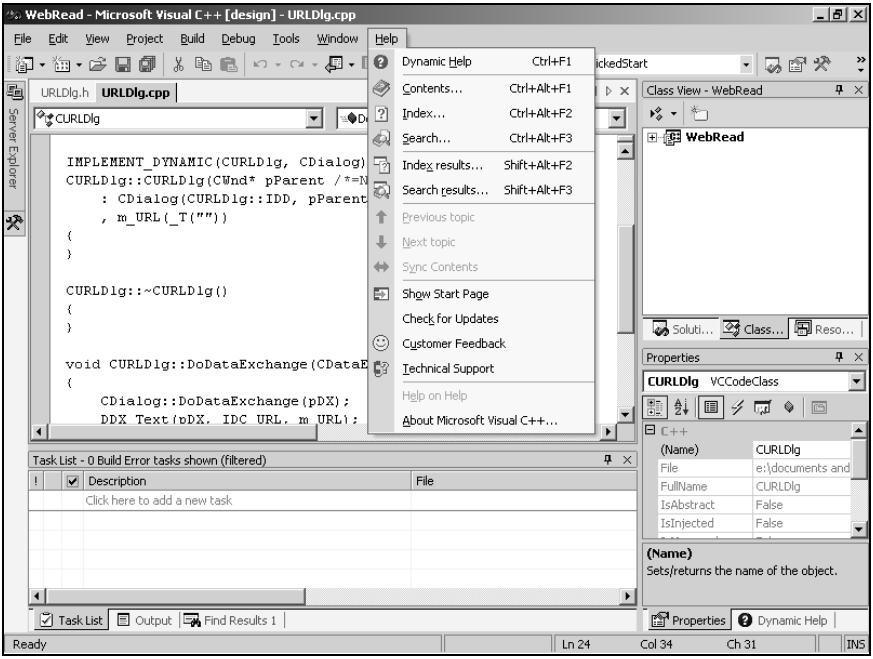


Рис. П2.101. Меню **Help**

Команда *Help | Dynamic Help*

Эта команда меню используется для получения краткой справки по элементу среды программирования, с которым в данный момент производятся некоторые действия. Информация выводится в окно **Dynamic Help** (Оперативная справка), изображенное на рис. П2.102. Это окно, как правило, выводится в виде вкладки.

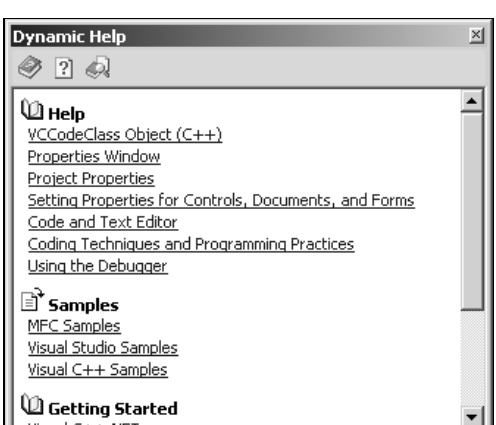


Рис. П2.102. Окно **Dynamic Help**

Команда *Help | Contents*

При выборе данной команды меню на экран выводится окно **Contents** (Содержание), изображенное на рис. П2.103. В зависимости от положения переключателя **Show Help** (Показать справку) на первой странице Visual Studio.NET это окно выводится в интегрированной среде разработки или в специальном справочном приложении. В данном случае окно выводится в специальном приложении, поскольку, по-моему, это удобнее.

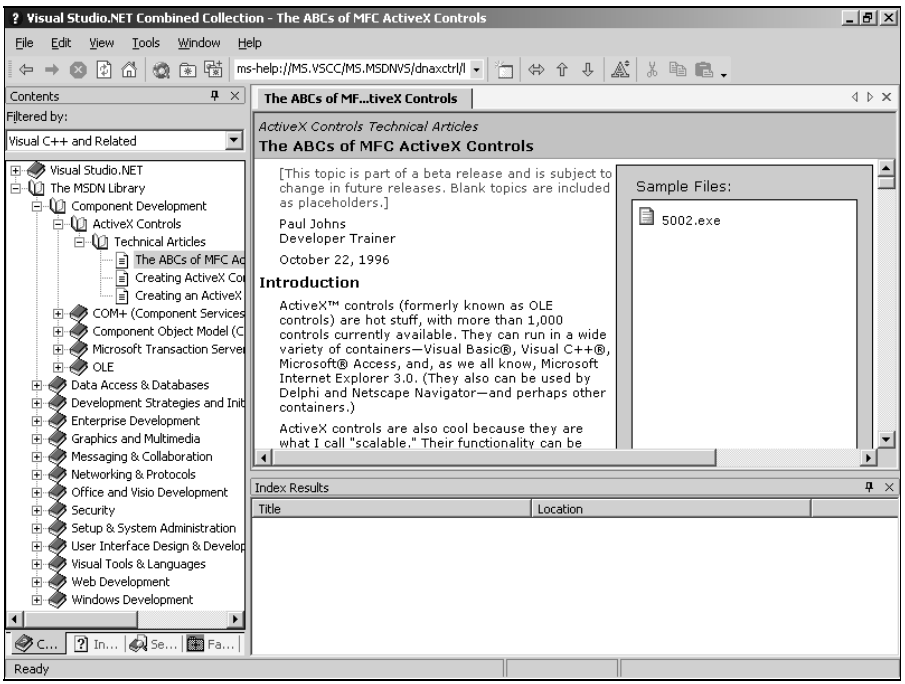


Рис. П2.103. Окно **Contents**

Это окно позволяет получить доступ к интерактивной справке системы программирования Visual Studio.NET.

Команда *Help | Index*

При выборе данной команды меню на экран выводится окно **Index** (Предметный указатель), изображенное на рис. П2.104. Это окно так же, как и описанное выше окно **Contents** (Содержание), может выводиться в интегрированной среде разработки или в специальном справочном приложении. Режим вывода этого окна определяется тем же переключателем, что и режим вывода окна **Contents** (Содержание).

Данное окно позволяет произвести поиск по ключу в интерактивной справке системы программирования Visual Studio.NET.

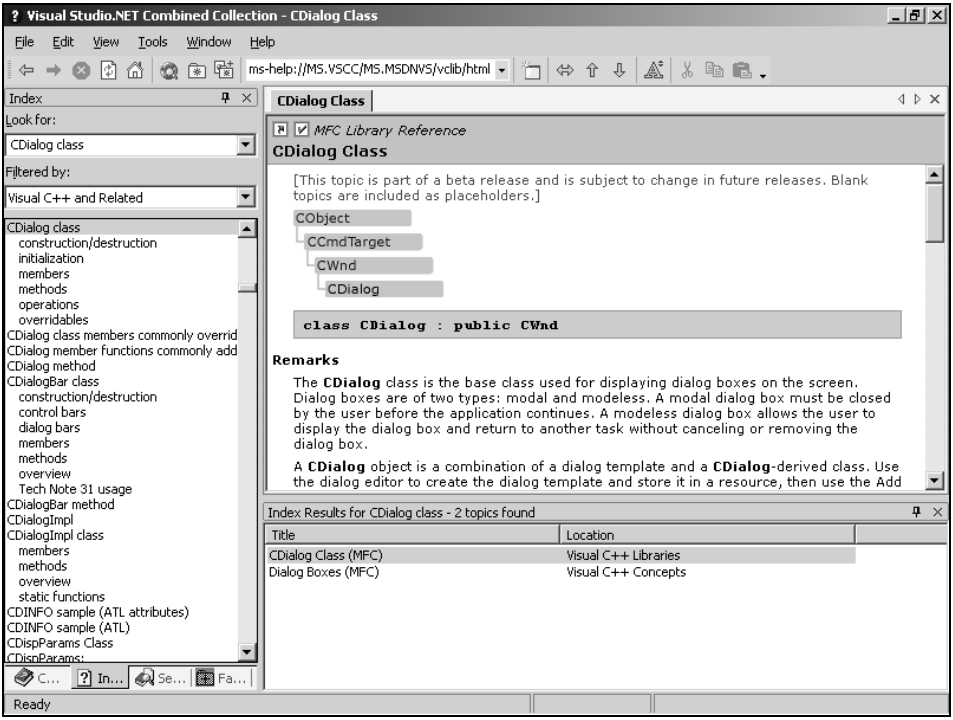


Рис. П2.104. Окно **Index**

Команда *Help | Search*

При выборе данной команды меню на экран выводится окно **Search** (Поиск). Это окно так же, как и описанное выше окно **Contents** (Содержание), может выводиться в интегрированной среде разработки или в специальном справочном приложении. Режим вывода этого окна определяется тем же переключателем, что и режим вывода окна **Contents** (Содержание).

Данное окно позволяет произвести поиск текста в интерактивной справке системы программирования Visual Studio.NET. Результат поиска в этом окне приведен на рис. П2.105.

Команда *Help | Index results*

Если результаты поиска, проведенного по ключу, заданному в окне **Index** (Предметный указатель), не позволили однозначно определить выводимую тему справки, среда программирования Visual Studio.NET выведет окно **Index Results** (Найденные разделы). Это окно расположено в нижней правой части рис. П2.104 и содержит темы, из которых следует сделать выбор.

Если справочная информация выводится в специальном приложении, при выборе команды меню **Help | Index results** (Справка | Найденные разделы) открывается это приложение.

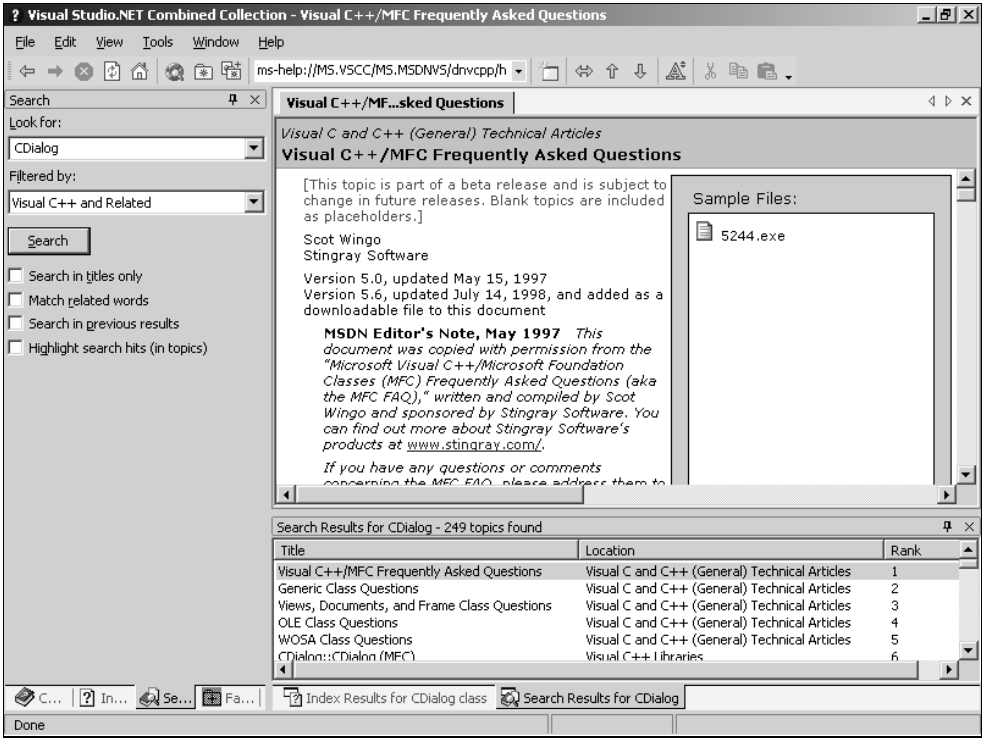


Рис. П2.105. Окно **Search**

Команда *Help | Search results*

Результаты поиска текста, заданного в окне **Search** (Поиск), выводятся в окно **Search Results** (Результаты поиска), расположенное в нижней правой части рис. П2.105. Действия, предпринимаемые при выборе данной команды, аналогичны действиям, предпринимаемым при выборе команды **Help | Index results** (Справка | Найденные разделы).

Остальные команды меню

Команды меню **Help | Previous topic** (Справка | Предыдущий раздел), **Help | Next topic** (Справка | Следующий раздел) и **Help | Sync Contents** (Справка | Синхронизировать содержание) используются для перемещения между темами справочной системы в том случае, если она выводится в интегрированной среде разработки Visual Studio.NET. Остальные команды меню используются для связи с центрами технической поддержки Microsoft и получения от них необходимой информации и новых версий продукта. Поскольку объем программы поддержки пользователей сервисными центрами Microsoft существенно сокращен и не удовлетворяет минимальным потребностям рядового пользователя, эти команды описываться не будут.

Окна Visual Studio.NET

При описании команд меню часто упоминались различные стандартные окна среды программирования Visual C++. Рассмотрим подробнее некоторые из них.

Окно *Solution Explorer*

Окно **Solution Explorer** (Проводник решения), изображенное на рис. П2.106, используется для работы с файлами, включенными в решение. Файлы данного окна сгруппированы по проектам, а в пределах проекта — по папкам. Существуют отдельные папки для файлов реализации (**Source Files**), файлов заголовков (**Header Files**), файлов ресурсов (**Resource Files**) и других типов файлов. Двойной щелчок левой кнопкой мыши по имени файла приводит к открытию окна редактирования данного файла. Нажатие клавиши <Delete> приводит к удалению выделенного файла из проекта.

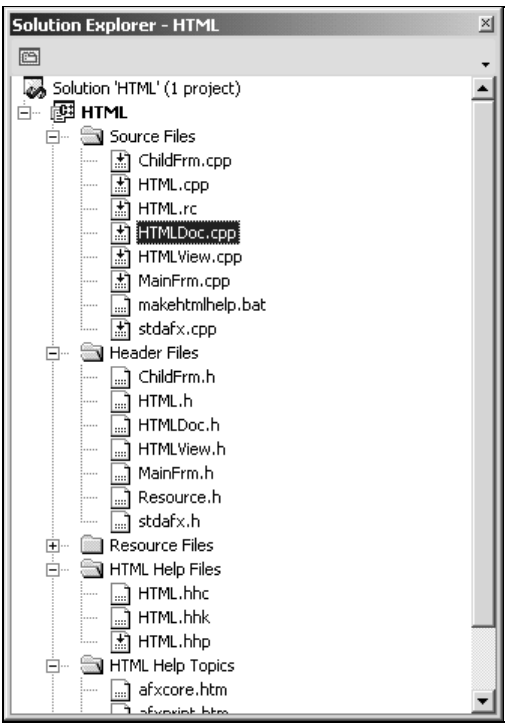


Рис. П2.106. Окно **Solution Explorer**

Окно *Class View*

Окно **Class View** (Просмотр класса), изображенное на рис. П2.107, используется для автоматизации работы с классами приложения.

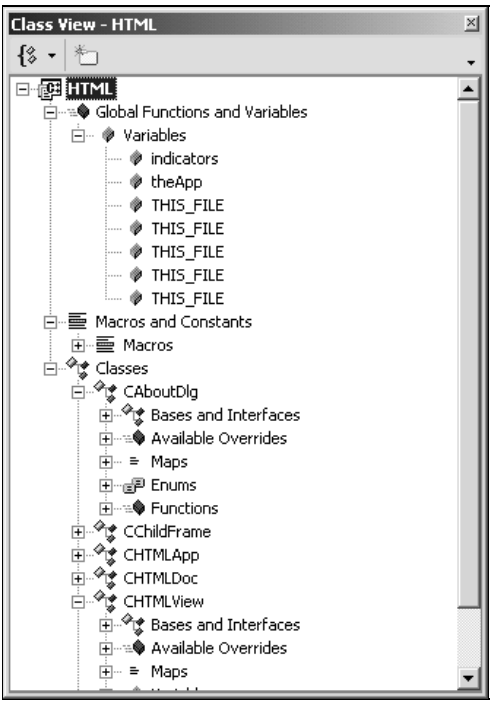


Рис. П2.107. Окно **Class View**

Верхним уровнем иерархии в данном окне являются папки проекта, содержащие папки составляющих его глобальных функций и переменных, макросов и классов. Каждая из этих папок, в свою очередь, содержит папки включенных в нее типов, а также значки членов данного типа, если это папка какого-либо типа. Защищенные члены класса (`protected`) отмечаются ключом, а закрытые члены класса (`private`) — замком.

При щелчке правой кнопкой мыши по папке данного меню появляется контекстное меню, изображенное на рис. П2.108.

Рассмотрим наиболее часто используемые команды данного меню:

- **Build** (Создать) — компилирует приложение, пропуская уже откомпилированные фрагменты. Данная команда появляется в меню, если выделено имя приложения;
- **Rebuild** (Создать заново) — полностью перекомпилирует приложение. Данная команда появляется в меню, если выделено имя приложения;
- **Clean** (Очистить) — удаляет файлы, созданные при компиляции данного приложения. Данная команда появляется в меню, если выделено имя приложения;
- **Link** (Компоновать) — компонует приложение. Данная команда появляется в меню, если выделено имя приложения;
- **Go to Definition** (Перейти к описанию) — действие этой команды зависит от выделенного элемента: если выделено имя класса — открывается его файл заголов-

ка и текстовый курсор помещается на первую строку заголовка этого класса, если же выделена функция — раскрывается файл, где реализована данная функция, и текстовый курсор помещается на первую строку ее реализации;

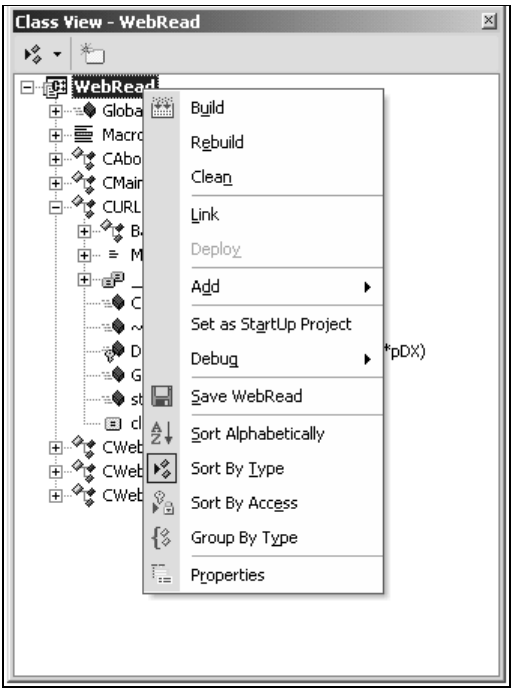


Рис. П2.108. Окно **Class View** с раскрытым контекстным меню

- **Go to Declaration** (Перейти к объявлению) — если выделена функция, эта команда меню открывает файл, в котором она объявлена и помещает текстовый курсор в объявление данной функции;
- **Add** (Добавить) — данная команда не является терминальной. При ее вызове появляется контекстное меню, содержащее команды, набор которых зависит от выделенного элемента. Команды данного меню дублируют одноименные команды меню **Project** (Проект);
- **Copy** (Копировать) — копирует в буфер обмена полное имя выделенного компонента;
- **Sort Alphabetically** (Упорядочить по алфавиту) — упорядочивает элементы в папках и сами папки в алфавитном порядке;
- **Sort By Type** (Упорядочить по типу) — упорядочивает элементы в папках по типу;
- **Sort By Access** (Упорядочить по режиму доступа) — упорядочивает элементы в папках по мере ужесточения режима доступа к ним;

- **Group By Type** (Группировать по типу) — группирует элементы по типу (классы, интерфейсы, методы, поля, индексаторы и т. д.);
- **Properties** (Свойства) — выводит окно **Properties** (Свойства), которое будет рассмотрено ниже.

Окно *Properties*

Окно **Properties** (Свойства), изображенное на рис. П2.109, используется для вывода информации о различных компонентах программы. Основным назначением данного окна является работа с ресурсами приложения. Это окно позволяет внести изменения в установленные по умолчанию свойства этих элементов управления и, таким образом, произвести их настройку в соответствии с требованиями создаваемого приложения.

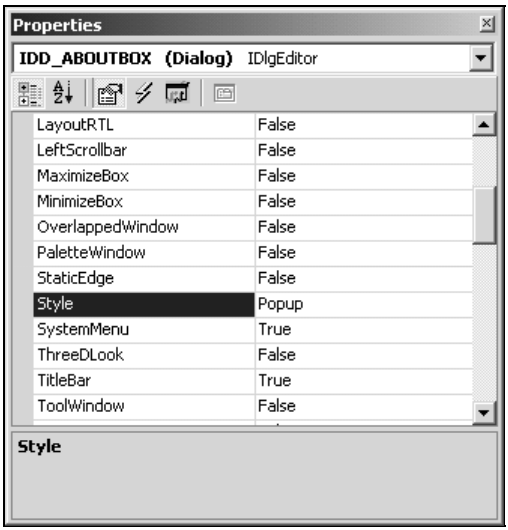


Рис. П2.109. Окно **Properties**

Подробнее использование этого окна при создании диалоговых окон приведено в *главе 5* и других главах, в которых говорится о создании приложений, использующих диалоговые окна.

Окно *Watch*

Окно **Watch** (Просмотр), изображенное на рис. П2.110, появляется в процессе выполнения программы и служит для получения информации о значениях переменных, выбранных пользователем. Если тип переменной, содержащейся в данном окне, является ссылкой на объект, то эта переменная выводится в виде папки, позволяющей получить доступ к полям данного объекта.

Помимо окна **Watch** (Просмотр) при запуске приложения на исполнение также выводятся окна **Autos** (Автоматически выбранные переменные) и **Locals** (Локальные пере-

менные), имеющие одинаковую с ним структуру полей. В первом из них выводятся значения переменных, выбираемых средой программирования, а во втором — значения локальных переменных. Подробнее работа с этими окнами описана в *главе 14*.

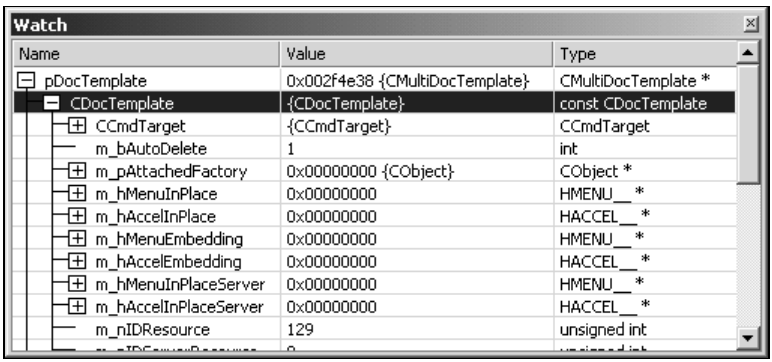


Рис. П2.110. Окно **Watch**

Окно *Breakpoints*

Окно **Breakpoints** (Точки останова), изображенное на рис. П2.111, появляется в процессе выполнения программы и используется для работы с точками останова.

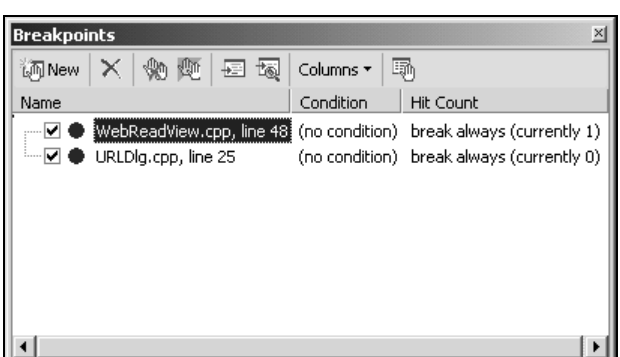


Рис. П2.111. Окно **Breakpoints**

Для изменения свойств точки останова, указанных в этом окне, достаточно щелкнуть на нем правой кнопкой мыши и выбрать в появившемся контекстном меню команду **Properties** (Свойства). Появится диалоговое окно **Breakpoint Properties** (Свойства точек останова), изображенное на рис. П2.112. В этом окне будут отражены все текущие установки для данной точки останова.

Подробнее работа с этим окном описана в *главе 14*.

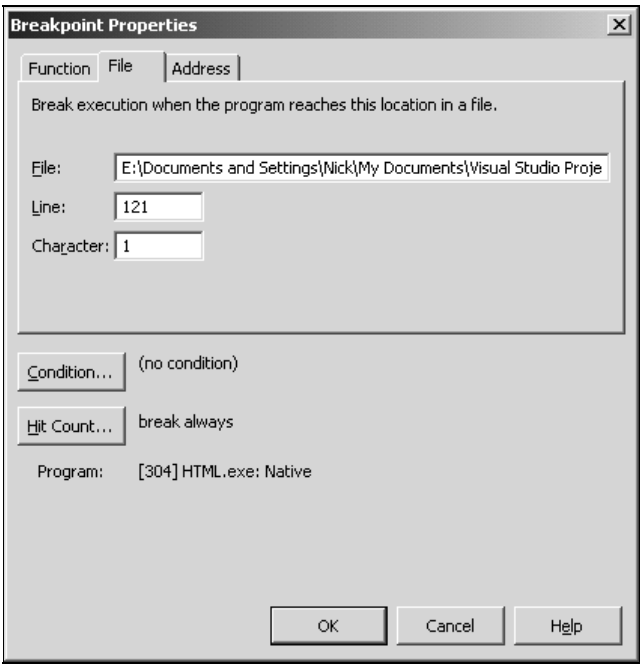
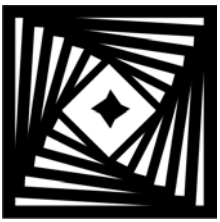


Рис. П2.112. Окно **Breakpoint Properties**

Приложение 3



Описание дискеты

На дискете находится самораспаковывающийся архив samples.exe с файлами примеров.

Каждый пример распаковывается в отдельный каталог и представляет собой полноценный проект, который можно загружать в среду разработки и компилировать. Имя каждого каталога совпадает с именем содержащегося в нем проекта.

Ниже приведен список проектов с указанием глав, к которым они относятся:

- Brush — приложение, демонстрирующее принципы работы с кистью;
- Console — приложение, демонстрирующее принципы работы с консолью;
- DateTime — приложение, демонстрирующее принципы работы с календарем;
- DDB — приложение, демонстрирующее принципы работы с аппаратно-зависимыми битовыми образами;
- DemoLib — приложение, использующее библиотеки Library и Extension;
- DIB — приложение, демонстрирующее принципы работы с аппаратно-независимыми битовыми образами;
- Dialog — диалоговое приложение. Описывается в *главах 1* и *3*;
- EditApp — простейший текстовый редактор;
- Extension — пример библиотеки расширения MFC;
- Help — приложение, демонстрирующее принципы работы со справочной системой приложения;
- HTML — приложение, демонстрирующее возможность преобразования проекта справки WinHelp в проект справки HTML Help;
- Library — пример регулярной библиотеки динамической компоновки (DLL);
- Line — приложение, демонстрирующее принципы работы с перьями;
- List — приложение, демонстрирующее принципы работы с окном списка;
- MDI — многооконное приложение. Описывается в *главах 1* и *2*;
- Print — приложение, демонстрирующее принципы прокрутки изображения в окне и вывода информации на печатающее устройство;
- Progress — приложение, демонстрирующее принципы работы с линейным индикатором и линейным регулятором;
- Rich — текстовый редактор с расширенными возможностями;
- Sheet — приложение, демонстрирующее принципы работы с вкладками и мастерами. Описывается в *главе 3*;

- Sync — приложение, демонстрирующее принципы синхронизации потоков;
- Text — приложение, демонстрирующее принципы вывода текстовой информации на экран;
- Tool — приложение, демонстрирующее принципы работы с панелью инструментов и со строкой состояния;
- WebRead — приложение, позволяющее считывать содержимое Web-страницы в формате HTML.

Примечание

На дискете не размещены приложения Listhdr, MtGDI и Mutexes, являющиеся неотъемлемой частью руководства по библиотеке MFC и электронной документации, поставляемой вместе с этим руководством. Файлы приложений поставляются с библиотекой MSDN Library Visual Studio 6.0.

Кроме того, на дискете приведено подробное описание классов и функций, встречающихся в примерах, рассмотренных в данной книге.

Предметный указатель

А

Атрибуты права доступа 607

Б

Библиотека динамической компоновки 538
Битовый образ 195
  аппаратно-зависимый 195
  аппаратно-независимый 202
Блок:
  catch 317
  try 317
Бордюр 623

В

Вкладка:
  Server Explorer 659
  Toolbox 15, 661

Г

Графический объект 163

Д

Демонстрационное приложение:
  Brush 186
  Console 21
  DateTime 125
  DDB 195
  DemoLib 579
  Dialog 15, 57, 149
  DIB 202
  EditApp 236
  Extension 562
  Help 415
  HTML 487
  Library 555
  Line 181
  List 101
  MDI 18, 25, 43
  Print 286
  Progress 110
  RichApp 244
  Sheet 82
  Sync 369
  Text 176, 208
  Tool 256
  WebRead 588
Диалоговое окно:
  Add Class 555, 591, 666
  Add Existing Item 487, 669
  Add Member Function Wizard 666
  Add Member Variable Wizard 70, 105, 666
  Add New Item 668
  Add Resource 563, 668
  Add Server 680
  Add Web Reference 670
  Add-in Manager 681
  Advanced Save Options 632
  Batch Build 672
  Breakpoint Properties 516, 705
  Build Comment Web Pages 683
  Configuration Manager 673

(окончание рубрики см. на стр. 710)

Диалоговое окно *(окончание)*:
Customize 689
Customize Toolbox 680
Data Link Properties 680
Error Lookup 686
Exceptions 676
External Tools 687
Find 638
Find in Files 642
Find Symbol 645
Generic Class Wizard 558
Go To 647
Insert File 648
MFC Application Wizard 255, 285
MFC Class Wizard 565, 591
MFC DLL Wizard 555
New Breakpoint 677
New File 439, 500, 627
New Project 555, 625
New Symbol 274
Open 229
Open File 629
Open Project 628
Open Project From Web 628, 630
Open Solution 631
Open With 629
Options 493, 689
Page Setup 634
Paste Special 245
Path or URL 496
Print 634
Processes 675
Property Pages 321, 524, 571, 664
QuickWatch 522, 677
Replace 641, 642
Replace in Files 644
Resource Symbols 274
Run 232
Save As 229
Save Dump As 679
Save File As 501, 632
Selected Components 646
Table of Contents Entry 494
Topics Found 399
VS Wizards Select File 240
Windows 696

И

Инкапсуляция 602
Интерфейс:
графических устройств 34, 162
приложения:
многооконный 17
однооконный 19
Исключения 317

К

Карты:
дескрипторов 153
отображений 344
сообщений 138
Класс:
CArchive 29
CArchiveException 323
CArray 348
CBitmap 200
CBrush 125
CClientDC 164
CCmdTarget 138
CCriticalSection 369
CDaoException 323
CDBException 323
CDC 164
CDocument 26
CDumpContext 29, 528
CDynLinkLibrary 548
CEdit 236
CEditView 597
CEvent 369
CFile 218
CFileException 323
CFont 308
CFrameWnd 41
CInternetSession 597
CList 348
CMap 348, 357
CMapPtrToPtr 349
CMapPtrToWord 349
CMapStringToOb 349
CMapStringToPtr 349
CMapWordToPtr 349

CMemoryException 323
CMemoryState 537
CMultiDocTemplate 40
CMultiLock 369
CMutex 369
CNotSuportedException 323
CObArray 349
CObject 347
CObList 349
COleDispatchException 323
COleException 323
CPaintDC 164
CPen 185
CPrintInfo 308
CPtrArray 347
CPtrList 347
CResourceException 323
CRgn 125
CRichEditView 244
CRuntimeClass 548
CSemaphore 369
CSingleDocTemplate 40
CSingleLock 369
CSize 292
CSpinButtonCtrl 81
CTime 134
CToolbar 269
CTypedPtrArray 349
CTypedPtrList 349
CTypedPtrMap 349
CUserException 323
CView 31
CWinApp 38, 217, 361
CWindowDC 164
CWinThread 38, 361
деструктор 608
дружественная функция 614
иерархия 603
конструктор 608
синхронизации доступа 369
синхронизации работы 369
Кнопка панели инструментов:
About 255
Add New Item 624
Class View 624
Copy 255, 624
Cut 255, 624
Find in Files 624
Navigate Backward 624
Navigate Forward 624
New 255
New Project 624
Open 255
Open File 624
Paste 255, 624
Print 255
Properties Window 624
Redo 624
Save 255, 624
Save All 624
Solution Explorer 624
Start 624
Toolbox 624
Undo 624
Колонтитулы 308
Команда:
HELP_CONTENTS 397
HELP_CONTEXTMENU 398
HELP_CONTEXTPOPUP 398
HELP_FINDER 397
HELP_INDEX 397
HELP_KEY 399
HELP_MULTIKEY 399
HELP_PARTIALKEY 399
HELP_QUIT 402
HELP_SETWINPOS 402
HELP_TCARD 402
HELP_WM_HELP 398
Команды меню:
Add:
Add Variable 70
Build:
Batch Build 672
Build 671
Build Solution... 562, 573
Clean 672
Compile 673
Configuration Manager 673
Deploy 673
Rebuild All 671
Debug:
Apply Code Changes 675
Break 675
Clear All Breakpoints 679
Continue 513

(продолжение рубрики см. на стр. 712)

Команды меню *(продолжение)*:
Detach All 675
Disable Breakpoint 679
Exceptions 676
New Breakpoint 677
Processes 675
QuickWatch 521, 677
Restart 513, 675
Save Dump As 679
Start 79
Start 675
Step Into 514, 676
Step Out 514, 677
Step Over 513, 676
Stop Debugging 513, 675
Windows 673
Windows Breakpoints 515
Edit:
Bookmarks 650
Copy 638
Cut 637
Delete 638
Find and Replace 638
Go To 647
Insert File As Text 647
IntelliSense 655
Outlining 651
Paste 638
Redo 637
Select All 638
Undo 636
File:
Add Existing Item 630
Advanced Save Options 632
Exit 635
New 625
New File... 500
New Project... 555
New Item 630
Open 628
Open File... 579
Page Setup 634
Print 634
Print Preiew 294
Recent Files 635
Recent Projects 635
Save 632
Save All 633
Save As 632
Save As... 501
Source Control 633
Help:
Contents 698
Dynamic Help 697
Index 698
Index results 699
Search 699
Search results 700
Show Start Page 618
Insert Dialog 239
Project:
Add Class 666
Add Existing Item 669
Add Existing Item... 487
Add Function 666
Add New Item 668
Add Resource 668
Add Variable 666
Add Web Reference 670
New Folder 669
Reload Project 670
Set as StartUp Project 670
Unload Project 669
Tools:
ActiveX Control Test Container 684
Add-in Manager 681
Build Comment Web Pages 683
Connect to Database 680
Connect to Server 680
Create GUID 684
Customize 689
Customize Toolbox 680
Debug Processes 680
Error Lookup 686
External Tools 687
Macros 684
MFC/ATL Trace Tool 686
OLE/COM Object Viewer 687
Options 689
Spy++ 687
View:
Class View 43, 70, 659
Full Screen 662
Navigate Backward 664
Open 657
Open With 657

Other Windows 661
Properties Window 659
Property Pages 321, 664, 671
Resource View 50, 57, 659
Server Explorer 659
Show Tasks 662
Solution Explorer 657
Toolbars 662
Toolbox 661
Web Browser 661
Window:
Auto Hide 692
Auto Hide All 692
Dockable 692
Floating 692
Hide 692
Move to Next Tab Group 695
Move to Previous Tab Group 695
New Horizontal Tab Group 694
New Vertical Tab Group 694
New Window 691
Split 691
Windows 696
Close All Documents 695
Контекст устройства:
атрибуты 293
Координаты:
логические 293
физические 293
экранные 293

Л

Логические единицы 293

М

Макрос:
AFX_EXT_CLASS 548
AFX_MANAGE_STATE 545
ASSERT 526
BEGIN_MESSAGE_MAP 138
CATCH 329
DECLARE_DYNACREATE 363
DECLARE_DYNCREATE 608
DECLARE_MESSAGE_MAP 138
END_MESSAGE_MAP 138
IMPLEMENT_DYNACREATE 363
IMPLEMENT_DYNCREATE 610
IMPLEMENT_SERIAL 351
TCard 402
TRACE 270, 527
UNREFERENCED_PARAMETER 547
исключений 320
Массивы 344
Мастер MFC Application Wizard 13
Меню:
Build 671
Debug 673
Edit 636
File 625
Help 696
Project 664
Tools 679
View 657
Window 691
Многозадачность:
вытесняющая 360
кооперативная 360

Н

Наследование 602
Начертание 166

О

Объект 605
Объектно-ориентированное программирование 601
Окно:
Autos 520, 704
Breakpoints 515, 705
Call Stack 524
Class View 70, 659, 701
Contents 698
Disassembly 524
Dynamic Help 697
Find Setup Wizard 392
Help Topics 389
HTML Help 390
Index 698

(окончание рубрики см. на стр. 714)

Окно *(окончание)*:
Index Results 699
Locals 520, 705
Memory 523
MFC Application Wizard 404
Properties 57, 154, 659, 704
Registers 523
Registry Editor 232
Resource View 57, 659
Search 699
Search Results 700
Solution Explorer 579, 657, 701
Toolbox 58
Watch 520, 704
справка 390
Оператор:
template 335
throw 317
Операторы условной трансляции 607

П

Панель инструментов 254
Dialog Editor 68
Standard 623, 636
плавающая 623
фиксированная 622
Перегрузка:
бинарной операции 614
оператора присваивания 614
операторов 613
унарной операции 613
функций 612
Печать:
предварительный просмотр 294
Полиморфизм 602
Полоса прокрутки:
бегунок 289
страница 292
строка 292
Поток 360
интерфейсный 363
исполняющая функция 362
рабочий 361
синхронизация 367
Предотвращение утечек памяти 534
Приложение:
ActiveX Control Test Container 684
guidgen 684
Help Workshop 481
HTML Help 485
HTML Help Workshop 486
MFC/ATL Trace Tool 686
RegEdit.exe 232
Spy++ 687
Sync 369
диалоговое 13
многооконное 17
однооконное 19
Прокрутка изображения в окне 286
Процесс 360

Р

Разбиение документа на страницы 299
Режимы трансляции приложений 526

С

Семейство шрифтов 166
Система координат:
оконная 294
пользовательская 294
Системный реестр 232
Скрытие данных 603
Создание:
очереди 358
стека 357
Сообщение:
UPDATE_COMMAND_UI 261, 271
WM_CLOSE 261, 271
WM_COMMAND 388, 407
WM_COMMANDHELP 409
WM_CONTEXTMENU 389, 407
WM_DESTROY 402
WM_FONTCHANGE 171
WM_HELP 389, 407
WM_HELPHITTEST 412
WM_NOTIFY 141
WM_TCARD 402
отраженное 145
Списки 344

Справка:
всплывающее окно 390
командная 403
контекстная 403
о методике решения задачи 403
по ключевому слову 403
способы запроса 388
типы сносок 448
файл:
адресации 404
группового списка поиска 405
заголовка 404
конфигурации 405
поиска по всему тексту 405
проекта 405
таблицы содержания 404
текста 404
Справочная система:
HTML 485
гиперграфические ссылки 480
макросы 481
Стиль 166
Строка состояния 254
Структура:
AFX_EXTENSION_MODULE 548
BITMAPINFO 206
BITMAPINFOHEADER 206
CHARFORMAT2 252
HELPINFO 388, 511
HELPWININFO 402
IMAGE_DOS_HEADER 554
LOGFONT 171
NMHDR 142
OUTLINETEXTMETRIC 172
RGBQUAD 207
TEXTMETRIC 175

Т

Таблица:
имен 540
ключевых слов 401
Текст справки:
форматирование в MS Word 478
Точка останова 513

Ф

Файл:
afxcore.rtf 406, 450
afxprint.rc 415
afxprint.rtf 406, 469
afxpriv.h 473
afxres.rc 415
DateTimeDlg.cpp 130
DemoDlg.cpp 444, 499
DemoDlg.h 444, 499
DemoLib.cpp 581
Dialog.rc 65
DialogDlg.cpp 74
DialogDlg.h 72
EditAppView.cpp 242
Help.cnt 472
Help.hpj 441
Help.rc 417
help.rtf 475
HelpApp.hpj 473
HelpView.cpp 444
HTML.rc 487
Library.def 562
ListDlg.cpp 106
MainFrm.cpp 262, 275
MainFrm.h 262, 275
MDI.cpp 35, 46
MDIDoc.cpp 27
MDIDoc.h 25
MDIView.cpp 31
MDIView.h 29
Median.cpp 559
Median.h 558
MedianDlg.cpp 567
MedianDlg.h 567
MySpinCtrl.cpp 151
MyView.cpp 49
MyView.h 48
NewHelp.hm 440, 445
Page1.cpp 87
Page2.cpp 89
Page3.cpp 90
PrintAppDoc.cpp 286
PrintAppView.cpp 295
(окончание рубрики см. на стр. 716)

Файл *(окончание)*:
PrintView.cpp 288, 300
ProgressDlg.cpp 115
Resource.h 136
RichAppView.cpp 248
StdAfx.h 584
Sync.cpp 377
SyncDlg.cpp 373, 380
SyncDlg.h 373, 380
Text.cpp 227
TextDoc.cpp 226
TextView.cpp 223
ToolAppView.cpp 265, 280
заголовка 606
импорта 542
инициализации 232
определения модуля 540
реализации 606
Функция:
_CrtCheckMemory 536
_CrtDumpMemoryLeaks 536
_CrtSetDbgFlag 536
_DllMainCRTStartup 539
AddFontResource 170
AfxBeginThread 361, 377
AfxCheckError 529
AfxCheckMemory 529
AfxDumpStack 530
AfxEnableControlContainer 39
AfxEnableMemoryTracking 529
AfxEndThread 365
AfxGetApp 217
AfxGetResourceHandle 549
AfxInitExtensionModule 548
AfxIsMemoryBlock 530
AfxIsValidAddress 530
AfxIsValidString 530
AfxMessageBox 409
AfxOleInit 39
AfxSetAllocHook 530
AfxTermExtensionModule 548
AfxThrowMemoryException 328
CArchive:
IsStoring 29, 212
operator << 212
operator >> 213
CArray:
Add 348
GetSize 356
CBitmap:
CreateCompatibleBitmap 200
GetBitmapBits 201
SetBitmapBits 202
CButton:
SetCheck 97
CCmdUI:
ContinueRouting 271
Enable 222, 272
SetCheck 271
SetText 278
CControlBar:
EnableDocking 270
CDateTimeCtrl:
GetTime 134
SetFormat 135
SetTime 134
CDC:
SetBkColor 173
CDC:
BitBlt 201
CreateCompatibleDC 201
DrawText 176
EndDoc 311
EndPage 311
Escape 309
FillRect 124, 200
GetBkColor 173
GetBkMode 173
GetCharABCWidths 174
GetDeviceCaps 175, 306
GetFontData 172
GetOutlineTextMetrics 172
GetTextAlign 173
GetTextCharacterExtra 173
GetTextColor 173
GetTextExtent 283
GetTextMetrics 175
IsPrinting 307
Rectangle 292
SelectClipRgn 125
SelectObject 178, 201
SetBkMode 173
SetMapMode 307
SetMapperFlags 171
SetTextAlign 173
SetTextCharacterExtra 173
SetTextColor 173

SetTextJustification 173
SetViewPortOrg 307
StartDoc 310
StartPage 310
TabbedTextOut 176
TextOut 179
CDialog:
DoModal 244
EndDialog 385
OnOK 386
CDocManager:
DoPromptFileName 229
CDocTemplate:
GetDocString 216
CDocument:
GetFile 221
GetPathName 221
OnNewDocument 29
ReleaseFile 222
SetModifiedFlag 164, 212
UpdateAllViews 164
CEdit:
CharFromPos 244
Clear 585
Cut 585
LineLength 585
ReplaceSel 585, 598
SetSel 585
CEditView:
FindText 244
GetBufferLength 244
GetEditCtrl 243, 597
OnFindNext 244
CEvent:
ResetEvent 386
SetEvent 386
CException:
Delete 324
ReportError 598
CFile:
Close 231
GetFileName 231
GetFilePath 231
GetLength 230
Read 221
Remove 231
SeekToBegin 230
Write 222
CFrameWnd:
Create 42
DockControlBar 271
EnableDocking 270
LoadBarState 271
LoadFrame 41
SaveBarState 271
ShowControlBar 271
ChooseFont 169
CInternetFile:
ReadString 598
SetReadBufferSize 597
CInternetSession:
Close 598
OpenURL 597
CListBox:
AddString 109
DeleteString 109
GetCurSel 109
GetText 109
SetSel 109
CloseHandle 385
CMap:
GetNextAssoc 354
GetStartPosition 354
Lookup 355
RemoveAll 357
RemoveKey 357
CMemoryState:
Checkpoint 537
Difference 537
DumpAllObjectsSince 536
CMenu:
GetMenuContextHelpId 399
SetMenuContextHelpId 399
CMonthCalCtr:
GetCurSel 134
SetCurSel 134
CMultiLock:
Lock 368
Unlock 368
CObArray:
operator [] 349
CObject:
AssertValid 34, 528
(продолжение рубрики см. на стр. 718)

Функция *(продолжение)*:
Dump 34, 528
Serialize 29, 212
ConstructElement 350
CPrintInfo:
GetFromPage 312
GetMaxPage 312
GetMinPage 312
GetToPage 312
SetMaxPage 306
CProgressCtrl:
Create 120
SetPos 119
SetRange 118
CPropertyPage:
DoModal 92
OnSetActive 92
OnWizardBack 96
OnWizardFinish 97
OnWizardNext 96
SetWizardMode 92
CPropertySheet:
SetActivePage 96
SetFinishText 96
SetWizardButtons 92
CreateDC 293
CreateEvent 384
CreateFont 171
CreateFontIndirect 171
CreateScalableFontResource 170
CreateWindowEx 389
CRect:
PtInRect 201
CRgn:
CreateRectRgnIndirect 125
CRichEditView:
GetCharFormatSelection 253
OnCharEffect 250
OnParaAlign 250
SetCharFormat 253
CScrollView:
SetScrollSizes 292
CSingleLock:
IsLocked 376
Lock 368, 376
Unlock 368, 376
CSliderCtrl:
GetPos 119
SetLineSize 118
SetPageSize 118
SetRange 118
SetTicFreq 118
CStatusBar:
CommandToIndex 284
Create 282
SetIndicators 282
SetPaneInfo 283
SetPaneText 284
CString:
Format 278
GetBuffer 597
LoadString 320
ReleaseBuffer 597
CToolBar:
CreateEx 270
LoadToolbar 270
CTypedPtrArray:
GetAt 354
operator [] 354
CTypedPtrList:
AddHead 358
AddTail 358
GetAt 349
GetHead 349
GetHeadPosition 354, 356
GetNext 349, 356
GetPrev 349
GetTail 349
RemoveAll 356
RemoveAt 356
RemoveHead 349, 358
RemoveTail 349
CTypedPtrMap:
GetNextAssoc 350
Lookup 350
CView:
DoPreparePrinting 311
GetDocument 34, 292
OnBeginPrinting 310
OnDraw 34, 164
OnEndPrinting 311
OnFilePrint 33
OnFilePrintPreview 33
OnPrepareDC 164, 310
OnPreparePrinting 310

OnPrint 311
OnUpdate 165
CWinApp:
AddDocTemplate 41
EnableHtmlHelp 486
GetProfileInt 235
GetProfileString 217, 235
InitInstance 39, 227
LoadStdProfileSettings 39, 235
OnContextHelp 412
OnHelp 409
OnHelpIndex 409
OnIdle 412
ParseCommandLine 42
PreTranslateMessage 412
ProcessShellCommand 42
SetRegistryKey 39, 234
WinHelp 396
WriteProfileInt 235
WriteProfileString 217
CWinThread:
CreateThread 364
ExitInstance 363
InitInstance 364
OnIdle 364
PostThreadMessage 137
PreTranslateMessage 364
ProcessWndProcException 364
Run 153, 364
CWnd:
BeginPaint 164, 294
DefWindowProc 388
DoDataExchange 81
EndPaint 164
GetClientRect 199, 292
GetDC 164, 294
GetParentOwner 284
GetSafeHwnd 385, 510
GetStyle 271
GetWindowContextHelpId 399
GetWindowDC 294
Invalidate 222, 292
IsDialogMessage 412
ModifyStyle 134
OnContextMenu 446, 510
OnHelpInfo 446, 511
OnHScroll 119
OnNotify 143
OnPaint 124, 165
OnWndMessage 153
PreCreateWindow 34
ReleaseDC 164
ScreenToClient 124, 294
SetDlgItemText 81, 109, 134
SetWindowContextHelpId 399
ShowWindow 43
UpdateData 81
UpdateWindow 43
DestructElement 350
DispatchMessage 137
DllMain 539, 547
DuplicateHandle 366
EnumFontFamilies 169
FindResource 549
FreeLibrary 543
GetCharABCWidthsFloat 174
GetCharWidth32 174
GetCharWidthFloat 174
GetCurrentDirectory 217
GetDlgItem 124
GetExitCodeThread 365
GetMessage 137
GetProcAddress 543
GetTabbedTextExtent 175
GetTextExtentExPoint 175
GetTextExtentPoint32 173
GetWindowRect 124
HtmlHelp 509
LoadLibrary 543
LoadResource 549
MessageBox 221, 320
PeekMessage 137
PostMessage 367, 385
PostQuitMessage 365
RemoveFontResource 171
ResumeThread 366
SerializeElement 351
SetCurrentDirectory 217
SetDIBitsToDevice 207
SetEvent 386
SetFileAttributes 231
SetThreadPriority 362, 364
SetWindowLong 270
Sleep 377
TranslateMessage 137
(окончание рубрики см. на стр. 720)

Функция *(окончание)*:
UpdateAllViews 367
WaitForMultipleObjects 384
WaitForSingleObject 385
WinHelp 395
WinMain 360
виртуальная 604, 614

Ц

Цикл обработки сообщений 361

Ш

Шаблон:
класса 341
функции 338
Шрифт 166

Э

Элемент управления:
Button 103
Check Box 63
Combo Box 62
Date Time Picker 126
Edit Control 59
Group Box 64
List Box 102
Month Calendar Control 126
Picture Control 111
Progress 110
Radio Button 64
Slider Control 110
Spin Control 61
Static Text 58
изменение положения 59