А. Мешков, Ю. Тихомиров

bhv
www.bhv.ru
www.bhv.kiev.ua

Visual C++ и MFC

2 издание

МАСТЕР
РУКОВОДСТВО ДЛЯ ПРОФЕССИОНАЛОВ

CITYLINE
Скидка 10%
на подключение
к Internet

Дискета содержит файлы всех примеров

Андрей Мешков
Юрий Тихомиров

Visual C++ и MFC

Издание второе

Дюссельдорф Киев
Москва Санкт-Петербург

УДК 681.3.06

Книга посвящена объектно-ориентированному программированию для 32-разрядных систем Windows 95/98 и Windows NT с использованием компилятора Visual C++ 6 и библиотеки классов MFC версии 4.23.
Книга содержит исчерпывающую информацию о библиотеке классов MFC (Microsoft Foundation Classes) и методах создания приложений Windows на основе классов этой библиотеки, а также большое количество наглядных и подробно прокомментированных примеров приложений, полные тексты которых приведены на сопроводительной дискете.

Для программистов, владеющих языками C и C++ и имеющих опыт программирования в системе Windows

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зав. редакцией	*Наталья Таркова*
Ответственный редактор	*Татьяна Темкина*
Компьютерная верстка	*Наталья Смирнова*
Корректура	*Светлана Журавина*
Дизайн обложки	*Наталья Смирнова*
Производство	*Николай Тверских*

Мешков А. В., Тихомиров Ю. В.
Visual C++ и MFC. — 2-е изд. — СПб.: БХВ — Санкт-Петербург, 1999. — 1040 с.: ил.

ISBN 5-8206-0073-8

© Мешков А. В., Тихомиров Ю. В.,1999
© Оформление, издательство "БХВ — Санкт-Петербург", 1999

Лицензия ЛР № 065953 от 15.06.98. Подписано в печать 26.10.99.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 83,85.
Тираж 3000 экз. Заказ
"БХВ — Санкт-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 ордена Трудового Красного Знамени ГП "Техническая книга"
Комитета Российской Федерации по печати.
198005, Санкт-Петербург, Измайловский пр., 29.

Содержание

ВВЕДЕНИЕ **1**

ЧАСТЬ I. ОБЗОР VISUAL C++ 6 ..... **9**

ГЛАВА 1. ЗНАКОМЬТЕСЬ — VISUAL C++ 6 ..... **9**

Что нового в этой версии? 9
Настройки проекта 13
Настройка опций компилятора 14
Настройка опций компоновщика 16

ГЛАВА 2. ИНТЕГРИРОВАННАЯ СРЕДА РАЗРАБОТКИ (IDE) **19**

ЧАСТЬ II. ПРОГРАММИРОВАНИЕ ПОД MS WINDOWS **31**

ГЛАВА 3. ЗНАКОМСТВО С WINDOWS **33**

Пользовательский интерфейс 33
Окна, элементы управления, модальные и немодальные блоки диалога 34
Ресурсы приложения 39
Система, управляемая сообщениями 40
Интерфейс прикладного программирования 42
Управление графическим выводом 44
Графические объекты 44
Аппаратно-независимый графический вывод 45
Контекст устройства 45
Графические режимы 48
Работа со шрифтами 49
APIShell — пример приложения 50

ГЛАВА 4. СТРУКТУРА ПРИЛОЖЕНИЙ ДЛЯ WINDOWS **59**

Стандартный цикл обработки сообщений 61
Оконная процедура 62
Основные типы оконных сообщений Windows 63
APIWin — пример приложения 64

ЧАСТЬ III. БИБЛИОТЕКА КЛАССОВ MICROSOFT FOUNDATION CLASSES (MFC) 75

ГЛАВА 5. ОБЗОР БИБЛИОТЕКИ MFC 77

Иерархия классов MFC 78
CObject — основа всех классов 78
Классы, определяющие архитектуру приложения 84
Классы приложения и поддержки потоков 85
Классы маршрутизации команд 86
Классы документов 87
Классы шаблонов документов 87
Окна, блоки диалога и элементы управления 88
Классы фреймов окон 88
Классы представлений 89
Классы блоков диалога 90
Классы элементов управления 92
Классы панелей элементов управления 94
Классы для вывода на экран и печать 95
Классы контекста устройства 95
Классы графических объектов 95
Классы простых типов данных 96
Массивы, списки и ассоциативные списки 97
Классы шаблонов массивов, списков и ассоциативных списков 97
Готовые к использованию классы массивов 98
Готовые к использованию классы списков 98
Готовые к использованию классы ассоциативных списков 98
Классы для работы с файлами и базами данных 99
Классы файлов ввода/вывода 99
Классы для работы с DAO 100
Классы для работы с ODBC 100
Классы для работы в Internet и сетях 101
Классы ISAPI 101
Классы сокетов Windows 102
Классы для работы с OLE 102
Контейнерные классы OLE 103
Серверные классы OLE 103
Классы OLE для передачи данных и drag-and-drop 104
Классы автоматизации OLE 104
Классы, связанные с OLE 104
Классы для отладки и обработки исключений 105
Классы поддержки отладки 106
Классы обработки исключений 106

ГЛАВА 6. МАКРОСЫ, ГЛОБАЛЬНЫЕ ФУНКЦИИ И ПЕРЕМЕННЫЕ 107

Типы данных 107
Получение информации о приложении 108
Модель объекта времени выполнения 109
Диагностика объектов 111
Основные макросы 111
Форматирование строк и окно сообщения 112

ЧАСТЬ IV. ПРОГРАММИРОВАНИЕ С ИСПОЛЬЗОВАНИЕМ БИБЛИОТЕКИ MFC 115

ГЛАВА 7. ОСНОВНЫЕ СОСТАВЛЯЮЩИЕ ПРИЛОЖЕНИЯ НА БАЗЕ БИБЛИОТЕКИ КЛАССОВ MFC 117

Соглашения об именах MFC 121
Включаемые файлы 122
Функция WinMain 122
Класс CWinApp 125
Минимальная программа для Windows 131
Регистрация класса окна 134
Создание окна 139

ГЛАВА 8. ОБРАБОТКА СООБЩЕНИЙ В БИБЛИОТЕКЕ КЛАССОВ MFC 141

Цикл обработки сообщений MFC 141
Категории сообщений 143
Карта сообщений 144
Компоненты карты сообщений 146
Класс CCmdTarget 150
Стандартный маршрут команды 151
Команды обновления и класс CCmdUI 154
Функции для работы с сообщениями 155

ЧАСТЬ V. СОЗДАНИЕ SDI- И MDI-ПРИЛОЖЕНИЙ НА БАЗЕ БИБЛИОТЕКИ MFC 159

ГЛАВА 9. КЛАССЫ ОКОН БИБЛИОТЕКИ MFC 161

Окна, определенные в ОС Windows 161
Класс CWnd 166
Класс CFrameWnd 169
Создание окна и библиотека MFC 171
Создание главного окна SDI-приложения 185
Создание дочерних окон 191

Ограничение размеров окна ... 198
Функция GetSystemMetrics ... 199
Ограничение доступа к окну ... 200

ГЛАВА 10. СОЗДАНИЕ МНОГОДОКУМЕНТНЫХ ПРИЛОЖЕНИЙ ... 202

Класс CMDIFrameWnd ... 203
Класс CMDIChildWnd ... 206
MDI — пример приложения ... 207
Обработка команд меню ... 226
Unicode ... 227
Обработка команд меню (продолжение) ... 229
Обработка сообщения WM_PAINT ... 232

ЧАСТЬ VI. ЭЛЕМЕНТЫ ИНТЕРФЕЙСА ПОЛЬЗОВАТЕЛЯ ... 237

ГЛАВА 11. ПАНЕЛИ ЭЛЕМЕНТОВ УПРАВЛЕНИЯ ... 239

Класс CControlBar ... 239
Класс CToolBar ... 243
Класс CToolBarCtrl ... 249
Класс CStatusBar ... 257
Класс CStatusBarCtrl ... 261
Класс CDialogBar ... 265
Класс CReBar ... 266
Класс CReBarCtrl ... 268
ControlBars — пример приложения ... 275

ГЛАВА 12. МЕНЮ ... 293

Основные типы меню ... 293
Класс CMenu ... 296
Создание меню на основе шаблона ... 306
Динамическое создание меню ... 311
Динамическое изменение меню ... 313
Системное меню ... 320
Контекстное меню ... 323
Создание собственных маркеров состояния ... 326
Самоотображение элементов меню ... 328

ГЛАВА 13. АКСЕЛЕРАТОРЫ, ГРАФИЧЕСКИЕ И ДРУГИЕ РЕСУРСЫ ... 330

Акселераторы ... 330
Графические ресурсы ... 331
Строковые ресурсы ... 333
Ресурсы, определяемые программистом ... 334

ЧАСТЬ VII. БЛОКИ ДИАЛОГА 335

ГЛАВА 14. МОДАЛЬНЫЕ И НЕМОДАЛЬНЫЕ БЛОКИ ДИАЛОГА..... 337

Класс CDialog 340
Template — пример приложения 352
Блок диалога в качестве главного окна приложения 367
Обмен данными с блоком диалога 368
Создание блока диалога на основе шаблона в памяти 372

ГЛАВА 15. СТАНДАРТНЫЕ БЛОКИ ДИАЛОГА 377

Класс CFileDialog 377
Класс CColorDialog 390
Класс CFontDialog 395
Класс CFindReplaceDialog 401
Класс CPrintDialog 405
Класс CPageSetupDialog 414

ЧАСТЬ VIII. ГРАФИЧЕСКИЙ ИНТЕРФЕЙС УСТРОЙСТВ (GDI) 421

ГЛАВА 16. КЛАССЫ ГРАФИЧЕСКОГО ИНТЕРФЕЙСА 423

Контексты устройств 424
Графические объекты 426

ГЛАВА 17. КЛАССЫ КОНТЕКСТОВ УСТРОЙСТВ..... 430

Класс CDC 430
Создание объекта класса и его связь с контекстом устройства 430
Настройка процесса рисования 434
Режимы отображения 440
Функции рисования 446
Отображение битовых массивов 452
Управление выводом текста 456
Дополнительные функции 462

ГЛАВА 18. КЛАССЫ ГРАФИЧЕСКИХ ОБЪЕКТОВ 463

Класс CGdiObject 463
Карандаш (класс CPen) 468
Кисть (класс CBrush) 474
Шрифт (класс CFont) 475
Битовый массив (класс CBitmap) 482
Регион (класс CRgn) 489
Создание регионов 489
Комбинирование регионов 491

PQCLOCK — ПРИМЕР ПРИЛОЖЕНИЯ 494
Класс приложения 494
Класс окна 497

ЧАСТЬ IX. АРХИТЕКТУРА "ДОКУМЕНТ/ПРЕДСТАВЛЕНИЕ" (DOCUMENT/VIEW) 511

ГЛАВА 19. ОСНОВНЫЕ ПОЛОЖЕНИЯ 513

МЕСТО ОБЪЕКТА-ПРИЛОЖЕНИЯ В АРХИТЕКТУРЕ "ДОКУМЕНТ/ПРЕДСТАВЛЕНИЕ" 516
Класс CDocTemplate 516
Класс CSingleDocTemplate 521
Класс CMultiDocTemplate 522
РОЛЬ ФРЕЙМОВ В АРХИТЕКТУРЕ "ДОКУМЕНТ/ПРЕДСТАВЛЕНИЕ" 530
СЕРИАЛИЗАЦИЯ 535
Класс CArchive 541

ГЛАВА 20. ДОКУМЕНТ И ЕГО ПРЕДСТАВЛЕНИЯ 544

ДОКУМЕНТЫ 545
Класс CDocument 547
ПРЕДСТАВЛЕНИЯ 555
Класс CView 558
Класс CCtrlView 562
Класс CEditView 562
Классы CListView и CTreeView 568
Класс CScrollView 568
Класс CSplitterWnd 573

ГЛАВА 21. ПЕЧАТЬ И ПРЕДВАРИТЕЛЬНЫЙ ПРОСМОТР ДОКУМЕНТОВ 583

ВЫБОР И НАСТРОЙКА ПАРАМЕТРОВ ПРИНТЕРА 585
СОЗДАНИЕ КОНТЕКСТА УСТРОЙСТВА 588
ПЕЧАТЬ ДОКУМЕНТОВ И БИБЛИОТЕКА MFC 588
ПРЕДВАРИТЕЛЬНЫЙ ПРОСМОТР ДОКУМЕНТА 598

ЧАСТЬ X. ЭЛЕМЕНТЫ УПРАВЛЕНИЯ 607

ГЛАВА 22. ОБЗОР СТАНДАРТНЫХ ЭЛЕМЕНТОВ УПРАВЛЕНИЯ 609

СОЗДАНИЕ ЭЛЕМЕНТОВ УПРАВЛЕНИЯ 610
Создание элементов управления в редакторе ресурсов 610
Создание элементов управления в тексте приложения 610

ГЛАВА 23. КЛАССЫ СТАНДАРТНЫХ ЭЛЕМЕНТОВ УПРАВЛЕНИЯ ... 612

СТАТИЧЕСКИЕ ЭЛЕМЕНТЫ УПРАВЛЕНИЯ 612

Создание ... 612
Отображение битового массива ... 615
Отображение значка ... 616
Отображение курсора ... 616
Отображение расширенных метафайлов ... 617
Изменение цвета ... 617
Кнопки ... 619
Создание ... 622
Определение и изменение текущего состояния кнопок ... 622
Определение и изменение стиля кнопки ... 624
Назначение изображений ... 624
Переопределяемые функции ... 625
Простейший текстовый редактор ... 625
Инициализация ... 626
Выделение текста ... 628
Работа со строками ... 629
Работа с блоками текста ... 631
Параметры и флаги ... 632
Дополнительные функции ... 633
Список ... 634
Инициализация ... 635
Изменение параметров списка ... 636
Функции для работы с содержимым списка ... 638
Операции над элементами списка ... 641
Переопределяемые функции ... 643
Пример самоотображаемого списка ... 646
Комбинированный список ... 649
Инициализация ... 650
Основные операции с комбинированным списком ... 652
Операции со строками и виртуальные функции ... 654
Полоса прокрутки ... 655
Создание ... 657
Работа с полосой прокрутки ... 658

ГЛАВА 24. КЛАССЫ ЭЛЕМЕНТОВ УПРАВЛЕНИЯ С РАСШИРЕННЫМИ ВОЗМОЖНОСТЯМИ ... 663

Кнопки с нанесенным битовым изображением ... 663
Кнопки в блоке диалога ... 664
Кнопки, созданные программным путем ... 665
Функции класса ... 665
Список, имеющий флажки ... 667
Инициализация ... 667
Функции для работы с расширенным списком ... 667
Виртуальные функции ... 668

Комбинированный список с поддержкой списка образов 674
Создание комбинированного списка 674
Основные операции 675
Функции работы с атрибутами 676
Список с возможностью перетаскивания элементов 678
Ограничения на использование 678
Создание списка 678
Функции, предназначенные для перемещения элементов 679
Методика обхода ограничений 680
Реализация копирования элементов списка 680

ГЛАВА 25. ВСПОМОГАТЕЛЬНЫЕ ЭЛЕМЕНТЫ УПРАВЛЕНИЯ 683

Просмотр видеоклипов 683
Элемент управления "анимация" 684
Класс CAnimateCtrl 684
Уведомления 686
"Горячая клавиша" 688
Глобальные "горячие клавиши" 691
Поточно-ориентированные "горячие клавиши" 691
Пример использования класса CHotKeyCtrl 693
Индикатор 698
Использование индикатора в панели диалога 700
Линейка с ползунком 703
Уведомления 703
Создание элемента управления 704
Параметры элемента управления 706
Положение меток 706
Положение ползунка 707
Счетчик 707
Уведомления 708
Автоматическое изменение 710
Создание счетчика 710
Параметры элемента управления 711
Всплывающая подсказка 713
Оконная поддержка 714
Возвращаемся к классу CToolTipCtrl 716

ГЛАВА 26. КОМПЛЕКСНЫЕ ЭЛЕМЕНТЫ УПРАВЛЕНИЯ 720

Список изображений 720
Создание списка 721
Изменение содержимого списка 722
Рисование изображений 723
Перемещение изображений 724
Информация о списке 727

Заголовок ... 727
Создание элемента заголовка ... 728
Атрибуты элементов заголовка ... 732
Обработка извещений элемента заголовка ... 733
Элемент заголовка, определяемый пользователем ... 735
Просмотр списка ... 736
Режимы вывода ... 736
Создание просмотра списка ... 737
Работа со столбцами ... 741
Параметры просмотра списка ... 743
Функции для работы со списком в целом ... 744
Рабочие области просмотра списка ... 746
Списки изображений ... 746
Виртуальные списки ... 748
Основные и дополнительные поля ... 748
Записи по запросу ... 748
Поиск и сортировка записей ... 751
Редактирование надписей записей ... 752
Обработка уведомлений ... 753
Просмотр списка с возможностью перемещения записей ... 754
Переопределяемые функции ... 755
ListCtrl — пример приложения ... 755
Просмотр дерева ... 763
Создание просмотра дерева ... 763
Списки изображений ... 764
Изменение содержимого дерева ... 765
Поиск и сортировка записей дерева ... 770
Обработка уведомлений ... 772
Класс CMonthCalCtrl ... 774
Обработка уведомлений ... 777
Класс CDateTimeCtrl ... 777
Обработка уведомлений ... 780

ГЛАВА 27. РАСШИРЕННЫЙ ТЕКСТОВЫЙ РЕДАКТОР ... 781

Класс CRichEditCtrl ... 781
Создание объекта класса CRichEditCtrl ... 782
Изменение параметров редактора ... 783
Настройка окна редактора ... 785
Размер содержимого редактора ... 786
Работа со строками ... 787
Выделение содержимого редактора ... 789
Работа с буфером обмена ... 791
Печать содержимого редактора ... 792
Уведомляющие сообщения ... 793

Расширенные возможности ... 795
Расширенный редактор в рамках архитектуры "документ/представление" ... 802
Класс CRichEditView ... 804
Класс CRichEditDoc ... 810

ГЛАВА 28. НАБОРЫ СВОЙСТВ ... 811

Создание набора свойств ... 812
Создание страницы свойств ... 813
Создание объекта "набор свойств" ... 814
Настройка окна набора свойств ... 815
Добавление страниц ... 816
Создание модального окна свойств ... 818
Создание немодального окна набора свойств ... 818
Обмен данными ... 819
Операция Apply ... 820
Класс CPropertySheet ... 821
Операции над набором свойств ... 825
Вспомогательные функции ... 827
Изменение параметров отображения ... 828
Класс CPropertySheetEx ... 830
Класс CPropertyPage ... 831
Создание страницы свойств ... 834
Переопределяемые функции ... 835
Класс CPropertyPageEx ... 836
Вкладки ... 838
Создание вкладок ... 838
Вспомогательные функции ... 841
Изменение вкладок ... 842
Мастера ... 845
Создание мастеров ... 846
Переопределяемые функции ... 846

ЧАСТЬ XI. ПОДДЕРЖКА МНОГОЗАДАЧНОСТИ ... 849

ГЛАВА 29. ВЫТЕСНЯЮЩАЯ МНОГОЗАДАЧНОСТЬ ... 851

Общие положения ... 851
Классы процессов и приоритеты потоков ... 853
Класс Idle ... 854
Класс Normal ... 854
Класс High ... 855
Класс Realtime ... 855
Относительные приоритеты потоков ... 855

Взаимодействие программ 858
Синхронизация потоков 859
Критические секции 859
Ожидающие функции 860
Объекты синхронизации 861
Библиотека MFC и многозадачность 864
Класс и функции для поддержки многопотоковых приложений 865
Классы и функции для поддержки взаимодействия потоков и процессов 866

ГЛАВА 30. СОЗДАНИЕ ПОТОКОВ 868

Пример создания наследуемых потоков 871
Создание интерфейсных потоков 874
Класс CWinThread 875
Создание интерфейсных потоков (продолжение) 880
CUIThreadApp — класс всего приложения 881
CUIThreadFrame — класс основного окна 881
CLineThread — класс дополнительного потока 886
CLineWnd — класс окна "бегущая строка" 888
CEditWnd — класс окна редактора 893
Создание рабочих потоков 895
Управляющая функция и ее параметр 895
Пример управляющей функции 896
Завершение потоков 897
Нормальное завершение 897
Экстренное завершение 899
Код завершения 900
Права доступа к потоку 901

ГЛАВА 31. ВЗАИМОДЕЙСТВИЕ ПОТОКОВ 903

Взаимоисключение доступа 903
Организация доступа к объектам классов MFC 905
Списки дескрипторов объектов Windows 906
Graphics — пример приложения 907
Использование классов объектов синхронизации 915
Создание класса разделяемых объектов 916
Использование классов объектов синхронизации 921
Классы объектов синхронизации 922
Класс CSyncObject — базовый класс объектов синхронизации 922
Критическая секция (класс CCriticalSection) 925
Взаимоисключение (класс CMutex) 926
Событие (класс CEvent) 927
Семафор (класс CSemaphore) 929
Классы, обеспечивающие синхронизацию 930

TERMINAL — МНОГОПОТОКОВОЕ ПРИЛОЖЕНИЕ, ВЗАИМОДЕЙСТВУЮЩЕЕ С ПОСЛЕДОВАТЕЛЬНЫМ ПОРТОМ 934
Класс блока диалога и функции потоков 934
Окно редактора 942
Работа с последовательным портом 943
Кольцевой буфер общего доступа 944

ЧАСТЬ XII. INTERNET И БАЗЫ ДАННЫХ 947

ГЛАВА 32. РАБОТА С ИНФОРМАЦИОННЫМ СЕРВЕРОМ INTERNET 949

ОБЗОР ISAPI (API СЕРВЕРА INTERNET) 950
КЛАССЫ ISAPI 953
Класс CHttpServer 954
Класс CHttpServerContext 955
Карта разбора 956
СОЗДАНИЕ РАСШИРЕНИЯ ISAPI 957

ГЛАВА 33. КЛАССЫ БИБЛИОТЕКИ MFC ДЛЯ РАБОТЫ С БАЗАМИ ДАННЫХ 961

КЛАССЫ ДЛЯ РАБОТЫ С ODBC 961
Класс CDatabase 961
Класс CRecordset 965

ГЛАВА 34. ИСПОЛЬЗОВАНИЕ РАСШИРЕНИЯ ISAPI 982

ОПИСАНИЕ СОПРОВОДИТЕЛЬНОЙ ДИСКЕТЫ 993

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ 996

Введение

И снова программирование для Windows. Какой язык программирования вы бы ни выбрали, очевидно, что базой для этого является Windows API (Application Programming Interface — интерфейс прикладного программирования), освоив его, можно писать программы любой степени сложности, используя все возможности, предоставляемые операционной системой. Но сложность современных приложений так возросла, что разработка коммерческого программного обеспечения с использованием только Windows API и языка C уже не может удовлетворить программиста. Естественно, что технология программирования "не могла остаться в стороне", и в настоящее время предлагается два подхода к построению сложного программного обеспечения. С одной стороны, это использование систем визуального программирования. А с другой — применение библиотек классов, которые самостоятельно выполняют массу черновой работы и при этом гораздо в большей степени, нежели система Windows сама по себе, "ведут" программиста, структурируя и облегчая процесс создания программного обеспечения. Не будем сравнивать — каждый вправе сам решить, что ему больше подходит. Скажем только, что основная проблема, с которой сталкивается программист, начиная работать в некоторой операционной системе, состоит в понимании фундаментальных принципов и моделей, включенных в ее архитектуру. И только проникнувшись "образом мышления Windows", начинаешь понимать, что программирование под Windows не сложнее любого другого. А какой из подходов больше "прячет" эти принципы и модели, пусть каждый решает сам.

Мы не ставили своей целью написать традиционный учебник по программированию в Windows — об этом написано (и пишется) достаточно много хороших (и не очень) книг. Основным объектом нашего внимания является библиотека классов MFC. И именно *детальному* знакомству с ней посвящена основная часть книги. Но, с другой стороны, классы библиотеки настолько тесно связаны с системой Windows, а многие непосредственно инкапсулируют ее объекты, что мы не могли не описать саму систему Windows, предоставляемые ею возможности и те механизмы, без понимания которых попросту невозможно создать Windows-приложение. Тем не менее, мы отдаем себе отчет в том, что эти разделы не могут заменить литературу, посвященную программированию в Windows с использованием функций API.

У библиотеки классов MFC — основного предмета данной книги — две основные цели. Во-первых, скрыть от программиста то, что является скорее вспомогательным, чем обязательным аспектом при написании Windows-программ — всю "черновую работу". В частности, при использовании MFC

не нужно переносить из приложения в приложение практически не изменяющуюся функцию WinMain, а можно непосредственно приступать к решению своей конкретной задачи.

Во-вторых, обеспечить разработчиков средством, которое, в отличие от Win32 API, более структурированно и понятно. Материал книги как раз и посвящен представлению классов и методов их использования, которое позволяет существенно упростить создание как простейших приложений утилитарного характера, так и сложных систем.

Несмотря на то что сама библиотека классов сложна и многогранна, и на сегодняшний день существует более десяти ее версий, для начала работы с ней вполне достаточно владеть языком программирования C++. Новичкам даже не нужно знать принципы программирования в Windows. Дело в том, что в компилятор Visual C++ включено специальное средство, позволяющее автоматизировать процесс создания приложения. Это средство — MFC AppWizard — создаст для вас работающее приложение. Конечно, это будет только шаблон, и наполнять его конкретным содержанием придется самостоятельно. Но первоначальное знакомство со сгенерированным кодом принесет новичкам значительную пользу.

Должны сразу оговориться, что в одной книге невозможно осветить все аспекты программирования в Windows, как с использованием, так и без использования библиотеки классов MFC — это слишком обширная тема и очень трудно определить, где следует остановиться. Тем не менее, в предлагаемую книгу мы постарались включить достаточное количество информации, чтобы не только помочь начать писать собственные приложения для Windows, но и предоставить опытным программистам разнообразный материал не только по стандартным и уже не раз описанным в литературе возможностям (хотя и со своей точки зрения), но и по тем "изюминкам", которые до сих пор нигде не описаны и были найдены авторами как в процессе работы, так и при написании книги.

В этом издании мы рассматриваем библиотеку MFC версии 4.23, предназначенную для 32-разрядных операционных систем Windows 95/98 и Windows NT и включенную в шестую версию Visual C++.

Материала по предлагаемой теме так много, что от рассмотрения многих вопросов пришлось отказаться. Настоящее издание, в отличие от предыдущего, выходит в одном томе, поэтому мы убрали некоторые главы, в основном, по программированию с использованием Win32 API. Тем не менее, мы добавили часть, посвященную современным вопросам программирования для Internet и баз данных. Следует учитывать также, что изложение материала, в силу специфики Windows, не может быть последовательным. Долгие поиски привели к оптимальной, на наш взгляд, структуре изложения. За основу мы взяли три типа стандартных приложений Windows: приложения, поддерживающие SDI (Single Document Interface — однодокументный ин-

терфейс) и MDI (Multiple Document Interface — многодокументный интерфейс), а также приложение на базе блока диалога.

- *Часть I (главы 1 и 2)* посвящена краткому знакомству с Visual C++ 6 и интегрированной средой разработки — IDE, общим принципам их организации и тому новому, что появилось в шестой версии.
- В *Части II (главы 3 и 4)* приведен краткий обзор ОС Windows, а также рассмотрена общая структура приложений для этих операционных систем.
- *Часть III (главы 5 и 6)* посвящена обзору классов библиотеки MFC. Подробно описан базовый класс библиотеки — CObject и кратко описаны все компоненты библиотеки. В главе 6 представлены глобальные библиотечные функции и макросы, без которых не обходится ни одно приложение, написанное с использованием даже базовых возможностей библиотеки.
- *Часть IV (главы 7 и 8)* содержит описание требований к структуре и составляющим каждого приложения, созданного с использованием библиотеки классов MFC, а также рассматривается класс приложения CWinApp (глава 7). Глава 8 посвящена подробному изложению одного из основных механизмов Windows — работы с сообщениями, естественно с точки зрения библиотеки MFC.

Эти четыре части представляют собой некоторое теоретическое введение в Windows и библиотеку MFC, но при этом не забыты и практические аспекты программирования.

- *Часть V (главы 9 и 10)* посвящена основным объектам Windows — окнам и созданию на их основе однодокументных и многодокументных приложений. В главе 9 подробно рассмотрен базовый класс для всех типов и видов окон — класс CWnd, а также механизм создания однодокументных приложений. Глава 10 целиком посвящена описанию процесса создания многодокументных приложений.
- Элементы интерфейса пользователя подробно рассмотрены в *Части VI (главы 11—13)*. Глава 11 знакомит читателя с мощными средствами создания и использования панелей элементов управления, таких как панель инструментов и строка состояния. Меню, графические и другие ресурсы подробно рассматриваются в главах 12 и 13.
- *Часть VII (главы 14 и 15)* полностью охватывает третий тип стандартных приложений Windows — приложения на базе блока диалога. В главе 14 рассматриваются основные принципы создания и работы с блоками диалога и класс CDialog, предназначенный для решения этих задач. Системные (или стандартные) блоки диалога описаны в главе 15. Они позволяют настраивать практически все параметры (открытие и сохранение файла, выбор шрифта и цвета, поиск и т. д.), которые могут понадобиться создателям приложений.

- *Часть VIII (главы 16—18)* содержит практические аспекты процесса графического вывода. Знакомство с графическими возможностями Windows и библиотеки классов MFC в дальнейшем упростит понимание того, например, как следует создавать самоотображаемые (owner-draw) элементы управления. Описываемые классы практически полностью инкапсулируют соответствующие объекты Windows, не добавляя ничего нового. Однако функции этих классов группируют функции Win32 API, которые так или иначе вызываются в реализации классов. Одного этого было бы достаточно, чтобы создать и использовать классы взамен функций Win32 API. Сама реализация класса уже сводит воедино все или практически все необходимые возможности. Описывая функции классов, мы раскрыли все, что поддерживается в Windows. Таким образом, эту часть можно рассматривать как всеобъемлющее руководство по использованию возможностей графического вывода. Здесь также представлен большой набор примеров функций и приложений, который позволит даже новичку в программировании для Windows разобраться в том, как следует осуществлять графический вывод.
- В *Части IX (главы 19—21)* сосредоточено все, что необходимо знать о поддержке со стороны библиотеки MFC архитектуры "документ/представление" (Document/View architecture). Данная архитектура напрямую не поддерживается Windows, поэтому только описания классов и функций явно недостаточно. В части представлена взаимосвязь различных классов, последовательность вызовов тех или иных функций. Эти знания позволят вам в дальнейшем легко, с минимальными накладными расходами переопределять только те функции, новое содержание которых наполнит нужным вам смыслом разрабатываемые приложения.
- *Часть X (главы 22—27)* посвящена классам, инкапсулирующим элементы управления Windows. В главах 22—24 сосредоточена информация о стандартных элементах управления (controls) и большая часть теории, касающейся всех элементов управления. Главы 25 и 26 посвящены новым общим элементам управления (common controls). В главе 27 описывается элемент управления Windows — текстовый редактор формата RTF, который позволяет легко и быстро создать приложение наподобие WordPad, входящего в комплект поставки Windows. В этой части не только подробно излагаются и описываются соответствующие классы, но также содержится информация по самим элементам управления и описывается ряд тонких моментов, которые вряд ли можно найти в документации.
- *Часть XI (глава 28)* завершает знакомство с классами, инкапсулирующими элементы управления Windows. Здесь собрана информация, которая позволит создавать наборы свойств, состоящие из нескольких блоков диалога с вкладками, и мастера, также представляющие собой наборы блоков диалога. Различие между ними заключается в том, что в наборе свойств пользователь переходит от одного блока диалога к другому, вы-

бирая соответствующую вкладку, доступную в произвольный момент времени, а в мастере переход от одного к другому может осуществляться только последовательно. Но несмотря на различия, эти элементы управления по способу работы с ними очень схожи и базируются на одних и тех же классах — CPropertySheet(Ex), CPropertyPage(Ex) и CTabCtrl.

- И Windows 95/98, и Windows NT — многозадачные системы и, естественно, мы не могли обойти вниманием этот важный вопрос и его поддержку в библиотеке классов MFC. Поэтому в *Части XII (главы 29—31)* подробно рассматриваются вопросы многозадачности и создания потоков.
- Одним из мощных средств современных языков программирования и операционных систем является встроенная обработка исключений. В *Части XIII (главы 32—36)* подробно рассматриваются обработка исключений в языке C++ и структурная обработка исключений, а также поддержка механизма обработки исключений в библиотеке классов MFC.
- Хотя часть материала предыдущего издания пришлось сократить, мы решили дополнить книгу информацией о базах данных и Internet. Каждая тема сама по себе чрезвычайно велика, интересна и не может быть рассмотрена в рамках одной книги. Поэтому материал, представленный в *Части XIV (главы 37—39)*, "идет от конкретного примера", раскрывая один из аспектов программирования для Internet — работа с базами данных и предоставление результатов пользователю.
- Примеры приложений иллюстрируют то, что можно найти на страницах книги. Сопроводительная дискета содержит полные исходные тексты и исполняемые файлы всех рассматриваемых примеров. Мы надеемся, что они будут полезны читателям.

В заключение хотелось бы сказать несколько добрых слов в адрес нашего редактора Татьяны Темкиной. То, что книга не только содержит полезный материал, но и, на наш взгляд, достаточно легко читается, во многом именно ее заслуга. Мы искренне признательны ей за это.

Все замечания и пожелания можно присылать по адресу:

- Издательство "БХВ — Санкт-Петербург", 199397, С.-Петербург, а/я 194
- E-mail: bhv@mail.nevalink.ru

ОБЗОР VISUAL C++ 6

Глава 1

Знакомьтесь — Visual C++ 6

Новый 32-разрядный компилятор Microsoft Visual C++ 6.0 для операционных систем Windows (Windows 95/98 и Windows NT) — одна из составных частей мощного пакета компиляторов Developer Studio, в который входят новейшие версии таких известных программных продуктов, как Visual Basic, Visual J++, Visual FoxPro, Visual InterDev, Visual SourceSafe и других. Прежде чем переходить к деталям, окинем взглядом пакет в целом.

Что нового в этой версии?

Надо сказать, что изменений по сравнению с предыдущей (пятой) версией не так много. Основные изменения коснулись, как впрочем и ожидалось, работы с базами данных. Напомню, что основные отличия пятой версии Visual C++, в основном, касались программирования для Internet. Естественно, все лучшее, что было реализовано там, перешло и в новую версию. Следует сказать, что выпуск шестой версии произошел столь стремительно, что многие еще не успели как следует поработать с пятой. Поэтому коснемся и тех возможностей, которые появились в пятой версии.

Как мы только что отметили, Internet в этих версиях продукта играет едва ли не самую заметную роль. Все начинается непосредственно с инсталляции. Одно из окон мастера установки специально проверяет, установлена ли на вашем компьютере версия Internet Explorer 4. И попробуйте отказаться от его инсталляции. Не тут-то было — фирма Microsoft верна себе — если не хочешь пользоваться ее продуктами, не пользуйся ничем.

Примечание

Тех, кто работает в Windows NT 4, ждет еще один сюрприз — вы не сможете установить Visual C++, пока не инсталлируете как минимум Service Pack 3.

Поддержку Internet можно найти практически везде, включая IDE (Integrated Development Environment — интегрированная среда разработки). Ниже приведен обзор этих возможностей:

- *Классы сервера Internet.* Фирма Microsoft добавила пять новых классов, которые поддерживают ISAPI (Internet Server Application Programming Interface — Интерфейс программирования приложений сервера Internet). Эти классы помогают создавать фильтры сервера Internet и расширения в форме специальных DLL. Для облегчения работы предусмотрен специальный мастер ISAPI Extension Wizard (Мастер расширения ISAPI).
- *Win32 Internet.* Эти классы позволяют вам добавить поддержку Internet к любым приложениям; включает HTTP, FTP и сервисы протокола распространения электронной информации (gopher services).
- *Active X Controls Wizard (Мастер элементов управления Active X).* Эти элементы хорошо вам знакомы — фирма Microsoft произвела некоторые изменения, которые позволяют обычным элементам управления работать в Internet. Можно также создавать Active X Documents (документы Active X), которые предоставляют множество интерфейсов, что позволяет видеть документ внутри сетевого браузера.
- *ATL (Active Template Library, Библиотека активных шаблонов).* В Visual C++ 6.0 вы найдете обновленную версию ATL.
- *Cluster Resource Type Wizard (Мастер типа кластерных ресурсов).* Создает два проекта, отвечающих за создание динамически подключаемой библиотеки ресурса и динамически подключаемой библиотеки администратора кластеров (рис. 1.1).

Рис. 1.1. Первое окно мастера Cluster Resource Type Wizard

Перечисленные изменения в основном относятся к пятой версии компилятора. Они охватывают ту область программирования, которая чем дальше, тем требуется все больше. Управление базами данных — это целый комплекс решений задач по программированию. Visual C++ 6.0 содержит новые возможности для проектирования баз данных, которые действительно нужно знать, а именно: что они собой представляют и как можно их использовать:

- *Multithreading — ODBC (Многопоточный режим обработки).* Теперь можно создавать множество исполняемых потоков для приложений ODBC.
  К несчастью, фирма Microsoft при этом не дает гарантии безопасности потоков.
- *Класс CDatabase* предоставляет улучшенную поддержку транзакций. *Два новых метода GetCursorComitBehavior и GetCursorRollbackBehavior* исключают необходимость сохранения курсора. *Третий новый метод OpenEx* теперь предпочтителен для инициализации объекта CDatabase перед тем, как вы будете создавать объекты CRecordset.
- *Класс CRecordset предлагает сразу несколько усовершенствований.* Например, теперь можно получить и установить закладки, используя компоненты GetBookmark и SetBookmark. (Эти два компонента относятся к классу CDBVariant.) Используя класс CRecordset, можно также производить выборку строк больших объемов. Компонент Open тоже предоставляет несколько новых опций. И, наконец, CRecordset больше не имеет чисто виртуальных функций. Это означает, что можно непосредственно создавать объект CRecordset.

Рис.1.2. Первое окно мастера Extended Stored Proc Wizard

- *Поддержка DAO.* Теперь MFC поддерживает версию DAO 3.5. Это та версия, которая поставляется с Microsoft Office 97.
- *OLE DB.* Это множество стандартных интерфейсов, которые позволяют совместно использовать данные из множества источников информации без написания отдельного кода для каждого из них. Интерфейсы предоставляют уровень поддержки, соответствующий источнику данных, который вы хотите использовать.
- *Extended Stored Proc Wizard (мастер расширенных сохраненных процедур).* Использование этого мастера значительно облегчает создание расширенных сохраненных процедур для SQL Server (рис. 1.2).

Изменения коснулись и собственно языка программирования. Эти изменения помогают сохранить соответствие Visual C++ текущему стандарту:

- *bool, false, true (булевский, "ложь", "истина").* Ключевое слово bool позволяет вам определить переменную со значением "истина", либо "ложь". Вы можете перекодировать эту переменную в число, при этом вы получите значение 0, если значение переменной ложно, или 1, если оно истинно. Ключевое слово true устанавливает переменную в значение "истина", тогда как ключевое слово false устанавливает ее в значение "ложь".
- *Explicit (формальный).* Спроектированное для работы с объявлениями конструктора внутри класса, это ключевое слово делает невозможным использование конструктора в неявных преобразованиях.
- *Mutable (изменяемый).* Если вы объявляете, что компонент данных является видоизменяемым, вы можете присвоить ему значение из постоянного метода.
- *Typename (имя типа).* Это ключевое слово спроектировано для использования в определениях шаблонов. Оно сообщает компьютеру, что некоторый идентификатор является типом.
- *_declspec.* Это ключевое слово — новшество специфического языка фирмы Microsoft и используется для квалификации информации о классе хранения. Оно упрощает и стандартизирует специфические расширения Microsoft до языков C и C++. Существует множество различных атрибутов класса хранения, специфических для Microsoft, включая allocate, thread, naked, dllimport, dllexport, nothrow, и selectany.

Visual C++ предоставляет еще некоторые дополнительные обновления и изменения, которые нелегко отнести к какой-либо определенной области:

- *Поддержка механизма прокрутки с помощью мыши (Mouse Scroll Wheel Support).* Класс CScrollView теперь обеспечивает поддержку механизма прокрутки для мыши. Эта поддержка позволяет пользователю прокручивать документ без использования полосы прокрутки. Пользователь запрашивает прокрутку с помощью события WM_MOUSEWHEEL.

- *Утилита ERRLOOK.* Эту служебную программу можно использовать, чтобы отыскивать ошибочные запросы системы, которые включают HRESULT OLE, по их значению. Формат значений может быть либо десятичным, либо шестнадцатеричным, кроме того, можно вставлять полученный текст в приложения, если это необходимо. Эта утилита включает контекстно-зависимую справку, которая показывает вам, как ее использовать.
- *Сервер транзакций Microsoft.* Он предоставляет все, что вам необходимо для написания приложений на основе транзакций. Microsoft Transaction Server поддерживает программирование баз данных через интерфейс ODBC.
- *Визуальные инструменты работы с базами данных.* Любой, кто работал с Microsoft Access, знает о времясберегающем аспекте использования инструментов визуального проектирования при создании приложений баз данных. Визуальные инструменты для разработки баз данных, предоставляемые Enterprise Edition Visual C++, включают как Query Designer, так и Database Designer. Query Designer помогает получить или обновить данные в базе данных путем использования операторов SQL. Database Designer помогает создать множество различных объектов баз данных. Его устанавливают для работы с Microsoft SQL Server версии 6.5 (или выше).
- *Data View Pane (подокно отображения данных).* Это специальная область просмотра (подокно) в окне Workspace (рабочее пространство), которое позволяет вам работать со множественными источниками данных ODBC. Вы можете использовать Data View Pane для выполнения множества различных задач, подобных добавлению или удалению источника данных из проекта, а также восстановлению локальной копии источника данных из исходной версии. Data View Pane также подходит для таких действий, как выполнение сохраненной процедуры или сброс одного из источников данных.
- *Debugger Performance (производительность отладчика).* Debugger Performance позволяет улучшить эксплуатационные характеристики во время отладки соединения баз данных.
- *Расширенная поддержка типов баз данных SQL.* Теперь можно просматривать все типы данных SQL, включая текст и образ.

Перечислив основные глобальные особенности Visual C++ 6.0, рассмотрим некоторые параметры компилятора.

Настройки проекта

Для того чтобы определить настройки проекта, в меню **Project** (проект) выберите пункт **Settings** (настройки). На экране появится блок диалога **Project Settings** (настройки проекта) (рис. 1.3).

Рис. 1.3. Блок диалога **Project Settings**

Перейдите на вкладку с названием типа опций, которые вы хотите установить. Доступны вкладки **General** (основные), **Debug** (отладка), **C/C++** (настройки компилятора), **Link** (настройки компоновщика), **Resources** (ресурсы), **MIDL**, **Browser Info** (информация программы просмотра), **Custom Build** (обычное построение), **Pre-link step** (действия перед компоновкой) и **Post-build step** (действия после построения). Выбрав вкладки **C/C++** или **Link**, можно установить необходимые опции для, соответственно, компилятора и компоновщика.

Настройка опций компилятора

При выборе в раскрывающемся списке **Category** (категория) (рис. 1.4) категории опций, в блоке диалога будут отображаться опции, относящиеся к этой категории. Определены следующие категории опций:

- **General** — наиболее часто используемые опции
- **Language** — определяет представление указателей C++ на компоненты классов, управление обработкой исключений и управление созданием скрытых виртуальных конструкторов/деструкторов в классах, базирующихся на виртуальных
- **Code Generation** — определяет тип процессора, библиотеку времени выполнения, соглашения о вызовах и выравнивание элементов структур
- **Customize** — отключает расширения языка Microsoft, допускает компоновку на уровне функций, устраняет дублирование строк, допускает минимальное перепостроение проекта, пошаговую компиляцию, отключает сообщения при запуске

Рис. 1.4. Вкладка **C/C++** блока диалога **Project Settings**

- **Listing Files** — создает файлы с информацией для просмотра (BSC) и файлы с листингом кода
- **Optimizations** — определяет, как компилятор настраивает скоростные показатели работы прикладной программы
- **Precompiled Headers** — позволяет ускорить время компиляции, разрешая прекомпиляцию кода C или C++
- **Preprocessor** — управляет символами, макросами и включает пути, используемые препроцессором C/C++

Ниже описана категория наиболее часто используемых опций — **General**:

- **Warning level** — устанавливает уровень выдачи предупреждений об ошибках (только для файлов с исходными текстами, не для объектных (OBJ) файлов)
  - **None** — отключает все предупреждения. Параметр командной строки /W0 или /w
  - **Level 1** — выводит только строгие предупреждения. Параметр командной строки /W1
  - **Level 2** — выводит менее строгие предупреждения, такие как использование функций без объявления типа возвращаемого значения, ошибка передачи возвращаемого значения для функции, тип возвращаемого значения которой не void, ошибки преобразования данных, которые могут привести к потере данных или точности. Параметр командной строки /W2
  - **Level 3** — выводит еще менее строгие предупреждения, такие как вызов функций до их объявления. Параметр командной строки /W3

- **Level 4** — выводит наименее строгие предупреждения. Параметр командной строки /W4

❒ **Warnings as errors** — останавливать компиляцию в случае появления предупреждений. Параметр командной строки /WX

❒ **Generate browse info** — предписывает создавать SBR-файлы с полной информацией программы просмотра. В процессе построения проекта утилита BSCMAKE использует SBR-файлы для создания BSC-файла, который можно открыть в окне просмотра. Параметр командной строки /FR

❒ **Debug info** — определяет, какой тип информации создается для отладки и где она хранится (в объектных файлах или в базе данных программы PDB)

- **None** — без информации для отладки, компиляция производится быстрее. Нет параметров для командной строки
- **Line Numbers Only** (только номера строк) — создает объектные (OBJ) файлы или исполняемые (EXE) файлы, содержащие только глобальные и внешние переменные и информацию о номерах строк. Используйте эту опцию, если только хотите уменьшить размер EXE-файла или не собираетесь использовать вычисление значений выражений при отладке. Параметр командной строки /Zd
- **C7 Compatible** (совместимая с C7) — производит объектные (OBJ) и исполняемые (EXE) файлы, содержащие номера строк и полную символьную информацию для отладчика. Параметр командной строки /Z7. (Имеет то же значение, что и /Zi в версии 7 Microsoft C/C++.)
- **Program Database** (база данных программы) — производит базу данных программы (PDB), которая содержит символьную информацию для отладчика. Параметр командной строки /Zi. Если вы используете эту опцию, то размеры объектных файлов уменьшаются, так как информация для отладки содержится не в них, а в файле базы данных программы

❒ **Optimizations** (оптимизация) — позволяет настроить показатели эффективности программы. Четыре типа опций оптимизации из пяти (**Default**, **Disable (Debug)**, **Maximize Speed** и **Minimize Size**) в раскрывающемся списке не требуют дополнительной настройки. Вариант **Customize** позволяет более точно настроить параметры оптимизации программы

❒ **Preprocessor definitions** (определения препроцессора) — позволяет задать один или несколько макросов для работы препроцессора

Настройка опций компоновщика

При выборе в окне **Project Settings** вкладки **Link** (рис. 1.5) можно задать параметры для 32-разрядного компоновщика (LINK.EXE).

Рис. 1.5. Вкладка **Link** блока диалога **Project Settings**

Доступны следующие категории опций:

- **General** — наиболее часто используемая категория опций
- **Customize** — управляет процессом компоновки
- **Debug** — управляет процессом создания отладочной информации
- **Input** — управляет параметрами компоновщика, необходимыми при использовании библиотек
- **Output** — управляет компоновщиком при создании приложения Win32

Ниже описаны опции категории **General**:

- **Output file name** (имя выходного файла) — переопределяет заданное по умолчанию имя программы и ее местоположение. Параметр командной строки: /OUT:*имя_файла*
- **Object/library modules** (объектные/библиотечные модули) — передает компоновщику объектные файлы или стандартную библиотеку (статическую или импортируемую). Параметр командной строки: *имя_файла*
- **Generate debug info** (генерировать отладочную информацию) — предписывает создавать отладочную информацию для EXE- или DLL-файлов. Параметр командной строки /DEBUG
- **Link incrementally** (компоновать по шагам) — определяет пошаговую компоновку. Параметр командной строки /INCREMENTAL:{YES|NO}
- **Enable profiling** (разрешить измерение временных характеристик) — создает на выходе файл, который может использоваться профайлером (ути-

литой для измерения временных характеристик программы). Параметр командной строки /PROFILE

- **Ignore all default libraries** (игнорировать все библиотеки, заданные по умолчанию) — предписывает компоновщику не учитывать все библиотеки, заданные по умолчанию, при разрешении им внешних ссылок. Параметр командной строки /NODEFAULTLIB

Остальные опции проекта изменяются значительно реже, и мы не будем на них останавливаться, а перейдем к знакомству с интегрированной средой разработки.

Глава 2

Интегрированная среда разработки (IDE)

В IDE (Integrated Development Environment — интегрированная среда разработки) произошло много различных изменений. Правда, справедливости ради, следует сказать, что чисто внешне это малозаметно. Многие из изменений скорее направлены на улучшение сервиса. В других случаях — это совершенно новая возможность, которую не сразу и обнаружишь. IDE работает в фоновом режиме, чтобы сделать написание приложений более быстрым и более эффективным. Ниже дан обзор изменений IDE, многие из которых отражены во всплывающих подсказках и примечаниях:

- *Compiler Support for COM* (поддержка компилятора для COM). В этой версии Visual C++ вы найдете расширенную поддержку COM. Фактически, фирма Microsoft заготовила некоторые дополнительные образцы, которые демонстрируют "родную" поддержку COM компилятором в дополнение к пакету программ.
- *Optimazed Code Generation* (формирование оптимизированного кода). Компиляция с опцией /O1 уменьшит размер кода на 5—10%, дополнительный выигрыш даст то, что компоновщик автоматически использует опцию /FIXED, создав исполняемый файл без перераспределения сегмента памяти (relocation section). Однако если вы планируете использовать профилировщик или внешний отладчик, подобный Bounds Checker, то убедитесь, что вы определили переключатели /FIXED:NO или /PROFILE. И, наконец, изменились опции /G3, /G4, /G5, /G6, /GB. Эти опции компилятора задают, для какого процессора, выполняющего задание, будет генерироваться ваш код. Например, опция /G6 сообщает компилятору, что вы хотите дать задание процессору Pentium Pro.
- *Exception Handling* (обработка исключений). Новая опция компилятора /EH позволяет установить желаемый тип обработки исключений (/Ена, например, сообщает компилятору, что вы хотите использовать асинхронную обработку исключений). Опция компилятора /GX теперь преобразуется в опцию компилятора /EHsc, которая специально сообщает компи-

лятору о вашем желании использовать синхронную обработку исключений и принять допущение, что внешние функции C никогда не создадут исключительную ситуацию.

- *Debug Information Storage* (хранилище отладочной информации). Используйте опции компоновщика /PDBTYPE (хранение базы данных программы) для определения того, куда Visual C++ должен помещать типичную отладочную информацию: /PDBTYPE:CON предписывает компоновщику разместить отладочную информацию внутри одного файла PDB; /PDBTYPE:SEPT, помещает отладочную информацию в отдельные файлы компилятора PDB. Преимущество второго метода заключается в том, что вы будете использовать меньше дискового пространства для хранения отладочной информации, и отладчик будет запускаться быстрее. Недостаток — отладчику для получения отладочной информации нужно будет просмотреть большое количество файлов.
- *Batch-Mode Processing* (обработка в пакетном режиме). NMake поддерживает обработку файлов в пакетном режиме. Большое преимущество этой возможности — вы только однажды активизируете компилятор, чтобы создать все объектные файлы. При изменениях в исходном коде программы теперь нет необходимости заново создавать объектные файлы, что приводит к сокращению общего времени компиляции.
- *Dialog Class (OLE) Automation* (класс диалога автоматизации OLE). Мастер AppWizard теперь поддерживает добавление Automation (автоматизации) к приложениям на основе диалогового окна. (В Microsoft термин "OLE Automation" сократили до "Automation".) Теперь вы также можете добавлять методы и свойства, чтобы показать элементы диалогового окна.
- *Custom AppWizard Update* (обновленный специализированный мастер AppWizard). Специализированные мастера AppWizard могут теперь модифицировать опции построения для проектов, которые они создают. Это опции браузера, компоновщика и компилятора. Вы можете также приказать AppWizard произвести специализированные этапы построения, которые будут выполняться после того, как объект назначения уже построен.

В Developer Studio (студия разработчика) также появились изменения, многие из которых связаны с включением компанией Microsoft других языков программирования, подобных Java (Visual J++); другие являются следствием ориентации Visual C++ на Internet:

- *Макросы и автоматизация*. Теперь доступна автоматизация определенных задач при написании кода в Visual C++ подобно тому, как это делалось в текстовых процессорах. Может показаться странным применение VBScript для автоматического выполнения задач при работе с Visual C++, но это язык, без которого вам все равно не обойтись. Microsoft предоставляет полный доступ ко всем объектам Developer Studio, так что вы можете полностью манипулировать вашей средой программирования.

- *Классы Java.* Developer Studio теперь поддерживает классы Java, поэтому вы можете просматривать их совместно с вашими классами C++. Окно отображения классов ClassView, используемое для просмотра классов, позволяет вам использовать папки, чтобы хранить все в организованном порядке.
- *Объекты COM.* ClassView отображает любой объект COM, созданный с использованием MFC или ATL.
- *Настраиваемые панели инструментов и меню.* Существует много способов настройки вашего окружения с использованием Developer Studio. Вы можете добавить или удалить панели инструментов с экрана. В случае необходимости, Developer Studio позволяет создавать новые панели инструментов или меню. Developer Studio позволяет даже перемещать элементы меню на панель инструментов или размещать в меню пиктограммы панели инструментов. В сущности, вы не ограничены в своих действиях, и можете настроить вашу среду программирования так, как хотите.
- *Отладчик.* Теперь можно вносить изменения в код приложения не выходя из отладчика.
- *Возможности подключения к Internet.* Из пятой версии привнесена специальная панель инструментов InfoView — элементы управления для Internet. С ее помощью можно, например, открыть специальную страницу в Internet, посвященную выбранной теме. Эта возможность гарантирует получение самой последней информации о Visual C++. Вы можете также использовать окно с раскрывающимся списком **Current URL** (текущие URL), чтобы ввести Internet-адрес. Скажем, вам необходимо просмотреть сайт Microsoft, чтобы найти текущие изменения, — теперь вы можете делать это, не выходя из Visual C++.
- *Рабочие области.* Теперь вы можете поместить различные типы проекта в одну рабочую область, которая использует расширение имени файла — DSW. Старое (до версии 5.0) расширение рабочей области было MDB. Кроме этого, новые файлы рабочей области не содержат специфических для вашего локального компьютера данных, это намного облегчает совместное использование файлов.
- *Внутренние и внешние проекты.* В настоящее время существует два вида проектов: внутренние и внешние. Файлы внешнего проекта по-прежнему используют расширение файла MAK. Новые внутренние проекты используют расширение DSP. Самое большое отличие между внутренним и внешним проектом заключается в том, что внутренний проект не совместим с утилитой NMake.
- *Поддержка документов ActiveX.* В некоторых случаях лучше скомпоновать форму или документ иного вида в текстовом или электронном табличном процессоре, чем создавать его в Visual C++ с самого начала. Поддержка

документов ActiveX означает, что теперь вы можете сразу добавить этот документ в ваш проект. Полный перечень типов документов ActiveX, инсталлированных на вашей машине, вы найдете на странице **Other Documents** диалогового окна **New**. Конечно, если в проекте использован документ ActiveX, то на машинах тех, кто работает с этим документом, должно быть инсталлировано соответствующее приложение.

- *WizardBar* (панель мастера). Новая панель WizardBar позволяет быстро создавать соединения между вашим кодом и визуальными элементами. Кроме того, вы можете использовать WizardBar с Visual J++.
- *Find Dialog* (диалоговое окно поиска). Теперь можно искать не только слова. Visual C++ позволит найти оперативные клавиши, строки ASCII, шестнадцатеричные байты, идентификаторы или метки элементов управления.
- *Изменение нескольких свойств*. Теперь есть возможность изменить общее свойство для нескольких объектов. Для этого необходимо выбрать все эти объекты, а затем использовать команду **View\Properties**, чтобы вывести на экран диалоговое окно **Properties**, содержащее свойства, которые являются общими для всех этих объектов. (Вы также можете щелкнуть правой кнопкой мыши на выделенных объектах и выбрать команду **Properties** контекстного меню.) Изменение какого-либо свойства будет применено ко всем выделенным объектам.

Рис. 2.1. Новое контекстное меню

- *Цветовое выделение ключевых и служебных слов.* В этой версии Visual C++ значительно шире используются цветовые выделения. Например, можно окрашивать выбранные поля, чтобы различать элементы управления и разделы данных кода. Это очень удобно при редактировании кода, в который не принято включать множество отступов, подобно HTML.
- *Новое контекстное меню.* Его содержимое меняется в зависимости от объекта, на котором нажата правая кнопка мыши (рис. 2.1). С его помощью теперь можно получить список компонентов класса, тип возвращаемого функцией значения, информацию о параметрах и многое другое.
- *Текущая подсказка.* Наконец-то отпала необходимость заглядывать в справочную систему, чтобы определить доступные компоненты класса (рис. 2.2) или аргументы функции (рис. 2.3). При наборе после имени объекта оператора доступа к компоненту класса или открывающей скобки функции появляется всплывающая подсказка.
- *Комбинированное диалоговое окно* ***New***. Теперь Visual C++ использует одно диалоговое окно для новых проектов, файлов и рабочих областей. Имеется также вкладка для других видов файлов. Используя комбинированные диалоговые окна, вы можете делать нечто подобное созданию объекта и вставлять его в проект. Вот как выглядит обновленное диалоговое окно **New** (рис. 2.4).

Рис. 2.2. Теперь можно выбрать необходимый компонент, не выходя из режима редактирования

Рис. 2.3. Появившаяся всплывающая подсказка существенно облегчает задание параметров функций

Рис. 2.4. Диалоговое окно **New**

Как видите, к 14 типам проектов предыдущей версии Visual C++ добавлено 3 новых типа: Cluster Resource Type Wizard, Extended Stored Proc Wizard и Utility Project.

Более подробно работа с некоторыми типами проектов будет рассмотрена в следующих главах. Знакомство с интегрированной средой разработки закон-

чим описанием некоторых возможностей, предоставляемых многочисленными мастерами.

Консольное приложение получило своего мастера (рис. 2.5) для создания заготовки:

- Как и раньше, можно создать пустой проект
- Новая возможность — создание каркаса простого приложения. Удобства здесь минимальные, но тем не менее они есть
- Традиционное приложение "Hello, World!" (мало отличается от предыдущего)
- Ценители языка программирования C++ теперь могут автоматически получить поддержку библиотеки классов MFC

Удивительно, насколько эти минимальные возможности (в отличие от предоставляемых мастерами Windows-приложений) улучшают восприятие программного продукта. Но переходим к полноценным приложениям для Windows.

Рис. 2.5. Окно нового мастера консольных приложений

Здесь тоже есть изменения. Прежде всего, можно отключить поддержку безусловно удобной, полезной, мощной, но... иногда (довольно часто) ненужной архитектуры "документ/представление" (Document/View) (рис. 2.6).

Создание проекта с отключенной поддержкой этой архитектуры мы рассмотрим более подробно позже.

Следующее существенное дополнение коснулось четвертого окна мастера (рис. 2.7) — появился переключатель **Internet Explorer ReBars**, который изменяет вид панелей инструментов, делая его совместимым с Internet Explorer.

Рис. 2.6. Первое окно мастера MFC AppWizard

Рис. 2.7. Четвертое окно мастера MFC AppWizard

Дополнение есть и в следующем окне мастера (рис. 2.8), где теперь можно выбрать тип проекта: стандартный (**MFC Standard**) или в стиле Windows Explorer.

И вот что мы получаем, когда построим приложение (рис. 2.9).

Примечание

Начиная с Visual C++ 5, вместо терминов "компиляция" и "компоновка" используется термин "построение" (build).

Рис. 2.8. Пятое окно мастера MFC AppWizard

Рис. 2.9. Ничего не делая, мы получаем полноценное приложение

Конечно же, следует отметить и некоторые недостатки:

- Время построения приложения несколько увеличилось — но это традиционная беда всех компиляторов языков C и C++

- Размер исполняемых файлов также увеличился, хотя и незначительно
- По непонятным причинам при создании каркаса приложения не все классы появляются в мастере ClassWizard. И чтобы исправить это положение, необходимо закрыть интегрированную среду разработки, удалить файл с расширением CLW, снова загрузить проект и вызвать мастер, который любезно спросит вас, какие файлы вы хотите включить (точнее, *исключить*, поскольку автоматически включены все файлы). Только после этого можно использовать всю мощь мастера ClassWizard

В заключение обратимся к основной теме книги и перечислим те изменения, которые коснулись непосредственно библиотеки классов MFC:

Класс	Состояние	Описание
CComboBoxEx	Новый	Улучшенный раскрывающийся список с поддержкой списков изображений
CDateTimeCtrl	Новый	Поддержка расширенного редактора текста и/или элемента управления "календарь", с возможностью выбора даты и/или времени
CHeaderCtrl	Измененный	Поддержка работы с битовыми массивами и операций перетаскивания
CHtmlView	Новый	Поддержка элемента управления для Web-браузера непосредственно в приложении; поддержка Dynamic HTML
CHttpServer	Измененный	Новые функции для записи данных на сервер
CHttpServerContext	Измененный	Новые функции управления размером блоков при записи данных на сервер
CImageList	Измененный	Поддержка изображений, используемых теперь в различных элементах управления
CInternetSession	Измененный	Поддержка cookies
CIPAddressCtrl	Новый	Поддержка работы поля редактирования с IP-адресами
CListCtrl	Измененный	Группировка элементов с использованием "рабочих областей"
CMenu	Измененный	Установка элемента меню по умолчанию и получение подробной информации об элементе меню
CMonthCalCtrl	Новый	Элемент управления "календарь"
COleClientItem	Измененный	Новые функции для получения дескриптора пиктограммы, ассоциированной с отдельным CLSID сервера

(продолжение)

Класс	Состояние	Описание
COleDBRecordView	Новый	Поддержка механизма OLE DB в представлении
COleDocObjectItem	Новый	Поддержка активных документов
COleDateTime	Измененный	Поддержка преобразования даты и/или времени в формат, совместимый со структурой SYSTEMTIME из Win32
CProgressCtrl	Измененный	Поддержка сплошного заполнения и вертикального расположения элемента управления
CPropertySheet	Измененный	Поддержка возможности прокрутки страниц свойств
CPropertySheetEx	Новый	Поддержка стиля Wizard97 набора страниц (недоступен в Windows 95)
CPropertyPageEx	Новый	Поддержка стиля Wizard97 страниц свойств (недоступен в Windows 95)
CReBar	Новый	Расширенная панель инструментов, которая может включать дополнительное дочернее окно
CReBarCtrl	Новый	Расширенный элемент управления "панель инструментов"
CSliderCtrl	Измененный	Поддержка всплывающей подсказки для поля с ползунком
CSpunButtonCtrl	Измененный	Поддержка 32-битного диапазона спина
CStatusBar	Измененный	Поддержка всплывающих подсказок, пиктограмм и изменения цвета фона для строки состояния
CStatusBarCtrl	Измененный	Расширенный элемент управления "строка состояния"
CString	Измененный	Улучшенная поддержка поиска, замены, удаления, вставки и форматирования
CTabCtrl	Измененный	Поддержка новых стилей и возможности вставки новой вкладки в уже существующую
CTime	Измененный	Поддержка преобразования даты и/или времени в формат, совместимый со структурой SYSTEMTIME из Win32
CToolBar	Измененный	Поддержка прозрачного и плоского стилей панели инструментов, характерных для Internet Explorer 4

(окончание)

Класс	Состояние	Описание
CToolBarCtrl	Измененный	Элемент управления "панель инструментов"
CToolTipCtrl	Измененный	Реализована поддержка обновления всплывающих подсказок и их атрибутов
CTreeCtrl	Измененный	Поддержка флажков и автоматического режима "одного узла"

На этом мы заканчиваем краткое знакомство с новыми и/или улучшенными возможностями Visual C++ 6 и переходим непосредственно к программированию.

ПРОГРАММИРОВАНИЕ ПОД MS WINDOWS

Глава 3

Знакомство с Windows

Сегодня трудно даже вспомнить то время, когда не было Windows. И как бы программисты ни относились к корпорации Microsoft и ее продуктам, приходится признать, что Windows — это настоящее, а может быть и будущее компьютерной индустрии.

Как известно, успех компьютера и операционной системы определяется количеством и разнообразием программного обеспечения (ПО), созданного для них. К 1990 году, когда ПО для персональных компьютеров достигло критической массы (не в последнюю очередь за счет программ самой фирмы Microsoft), как нельзя кстати появилась новая версия Windows — 3.0. Именно с этого момента начинается победное шествие по миру операционной оболочки, а затем и полноценной системы Windows.

Остановимся на основных чертах, присущих 32-разрядным операционным системам Windows 95/98 и Windows NT.

Чтобы оценить возможности и достоинства Windows, необходимо, прежде всего, перечислить требования к операционным системам, которые сформировались в результате создания и использования нескольких десятков операционных систем для компьютеров всех классов и поколений за последние 40 лет (да, за это время были созданы именно десятки, если не сотни систем!). И хотя сейчас лидерство по количеству проданных и установленных копий безусловно принадлежит Windows, кто сказал, что к лидеру нужно предъявлять менее жесткие требования, чем к остальным?

Пользовательский интерфейс

Первое, что видит пользователь, включив компьютер с ОС Windows, — это графический пользовательский интерфейс, а не командная строка или что-то вроде Norton Commander. Разработчики системы стремились сделать ее не только удобной и интуитивно понятной для работы, но и чтобы про-

граммы третьих фирм по своему пользовательскому интерфейсу как можно меньше отличались от продукции фирмы Microsoft. Для этого операционная система поддерживает множество объектов, на основе которых можно реализовать большую часть проектных идей.

Окна, элементы управления, модальные и немодальные блоки диалога

В этом разделе описана терминология Windows (рис. 3.1 и 3.2).

Окно (Window) — прямоугольная область экрана для организации обмена между пользователем и приложением. Окно совместно использует экран с другими окнами, в том числе и с окнами других приложений. Одновременно пользователь может осуществлять ввод только в одно окно и приложение, к которому это окно относится.

Диалоговое окно, или *блок диалога* (Dialog Box) — временное окно, создаваемое для запроса ввода пользователя. Обычно в блоке диалога вводится дополнительная информация для функционирования приложения. Блоки диалога содержат количество элементы управления (см. ниже), позволяющие ввести текст, выбрать режим и т. д. Диалоги могут быть как *модальные*, так и не *модальные*. Модальные диалоги служат для ввода пользователем некоторых данных или ответа на вопрос, необходимого для продолжения работы. Другие элементы (окна) приложения при этом недоступны. Немодальные диалоги служат для постоянного отображения на экране, например для быстрого изменения какого-либо параметра и немедленной реакции приложения на него.

Элемент управления (Control) — дочернее окно, которое приложение использует для взаимодействия с другим окном, обеспечивая простейшие операции ввода/вывода. Чаще всего элементы управления используются в окнах диалога, хотя их можно использовать и при работе с окнами других типов. Ниже перечислены стандартные элементы управления.

Кнопка (BUTTON) — обычно уведомляет, что пользователь выбрал этот элемент. Различают несколько видов кнопок: обычная, срабатывающая по нажатию (PUSH BUTTON), переключатель (RADIO BUTTON), флажок (CHECK BOX) и группа (GROUP BOX).

Текстовое поле (EDIT) — простейший редактор, позволяющий пользователю просматривать и редактировать текст.

Список (LIST BOX) — элемент отображения списка элементов, позволяет пользователю выбрать один или несколько его элементов.

Комбинированный список (COMBO BOX) — комбинация списка и однострочного текстового поля. Позволяет пользователю выбрать и редактировать элемент списка.

Полоса прокрутки (SCROLL BAR) — элемент управления, позволяющий пользователю выбрать направление и расстояние прокрутки информации в связанном с элементом окне.

Статический элемент (STATIC) — элемент, исключающий воздействие на него со стороны пользователя, обычно используется в качестве надписи для других элементов управления.

Рис. 3.1. Основные и дополнительные элементы окна

Операционная система предоставляет несколько путей для создания пользовательских элементов управления:

1. Для кнопок, списков и комбинированных списков можно использовать стиль *самоотображения* (owner-draw), который требует написания кода для самостоятельного прорисовывания элемента.
2. Наследование стандартного элемента управления с заменой функции окна, в которой можно предусмотреть любые интересующие возможности, оставляя возможность использования части кода стандартной реализации.
3. Создание своего собственного класса окна.

Кроме *стандартных элементов управления* (controls), ОС Windows поддерживает большое количество так называемых *общих элементов управления* (common controls). Поддержка осуществляется специальной динамически загружаемой библиотекой. Как и остальные элементы управления, описываемые элементы представляют собой дочерние окна, которые приложение

использует для взаимодействия с другими окнами, обеспечивая простейшие операции ввода/вывода. Отличие от обычных элементов управления состоит в их большем разнообразии и функциональности. Кроме того, целый ряд элементов активно используются помимо блоков диалога и в окнах других типов. Ниже перечислены стандартные элементы управления.

Рис. 3.2. Виды элементов управления в блоке диалога

Изменяемый список (DRAG LIST BOX) — специальный тип списка, позволяющий перетаскивать его элементы, меняя порядок отображения.

Заголовок (HEADER) — дочернее окно, обычно располагаемое над колонками текста или цифр. Содержит заголовок для каждого столбца и позволяет менять ширину столбца.

Просмотр списка (LIST VIEW) — мощное средство отображения элементов списка. Позволяет отображать кроме текста значки, связанные с элементами списка. Поддерживает четыре способа отображения. Три из них располагают элементы с значками по всему полю отображения, четвертый способ позволяет создать список, состоящий из набора столбцов, имеющих заголовки (рис. 3.3).

Строка состояния (STATUS LINE) — горизонтальное окно в нижней части родительского окна, служащее для отображения различной информации приложения, например, подсказки для выбранного пункта меню. Строка состояния может содержать несколько полей для отображения координат, времени, индикации нажатия специальных клавиш (<Ins>, <Scroll Lock>, <Num Lock>) и т. д.

Панель инструментов (TOOLBAR) — дочернее окно, состоящее из набора кнопок быстрого вызова тех или иных команд.

Подсказка (TOOLTIP) — небольшое всплывающее окно с текстом краткого описания некоторого элемента управления, например, кнопки (рис. 3.4).

Рис. 3.3. Заголовок и просмотр списка

Рис. 3.4. Список в режиме крупных значков, панель инструментов, строка состояния, подсказка

Просмотр дерева (TREE-VIEW) — окно просмотра данных, организованных в виде дерева, например содержимого диска или директория. Каждый элемент дерева может, кроме текста, иметь и некоторое растровое (битовое) изображение. Узлы дерева можно раскрывать и закрывать, т. е. отображать и прятать элементы узла (рис. 3.5).

Вкладка (TAB) — средство, позволяющее создавать элементы, аналогичные закладкам, которые используются при чтении книги. Вкладки используются для смены страниц отображения в одной и той же области диалогового окна. Специальный тип вкладок позволяет отображать каждую вкладку аналогично кнопке, выбор которой приведет к выполнению команды вместо смены страницы.

Окно свойств (PROPERTY SHEET) — окно, позволяющее просмотреть и редактировать свойства некоторого объекта. Достоинства этого элемента управления по сравнению с обычным диалоговым окном в том, что большое количество параметров можно сгруппировать, а доступ к каждой такой группе осуществлять, используя вкладки (TAB). В то же время на базе окна свойств можно строить так называемые *мастера*, где переход от одного этапа запроса пользователя к последующему или предыдущему осуществляется при помощи соответствующих кнопок. Создание каждого отдельного окна для группы свойств или группы запросов (для мастера) заключается в подготовке ресурса диалога.

Горячая клавиша (*сокращение*, *клавиша вызова*) (HOT-KEY) — позволяет пользователю задать комбинацию клавиш, которая в дальнейшем может

быть использована для быстрого выполнения команды. Этот элемент отображает пользовательский выбор и проверяет его допустимость.

Рис. 3.5. Окно свойств со вкладками и просмотром дерева

Элемент редактирования с форматированием (RICH EDIT) — окно редактирования. В отличие от предопределенного элемента EDIT, редактор RTF-формата позволяет управлять шрифтами, цветом, размером, выравниванием абзацев. Кроме того, этот элемент управления поддерживает работу с внедренными объектами OLE.

Линейка с ползунком (TRACKBAR) — элемент, состоящий из ползунка и, возможно, градуированной линейки. Положение ползунка изменяют при помощи клавиатуры или мыши. При этом ползунок посылает уведомляющие сообщения об изменении своего состояния.

Просмотр видеоклипов (ANIMATION) — этот элемент позволяет просматривать видеоклипы, состоящие из набора битовых изображений и записанные в формате Audio Video Interleaved (AVI). Этот элемент пригоден для отображения только клипов, не содержащих звуковое оформление (рис. 3.6).

Индикатор (PROGRESS BAR) — средство индикации некоторого продолжающегося процесса или операции, например копирования файлов. Реализуется в виде прямоугольника, постепенно заполняемого от начала до завершения операции.

Счетчик (UP-DOWN) — пара кнопок со стрелками для увеличения или уменьшения значения, связанного со счетчиком элемента управления, например текстового поля, содержащего числовую информацию.

Рис. 3.6. Визуализация процесса копирования при помощи элемента просмотра видеоклипа и индикатора

Список рисунков (IMAGE LIST) — вспомогательный элемент, используемый, прежде всего, другими общими элементами управления. Представляет собой список совпадающих по размеру значков и битовых массивов, доступ к которым осуществляется по индексу. Используется с элементами "просмотр списка" и "просмотр дерева", для хранения пиктограмм вкладок и т. д.

Все вышеперечисленные стандартные элементы управления, наверное, каждому вполне знакомы, т. к. многие из них использовались еще в рамках Windows 3.*x*. Представьте, как трудно было программистам реализовать их без нынешней поддержки операционной системы.

Ресурсы приложения

Каждое приложение обычно имеет целый ряд стандартных или пользовательских ресурсов. *Ресурсами* приложения называются массивы данных, которые разработчик может добавить в исполняемый модуль. Ниже перечислены стандартные ресурсы:

- Акселераторы (accelerators) — структуры данных, содержащие списки горячих клавиш и команд, ассоциированных с ними
- Битовые массивы (bitmaps) — массивы точек для отображения на экране в качестве, например, пунктов меню или элементов управления
- Курсоры (cursors) — битовые массивы, используемые в качестве растровых изображений указателя мыши
- Шаблоны диалогов (dialogs) — структуры, описывающие окно диалога, включая элементы управления, оконные стили, положение на экране и другие параметры. Используются для вывода диалоговых окон на экран
- Значки (icons) — битовые массивы, используются для визуального представления различных объектов в системе — приложений, документов, элементов списков некоторых типов и т. д.

- Шаблоны меню (menus) — описывают пункты меню, связанные с задаваемыми пользователем командами
- Строковые таблицы (string tables) — списки статических символьных массивов
- Шаблоны панелей инструментов (toolbars) — описывают элементы управления, состоящие из набора битовых массивов в качестве кнопок, каждая из которых связана с определенной командой
- Описания версий (version) — статические структуры специального вида, содержащие информацию о программе, включая имя приложения, номер версии, информацию об авторских правах и т. п. Используются операционной системой для вывода информации о программе

Кроме стандартных, разработчик может создавать и любые пользовательские ресурсы, однако в этом случае он не сможет пользоваться стандартными функциями доступа к ним, т. к. системе неизвестна их семантика.

Использование термина "ресурсы" может показаться несколько неожиданным, тем не менее приложения работают с вышеперечисленными объектами действительно как с ресурсами (причем разделяемыми), потому что несколько копий приложения могут пользоваться только одной копией некоторого ресурса.

Дополнительное преимущество использования ресурсов приложения в том, что все они сосредоточены в файле (файлах), являющемся отдельной единицей компиляции, что чрезвычайно полезно, например, при наличии большого количества строковых сообщений, которые в случае их реализации в виде ресурсов не разбросаны по различным файлам.

Особенно полезно использовать ресурсы для создания шаблонов диалогов, шаблонов меню и акселераторов.

Доступ к ресурсам осуществляется по числовым или строковым идентификаторам.

Система, управляемая сообщениями

Многие основные черты Windows обязаны своей спецификой тому факту, что она является операционной системой, основанной на сообщениях. Не разобравшись в механизме работы сообщений, вы не поймете и Windows. Такой подход к программированию был впервые реализован в операционной системе для компьютеров Macintosh и заключается он в том, что поведение (и реакция) программы определяется внешними событиями.

В отличие от традиционного "последовательного" программирования, программирование для Windows является объектно-ориентированным. И неважно, что для этого могут быть использованы вовсе не объектно-ориенти-

рованные языки. (Объектно-ориентированные языки обеспечивают взаимодействие между объектами при помощи функций-сообщений.) Говоря упрощенно, систему Windows можно представить в виде набора объектов, с которыми приходится сталкиваться и при работе, и при программировании. Самым важным таким объектом является *окно*. Оно первым появляется при запуске системы или любого приложения и последним исчезает при завершении работы. В процессе работы Windows сообщает приложению, точнее окну приложения, о некотором событии при помощи *сообщения*, которое описывает изменение, произошедшее в окружающей приложение среде. Кроме этого, механизм сообщений для приложений — единственный способ "общения" с их окнами. Происходит это не напрямую, а через операционную систему, которая, получив сообщение, передает его нужному объекту (окну). Сообщения используются для настройки, изменения и диагностики состояния всех окон. Сообщение — это вход в приложение. Оно представляет собой событие, на которое, при необходимости, должно реагировать приложение. Каждое сообщение связывается с *конкретным* окном, с каждым из которых, в свою очередь, связана собственная *оконная процедура* — специальная функция, отвечающая за обработку поступающих сообщений. При этом все сообщения, источником которых служат аппаратные средства, система помещает в определенную структуру данных — *первичную входную очередь (*raw input queue*)*.

Примечание

В версиях Windows 3.*x* и более ранних система поддерживала только одну очередь сообщений, из которой их и выбирали приложения. В этих системах реализована так называемая *невытесняющая многозадачность*. Однако при этом возникали проблемы при сбое в активном приложении, когда его невыбранные из очереди сообщения не давали возможности другим приложениям получить "свои" сообщения.

В 32-разрядных операционных системах Windows реализован метод поддержки индивидуальных очередей сообщений, который называется *десинхронизацией ввода* (input desynchronization), при котором новые сообщения лишь ненадолго попадают в первичную входную очередь 32-разрядного приложения, после чего перенаправляются в определенную очередь конкретного потока. Упрощенно механизм можно описать следующим образом. В самом начале своей работы 32-разрядное приложение получает от системы единственную очередь сообщений, которая относится к его первичному потоку. И даже если процесс заводит еще один поток, система не создает для него сразу же еще одну очередь. Такое положение длится до тех пор, пока дополнительный поток данного приложения впервые не направит в систему относящийся к очереди запрос. Только после этого создается дополнительная очередь. Если потоку очередь сообщений не нужна, то система очередь вообще не создает, чтобы не тратить ресурсы.

Примечание

Вполне возможно, что начинающие программисты сразу не смогут разобраться в обилии новых терминов. Пусть их это не пугает. По мере чтения все эти и многие другие вопросы будут рассмотрены подробно.

Каждое окно в Windows имеет специальную функцию, которая вызывается для обслуживания сообщений. Операционная система только посылает сообщение окну, а оно уже само решает, что делать с этим сообщением. Точнее, программист должен "научить" окно, как отвечать на получаемые сообщения. На всем протяжении этого процесса окно действует как автономный объект, живущий своей собственной жизнью.

Следует сказать, что Windows API имеет в своем составе сотни (если не тысячи) функций, но есть одна, которая по значимости не уступает всем вместе взятым. Именно она используется для посылки сообщений объектам системы, естественно, если известны их идентификаторы. Речь идет о функции SendMessage. Основная ее задача, что видно и из названия, — послать сообщение некоторому объекту (возможно, с дополнительными данными), дождаться реакции этого объекта на сообщение, обычно в виде числа, которое и вернуть системе. Вроде бы все просто и ясно. Объектом может быть не только окно приложения, документа, блока диалога и т. д., но и любой элемент управления, например, кнопка. Сообщение, передаваемое с помощью функции SendMessage, практически немедленно обрабатывается оконной процедурой, т. к. последняя получает сообщение не из очереди, а напрямую. При этом приложение дожидается окончания обработки посланного сообщения.

Примечание

Здесь требуются некоторые пояснения. До этого момента мы говорили только об очереди сообщений — "общение" окон между собой и с системой осуществляется через специальную очередь. И вдруг перешли к описанию функции, которая позволяет обойти этот механизм. Пусть вас это не пугает — Windows многоплановая система, и на данном этапе знакомства с ней хотелось бы дать общее представление о ее возможностях.

Конечно же, Windows предоставляет приложениям и возможность послать сообщение некоторому объекту, используя очередь сообщений. Осуществляется это посредством функции PostMessage. В этом случае приложение продолжает выполняться, не дожидаясь реакции на сообщение.

Интерфейс прикладного программирования

Интерфейс прикладного программирования (Application Programming Interface, API) — это набор необходимых функций, при помощи которых любое приложение может взаимодействовать с операционной системой.

За последние 15 лет только в рамках Windows появилось три таких API (фактически их больше). Самый первый — Win16 — интерфейс 16-разрядных операционных систем Microsoft. Попытка увеличить разрядность существующей версии привела к появлению Win32s — 32-разрядного интерфейса для 16-разрядных систем (!). Естественно, с появлением 32-разрядных систем Windows появился и 32-разрядный интерфейс — Win32. Самое интересное то, что API обеих систем представляют собой пересекающиеся подмножества "полного" API, т. е. в одном API есть функции, не реализованные в другом, и наоборот. К сожалению, об этом можно и не узнать, т. к. нереализованные функции представлены в виде "заглушек".

Примечание

Большинство функций возвращают код завершения, а интересующие значения передаются через указатели, заданные параметрами. Функция, реализованная в виде "заглушки", всегда возвращает код, соответствующий ошибке (NULL, FALSE, ..FAIL.. и т.д.).

Итак, API — это связующее звено между приложениями и операционной системой. Масштаб Win32 API — впечатляет, он содержит около двух тысяч функций и несколько сотен сообщений, макросов и предопределенных констант. С одной стороны, программист избавлен от необходимости создавать части операционной системы в своей программе, как это обязательно происходит при создании приложений MS-DOS, с другой стороны, широкие возможности серьезно усложняют процесс обучения, какие-то возможности, предоставляемые системой, не будут использованы. Вспомним, что из десятков тысяч слов русского языка средний человек использует не более пяти тысяч. И в программировании примерно то же самое. Если добавить к функциям API еще немалое количество "обычных" функций, наличие которых является стандартным для компиляторов языка C и C++, то картина становится еще сложнее. Кстати говоря, в отличие от последних, функции API во многом взаимозависимы, т. е. одну функцию невозможно использовать без другой, или параметры некоторых функций должны быть подготовлены другими функциями.

Win32 API, как и его предшественники, в основном состоит из трех компонентов: Kernel, User и GDI, которые обеспечивают интерфейс с базовой операционной системой, управление окнами и приложениями, а также поддержку графики. Таких функций порядка тысячи, значит, новые возможности обеспечиваются функциями, количество которых сравнимо с количеством всех функций API 16-разрядной Windows.

В Microsoft, видимо, очень популярны всякого рода макросы. Действительно, язык программирования C не поддерживает проверку типов передаваемых параметров, поэтому использование "функциональных" макросов не увеличивает общее количество ошибок на программу. Конечно, усложняется

отладка, но это уже проблема программистов. Наверное, не стоило бы об этом говорить, но дело в том, что реально многие, особенно новые, возможности, предоставляемые API, являются макросами. Не будем голословными. Вот фрагмент из оригинала — файла COMMCTRL.H:

```
#define ListView_Arrange(hwndLV, code) \
  (BOOL)SendMessage((hwndLV), LVM_ARRANGE, \
                    (WPARAM)(UINT)(code), 0L)
```

В документации сказано, что можно послать сообщение как с помощью этого макроса, так и напрямую, используя функцию SendMessage. Может быть, действительно, это и удобно, но надо во всяком случае представлять, где новые возможности, а где нет.

Управление графическим выводом

Система Windows создавалась прежде всего как графическая оболочка. Соответственно главное отличие (по крайней мере на первый взгляд) от MS-DOS — это графическое представление информации. Естественно, что практически все приложения используют экран, чтобы отобразить данные, которыми они управляют. Даже если это не так, то все равно визуальные элементы приложений отображаются на экране самой операционной системой. Как сказано выше, Windows обеспечивает универсальность представления информации как на экране, так и на других устройствах вывода, например, на принтере. Причем для этого используются одни и те же примитивы отображения. Система сама распознает целевое устройство и активизирует соответствующий ему модуль. В связи с тем что Windows — многозадачная система, к приложениям предъявляется ряд требований, исключающих конфликты при использовании функций вывода. Однако это не значит, что Windows обеспечивает приложения только набором функций вывода на экран или печать — система управляет всем выводом.

Правильнее будет сказать, что приложения используют в качестве первичного устройства вывода скорее окно, чем непосредственно экран. Каждое устройство вывода в Windows имеет набор текущих параметров, с использованием которых и происходит собственно вывод. Причем в каждый конкретный момент времени только одному приложению соответствует некоторое устройство вывода, что исключает одновременный доступ к последнему, изменение параметров вывода одним приложением перед началом процесса вывода другим.

Графические объекты

Windows обеспечивает приложения набором графических объектов, которые позволяют управлять выводом. Для этого применяются:

- Битовые массивы (bitmaps) — прямоугольные массивы точек, формирующие растровые изображения
- Карандаши (pens) — для задания параметров рисования линий, таких как толщина, цвет и стиль (сплошная, прерывистая и т. п.)
- Кисти (brushes) — для задания параметров заливки замкнутых контуров, таких как цвет и стиль
- Шрифты (fonts) — для задания параметров вывода текста, включая имя шрифта, размер символов и т. п.;
- Регионы (regions) — области окна, которые могут быть ограничены прямоугольником, многоугольником, эллипсом или их комбинацией для выполнения операции заполнения, заливки, инверсии и т. п. Кроме того, служат для определения местоположения курсора
- Логические палитры (logical palettes) — осуществляют интерфейс между приложением и цветным устройством вывода, таким как дисплей, содержат список цветов, необходимых приложению
- Контуры (paths) — используются для целей заполнения или выделения контура различных фигур

Аппаратно-независимый графический вывод

Одна из главных особенностей Win32 API — независимость графического вывода от устройства. Программное обеспечение, которое поддерживает независимость, содержится в двух динамически компонуемых библиотеках (DLL). Первая, GDI.DLL, обеспечивает графический интерфейс устройства (Graphics Device Interface, GDI), вторая является драйвером устройства, например, если приложение осуществляет вывод в область пользовательского окна на дисплее VGA, это библиотека — VGA.DLL, если на принтер Epson FX-80, это библиотека — EPSON9.DLL и т. п.

Перед операцией вывода на некоторое устройство приложение должно запросить GDI о загрузке соответствующего драйвера (обычно это происходит автоматически и не требует от программиста дополнительных усилий). Как только драйвер загружен, приложение может настроить ряд параметров вывода, таких как цвет линии и ее ширина, тип кисти и ее цвет, шрифт, область отсечения и т. д. Windows обеспечивает хранение этих и других параметров в специальной структуре, называемой *контекстом устройства*.

Контекст устройства

Контекст устройства — структура, определяющая набор графических объектов и связанных с ними атрибутов и графических режимов, которые, собственно, и воздействуют на вывод. Графические объекты включают карандаши для рисования линий, кисти для закрашивания и заполнения, битовые обра-

зы для копирования или прокрутки частей экрана, цветовые палитры для определения набора доступных цветов, области для отсечения и других операций, а также контуры для операций рисования и закрашивания.

Приложение не имеет прямого доступа к контексту устройства, как к большинству структур Win32 API, — настройка параметров осуществляется посредством вызова соответствующих функций Win32 API.

Win32 API определяет четыре типа контекстов устройств:

- Для экрана — обеспечивает операции рисования непосредственно на экране
- Для принтера — обеспечивает операции рисования на принтере или плоттере
- Для объектов в памяти — обеспечивает операции рисования без непосредственного вывода на экран
- Информационный — обеспечивает получение данных об устройстве

Используя контекст устройства, приложение может осуществлять следующий набор операций:

- Перечисление графических объектов
- Установка новых графических объектов
- Удаление графических объектов
- Сохранение графических объектов, их атрибутов и параметров графических режимов
- Восстановление графических объектов, их атрибутов и параметров графических режимов

Кроме того, приложение может использовать контекст устройства для определения процесса графического вывода, прерывания длительных графических операций, начатых другим потоком многопоточного приложения, а также инициализировать принтер.

Экран

Win32 API обеспечивает для экрана три типа контекста устройства: *контекст класса*, *общий* и *частный контексты*. Контекст класса и частные контексты устройства используются в приложениях, которые выполняют многочисленные операции рисования. Например, программы автоматизированного проектирования, настольные издательские системы, т. е. такие приложения, которые самостоятельно и постоянно осуществляют вывод и, соответственно, время, затрачиваемое на эти операции, критично с точки зрения производительности приложения. Общие контексты устройства используются в приложениях, которые лишь время от времени выполняют операции рисования.

Приложение получает контекст устройства экрана, вызывая функции BeginPaint или GetDC и идентифицируя окно, в которое соответствующий вывод будет производиться. Причем тип контекста устройства зависит от того, как приложение зарегистрировало класс окна. Обычно приложение получает контекст устройства экрана непосредственно перед выполнением операций вывода. После завершения вывода приложение должно обязательно освободить контекст устройства вызовом функций EndPaint или ReleaseDC соответственно.

- *Контекст класса* поддерживается только для совместимости с предыдущими версиями Windows. При создании Win32-приложения вместо контекстов класса следует использовать частные контексты.
- *Общий контекст устройства* — контекст устройства экрана, связанный с компонентом управления окна Win32 API находящимся в области администратора окна. Приложение получает общий контекст устройства, вызывая функцию GetDC, GetDCEx или BeginPaint. Windows инициализирует общие контексты устройства значениями по умолчанию, которые можно менять по мере необходимости при помощи специальных функций. Количество общих контекстов в системе ограниченно и, следовательно, после завершения операции вывода необходимо сообщить системе об этом, вызвав функции ReleaseDC или EndPaint. В результате вызова одной из этих функций те изменения параметров контекста, которые были произведены приложением, будут отменены. Таким образом, необходимо устанавливать параметры каждый раз.
- *Частные контексты устройства* отличаются от общих тем, что сохраняют любые изменения для заданных по умолчанию данных даже после вызова функции ReleaseDC или EndPaint. Частные контексты устройства не являются частью области администратора окна и, следовательно, лишь однократно инициализируются значениями по умолчанию. Администратор окна автоматически удаляет частный контекст устройства только после того, как последнее окно класса было разрушено.

  Приложение создает частный контекст устройства, определяя стиль CS_OWNDC для класса окна в момент инициализации структуры WNDCLASS с последующей регистрацией класса окна (функция RegisterClass). Для получения частных контекстов окон, имеющих стиль CS_OWNDC, используются те же функции: GetDC, GetDCEx или BeginPaint.

Принтер

Win32 API обеспечивает один и тот же тип контекста устройства как для принтера, который может быть матричным, струйным или лазерным, так и для графопостроителя. Приложение создает контекст устройства принтера, вызывая функцию CreateDC. При этом задаются необходимые параметры,

такие как имя драйвера принтера, имя принтера, файла или другого устройства вывода. После завершения печати приложение должно вызвать функцию DeleteDC для удаления созданного контекста, простое освобождение контекста при помощи функции ReleaseDC недостаточно и недопустимо!

Объект в памяти

Для улучшения характеристик вывода изображений, например для исключения мигания, Win32 API предоставляет возможность приложениям полностью подготавливать изображение в памяти до его вывода на экран дисплея и только затем осуществлять вывод. Для этого предназначен *контекст устройства памяти*, который сохраняет растровые изображения для специфического устройства. Приложение создает контекст устройства памяти, вызывая функцию CreateCompatibleDC. Результатом выполнения этой функции является растровое изображение (растр), имеющее цветной формат, совместимый с форматом первоначального устройства (отсюда и название функции).

Первоначальное изображение в контексте устройства памяти имеет размер один на один пиксел. Прежде чем приложение сможет начать работать с изображением, оно должно установить битовый массив с соответствующей шириной и высотой в контекст устройства, вызывая функцию SelectObject.

Информационный контекст

Win32 API поддерживает информационный контекст устройства, используемый, чтобы восстановить или получить заданные по умолчанию параметры устройства. Для создания информационного контекста приложение должно вызвать функцию CreateIC. Для получения информации об объектах, заданных по умолчанию для интересующего устройства, используются функции GetCurrentObject и GetObject. Использование информационного контекста устройства более эффективно, чем контекстов других типов, т. к. Win32 API работает с информационным контекстом на более низком уровне и не создает структуры, необходимые для их работы.

После завершения работы с информационным контекстом необходимо вызвать функцию DeleteDC для удаления созданного контекста.

Графические режимы

Windows поддерживает пять различных графических режимов, которые позволяют приложениям определять тип смешивания цветов, место и параметры вывода и т. д.

Режим настройки фона. Определяет, как происходит смешивание цветов фона текстовых объектов и растровых изображений и фона поля вывода (окна или экрана).

Режим отображения (рисования). Определяет, как происходит смешивание цвета карандашей, кистей, текстовых объектов и растровых изображений с цветом фона поля вывода (окна или экрана).

Режим масштабирования. Определяет преобразования логических координат при графическом выводе в окна, на экран или принтер.

Режим заливки контуров. Определяет использование шаблонов кистей при заливке сложных контуров.

Режим сжатия. Определяет, каким образом происходит преобразование цветов растровых изображений при их увеличении (уменьшении).

Как и в случае графических объектов, контекст устройства инициализируется значениями по умолчанию и для графических режимов.

Работа со шрифтами

Операционная система Windows может работать со шрифтами следующих трех типов:

- Растровые
- Векторные
- Шрифты TrueType

Отличие между типами шрифтов заключается в способе хранения параметров начертания символов в специальных шрифтовых файлах. В случае растровых шрифтов каждый символ хранится в виде растра (битового массива). Векторные шрифты хранят для каждого символа относительные координаты концов отрезков, из которых состоит соответствующий символ. Шрифты TrueType содержат информацию о линиях и командах изгиба, а также настроечную информацию (hints) для точного отображения символа, которая также используется при уменьшении или увеличении масштаба отображения.

Растровые шрифты имеют заданные размеры и, следовательно, качество их отображения зависит от устройства вывода. Напротив, векторные шрифты не зависят от устройства вывода, однако время, необходимое для их отображения, больше, чем у растровых или шрифтов TrueType. Последние обеспечивают приемлемую скорость вывода и могут быть промасштабированы с сохранением изначального вида символов.

Windows предоставляет разработчикам широкий набор функций для использования шрифтового оформления своих приложений, начиная с того, что каждый контекст устройства имеет шрифт по умолчанию, и заканчивая предоставлением системного (встроенного) диалога для выбора шрифтов, который можно использовать в приложении для предоставления пользователям возможности выбора любого шрифта из имеющихся.

В заключение главы приведем пример приложения — надо же с чего-то начинать!

APIShell — пример приложения

Следующий пример иллюстрирует целый класс приложений, основным окном которых является блок диалога (dialog box, окно диалога, диалог). Кроме того, в примере рассмотрена одна из возможностей управления графической оболочкой Windows при помощи функции Shell_NotifyIcon.

```
/*************************************************************
Файл APIShell.c
Copyright (c) 1999 Мешков А., Тихомиров Ю.
*************************************************************/

#define _INC_OLE
#include <windows.h>
#undef  _INC_OLE

#include <windowsx.h>

#include "resource.h"

/* определение номера сообщения */
#define WM_NOTIFYICON    (WM_APP+100)

HINSTANCE g_hinst;
UINT g_uState;

/* идентификатор значка */
#define IDC_NOTIFY 1000

/*
   Объявления используемых функций
*/

/* Оконная функция окна диалога */
BOOL CALLBACK TestAppDlgProc(HWND hDlg, UINT uMsg,
                             WPARAM wParam, LPARAM lParam);
/* Функция настройки уведомляющей области */
BOOL TrayMessage(HWND hDlg, DWORD dwMessage,
                 HICON hIcon, PSTR pszTip);
/* Функция изменения состояния значка уведомления */
```

```
void StateChange(HWND hDlg);

/*
   Реализация функций приложения
*/
int WINAPI WinMain(
  HINSTANCE hInstance,
  HINSTANCE hPrevInstance,
  LPSTR lpCmdLine,
  int nCmdShow)
{
/* Сохраняем дескриптор приложения, не проверяя
   наличие предыдущих копий */
  g_hinst = hInstance;

/* Создаем и выводим диалог настройки -
   основное окно приложения */
  DialogBox(
    hInstance,
    MAKEINTRESOURCE(IDD_APISHELL),
    NULL,
    TestAppDlgProc);

/* Основное окно закрыто - завершаем приложение */
  return(FALSE);
}

/* Функция настройки уведомляющей области */
BOOL CALLBACK TestAppDlgProc(HWND hDlg,
                             UINT uMsg,
                             WPARAM wParam,
                             LPARAM lParam)
{
  switch (uMsg)
  {
    case WM_INITDIALOG:
      /* Инициализируем окно диалога */
      TrayMessage(hDlg, NIM_ADD, NULL, NULL);
      StateChange(hDlg);
      break;
```

```
case WM_DESTROY:
  /* Разрушаем окно диалога */
  TrayMessage(hDlg, NIM_DELETE, NULL, NULL);
  break;

case WM_COMMAND:
  /* Обрабатываем сообщения от элементов окна диалога */
  switch (GET_WM_COMMAND_ID(wParam, lParam))
  {
    case IDCANCEL:
      EndDialog(hDlg, TRUE);
      break;

    case IDHIDE:
      ShowWindow(hDlg, SW_HIDE);
      break;
  }
  break;

case WM_NOTIFYICON:
  /* Обрабатываем сообщения от значка
     в области уведомления */
  switch (lParam)
  {
    case WM_LBUTTONDOWN:
      switch (wParam)
      {
        case IDC_NOTIFY:
          /* Изменяем состояние и значок */
          g_uState = !g_uState;
          StateChange(hDlg);
          break;

        default:
          break;
      }
      break;

    case WM_RBUTTONDOWN:
```

```
          /* Активизируем окно диалога */
          ShowWindow(hDlg, SW_SHOW);
          SetForegroundWindow(hDlg);
          break;

        default:
          break;
      }
      break;

    default:
      return(FALSE);
  }

  return(TRUE);
}

BOOL TrayMessage(HWND hDlg,
                 DWORD dwMessage,
                 HICON hIcon,
                 PSTR pszTip)
{
  BOOL retCode;
  NOTIFYICONDATA tnd;

  /* Формируем структуру для работы с областью уведомления */
  tnd.cbSize = sizeof(NOTIFYICONDATA);
  tnd.hWnd = hDlg;
  tnd.uID = IDC_NOTIFY;

  tnd.uFlags = NIF_MESSAGE|NIF_ICON|NIF_TIP;
  tnd.uCallbackMessage = WM_NOTIFYICON;
  tnd.hIcon = hIcon;

  if (pszTip)
    lstrcpyn(tnd.szTip, pszTip, sizeof(tnd.szTip));
  else
    tnd.szTip[0] = '\0';
```

```
  /* Настраиваем область уведомления */
  retCode = Shell_NotifyIcon(dwMessage, &tnd);

  if (hIcon)
    DestroyIcon(hIcon);

  return retCode;
}

void StateChange(HWND hDlg)
{
  LPCSTR pszIDIcon;
  char szTip[64];

  /* В зависимости от текущего состояние загружаем
     нужный текст и значок */
  if (!LoadString(g_hinst, g_uState == 0 ?
      IDC_STATE1 : IDC_STATE2, szTip, sizeof(szTip)))
    *szTip = '\0';

  pszIDIcon = MAKEINTRESOURCE(
    g_uState ? IDI_STATE1 : IDI_STATE2);

  TrayMessage(hDlg, NIM_MODIFY,
    LoadImage(g_hinst, pszIDIcon, IMAGE_ICON, 16, 16, 0),
    szTip);
}

/************************************************************
Файл APIShell.rc
Copyright (c) 1999 Мешков А., Тихомиров Ю.
************************************************************/
//Microsoft Developer Studio generated resource script.
//
#include "resource.h"
#define APSTUDIO_READONLY_SYMBOLS
/////////////////////////////////////////////////////////////
//
```

```
// Generated from the TEXTINCLUDE 2 resource.
//
#include <windows.h>
/////////////////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS
/////////////////////////////////////////////////////////////////////////////
// Russian resources
#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_RUS)
#ifdef _WIN32
LANGUAGE LANG_RUSSIAN, SUBLANG_DEFAULT
#pragma code_page(1251)
#endif //_WIN32
#ifdef APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
//
1 TEXTINCLUDE DISCARDABLE
BEGIN
  "resource.h\0"
END
2 TEXTINCLUDE DISCARDABLE
BEGIN
  "#include ""windows.h""\0"
END
#endif  // APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////////////////////
//
// Icon
//
// Icon with lowest ID value placed first to ensure
// application icon remains consistent on all systems.
IDI_STATE1       ICON  DISCARDABLE   "STATE1.ICO"
IDI_STATE2       ICON  DISCARDABLE   "STATE2.ICO"
/////////////////////////////////////////////////////////////////////////////
//
// Dialog
//
IDD_APISHELL DIALOG DISCARDABLE 0, 0, 89, 26
```

```
STYLE DS_MODALFRAME | DS_CENTER | WS_MINIMIZEBOX |
  WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "APIShell"
FONT 8, "MS Sans Serif"
BEGIN
  PUSHBUTTON   "&Спрятать",IDHIDE,19,6,50,14
END
/////////////////////////////////////////////////////////////
//
// String Table
//
STRINGTABLE DISCARDABLE
BEGIN
  IDC_STATE1       "Принтер занят"
  IDC_STATE2       "Принтер свободен"
END
#endif  // Russian resources
/*************************************************************
Файл resource.h
Copyright (c) 1999 Мешков А., Тихомиров Ю.
*************************************************************/
/////////////////////////////////////////////////////////////
//{{NO_DEPENDENCIES}}
// Microsoft Developer Studio generated include file.
// Used by APIShell.rc
//
#define IDD_APISHELL         103
#define IDI_STATE1           105
#define IDI_STATE2           106
#define IDC_STATE1           1009
#define IDC_STATE2           1010
#define IDHIDE               1019
// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE    111
#define _APS_NEXT_COMMAND_VALUE     106
#define _APS_NEXT_CONTROL_VALUE     1020
```

```
#define _APS_NEXT_SYMED_VALUE        101
#endif
#endif
```

Первое, с чего начинается код приложения, — включение необходимых для работы заголовочных файлов. Обратите внимание на макроопределение _INC_OLE. Таким нехитрым способом из процесса компиляции исключается большое количество строк кода, которые для данной программы абсолютно не нужны.

```
#define _INC_OLE
#include <windows.h>
#undef _INC_OLE
#include <windowsx.h>
```

Примечание

При создании приложений с использованием библиотеки классов MFC применяется аналогичный, но более корректный метод исключения неиспользуемых заголовочных файлов.

Рис. 3.7. Окно управления приложением APIShell

Рис. 3.8. Область уведомления до запуска приложения APIShell

Рис. 3.9. Область уведомления со значком принтера

Рис. 3.10. Область уведомления с измененным значком принтера

Создать диалоговое окно, как видите, очень просто: передаем идентификатор ресурса и указатель на функцию блока диалога. После завершения диалога выполнять больше нечего — завершаем функцию WinMain.

```
DialogBox(
  hInstance,
  MAKEINTRESOURCE(IDD_APISHELL),
  NULL,
  TestAppDlgProc);
```

В функции TestAppDlgProc, которая является функцией окна, необходимо явно обработать следующие сообщения:

- WM_INITDIALOG — в программе собственно блок диалога инициализируется по умолчанию на основе того, что указано в шаблоне диалога. Однако необходимо добавить в область уведомления значок принтера, передав системе также информацию об обработчике сообщений от той части области уведомления, где располагается значок.
- WM_DESTROY — разрушение или завершение диалога происходит, как и в случае инициализации, без нашего вмешательства; необходимо только удалить значок из области уведомления.
- WM_COMMAND — сообщения от элементов управления диалога (IDHIDE и IDCANCEL), которые позволяют пользователю, соответственно, скрывать блок диалога и завершать приложение.
- WM_NOTIFYICON — сообщение от значка в области уведомления, анализ которого позволяет менять значки и активизировать блок диалога в зависимости от кнопки мыши, которую нажал пользователь, указав значок.

Теперь пришло время поближе познакомиться с основными компонентами и общей структурой приложения для Windows.

Глава 4

Структура приложений для Windows

Любое приложение Windows, имеющее интерфейс с пользователем (за исключением консольных приложений), состоит из двух основных частей: функции WinMain и так называемой *функции окна*. Рассмотрим задачи, которые должна решать функция WinMain приложения, поддерживающего однодокументный (SDI) или многодокументный (MDI) интерфейс. Вот стандартная, хотя и немного усеченная реализация такой функции:

```
int WINAPI WinMain(
  HINSTANCE hInstance,
  HINSTANCE hPrevInstance,
  LPSTR lpCmdLine,
  int nCmdShow)
{
  MSG msg;
  // Хотя параметр hPrevInstance в Win32 всегда равен NULL,
  // продолжаем проверять его значение для совместимости
  if (!hPrevInstance)
  {
    // Инициализируем приложение -
    // подготавливаем данные класса окна и регистрируем его
    if (!InitApplication(hInstance))
      return (FALSE);
  }
  // Завершаем создание копии приложения -
  // создаем главное окно приложения
  if (!InitInstance(hInstance, nCmdShow))
    return (FALSE);
```

```
  // Цикл обработки сообщений
  while (GetMessage(&msg, NULL, 0, 0))
  {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
  }
  return (msg.wParam);
}
```

Функция состоит из трех частей:

- Регистрация класса окна — отвечает функция InitApplication
- Создание главного окна приложения — реализуется в функции InitInstance
- Цикл обработки сообщений — цикл while

Используемые для регистрации класса окна функции InitApplication и InitInstance не системные — они должны быть написаны разработчиком программы. Ниже приведены примеры типичной реализации этих функций:

```
BOOL InitApplication(HINSTANCE hInstance)
{
  WNDCLASS  wc;
  ...
  wc.style         = CS_HREDRAW | CS_VREDRAW;
  wc.lpfnWndProc   = (WNDPROC)WndProc;
  wc.cbClsExtra    = 0;
  wc.cbWndExtra    = 0;
  wc.hInstance     = hInstance;
  wc.hIcon         = LoadIcon(hInstance, IDI_APLICATION);
  wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
  wc.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
  wc.lpszMenuName  = NULL;
  wc.lpszClassName = szAppName;
  return RegisterClass(&wc);
}
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
  HWND hWnd;
  hWnd = CreateWindow(
    szAppName, szTitle, WS_OVERLAPPEDWINDOW,
```

```
    CW_USEDEFAULT, 0, CW_USEDEFAULT, 0,
    NULL, NULL, hInstance, NULL);
  if (!hWnd)
    return (FALSE);
  ShowWindow(hWnd, nCmdShow);
  UpdateWindow(hWnd);
  return (TRUE);
}
```

Более подробно об этих функциях мы поговорим позднее, а пока рассмотрим "кровеносную систему" любого приложения для Windows.

Стандартный цикл обработки сообщений

Стандартный цикл обработки сообщений имеет следующий вид:

```
while (GetMessage(&msg, NULL, 0, 0))
{
  TranslateMessage(&msg);
  DispatchMessage(&msg);
}
```

Данный цикл работает в течение всего времени выполнения программы. Каждая итерация представляет собой не что иное, как извлечение одного сообщения из очереди сообщений текущего потока. За это отвечает функция GetMessage, второй аргумент которой (NULL) говорит о том, что следует обрабатывать все сообщения. Когда сообщение выбирается из очереди, оно помещается в специальную структуру MSG, которая выглядит следующим образом:

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

hwnd — содержит дескриптор (логический номер) окна, оконная процедура которого получает сообщение; message — определяет номер сообщения; wParam и lParam — определяют

дополнительную информацию о сообщении. Конкретное содержание этих параметров зависит от типа сообщения; time и pt — описывают состояние системы в момент постановки сообщения в очередь: time хранит время, а pt — позицию указателя мыши в этот конкретный момент времени.

После того как сообщение выбрано из очереди сообщений, оно передается в функцию TranslateMessage. Ее основное назначение заключается в вызове драйвера клавиатуры Windows для преобразования виртуальных кодов клавиш в ASCII-значения, которые ставятся в очередь программных событий в виде сообщения WM_CHAR. Это позволяет программе отличить, например, "A" от "a" без анализа состояния клавиши регистра.

Последняя функция цикла — DispatchMessage — берет данные о сообщении из структуры MSG и передает их в соответствующую оконную процедуру для обработки. После того как сообщение передано, снова вызывается функция GetMessage, чтобы взять из очереди следующее сообщение, если таковое имеется. За исключением WM_QUIT, каждое сообщение заставляет GetMessage возвратить значение TRUE. Принимая сообщение WM_QUIT, программа выходит из цикла обработки и завершает свою работу.

Важно понимать, что этот (или подобный ему) цикл без конца повторяется в течение всего срока жизни приложения. Он составляет основу и суть любого приложения Windows.

Теперь пришло время поговорить о "получателе" сообщений.

Оконная процедура

Для того чтобы программа могла взаимодействовать "с внешним миром", необходимо это каким-либо образом обеспечить. В любом приложении Windows роль получателя сообщений выполняет специальная оконная процедура, которой передаются сообщения, поступающие из очереди приложения (говоря точнее — из очереди потока приложения), где они обрабатываются индивидуально:

```
LRESULT CALLBACK WindowProc(
  HWND hwnd,
  UINT message,
  WPARAM wParam,
  LPARAM lParam)
{
  switch(message)
  {
    case WM_DESTROY:
```

```
      PostQuitMessage(0);
      break;
    default:
      return DefWindowProc(hwnd, message, wParam, lParam);
  }
  return 0;
}
```

Как видите, приложение явно отвечает только на сообщение WM_DESTROY, а все остальные передаются в DefWindowProc — специальную функцию, управляющую поведением окна по умолчанию.

Примечание

Следует понимать, что такая структура оконной процедуры относится только к простейшему случаю.

Она "умеет" обрабатывать почти все виды поведения, ассоциированные с конкретным типом окна. Например, она "знает", как обрабатывать действия мыши, изменяющие форму и местоположение окна. Однако DefWindowProc не занимается сообщением WM_DESTROY, и поэтому вы должны сами обеспечить его обработку. Строки

```
case WM_DESTROY:
  PostQuitMessage(0);
  break;
```

представляют собой не что иное, как один из простейших *обработчиков сообщения*, которые широко используются при написании как традиционных С-программ, так и программ, базирующихся на какой-либо библиотеке классов, будь то OWL (библиотека классов компилятора Borland C++) или MFC.

Примечание

Обработчиком сообщения называют специальную функцию, которая отвечает за обработку одного-единственного сообщения.

Теперь, когда вы познакомились с главной артерией, поддерживающей жизнь приложения Windows, имеет смысл более подробно остановиться на типах сообщений, действующих в системе Windows.

Основные типы оконных сообщений Windows

В настоящее время в системе Windows имеется более 900 стандартных сообщений. К счастью, нет необходимости знакомиться со всеми, поскольку большинство из них используются очень редко. Например, большое количе-

ство сообщений характерно для одного типа окна, другие сообщения играют особую роль в совместном использовании данных или при реализации стандарта многодокументного интерфейса (MDI). Третьи проходят по нашему циклу сообщений, но никогда не используются. Это внутренние сообщения, которые Windows создает для своих собственных целей.

Все сообщения имеют имена, которые определены в WINUSER.H следующим образом:

```
#define WM_COMMAND 0x0111
```

или

```
#define WM_PAINT   0x000F
```

Префикс WM_ говорит о том, что речь идет об "оконном сообщении" (Window Message). Эти символические имена обозначают числа. Кроме WM_ широко используются и другие префиксы: EM_ (Edit Message) — для сообщений элемента управления EDIT, BM_ (Button Message) — для сообщений элемента управления BUTTON, LM_ (Listbox Message) — для сообщений элемента управления LIST BOX и т. д. Помимо этого, вы можете создавать и собственные сообщения.

APIWin — пример приложения

Приложение, исходные тексты которого приведены ниже, является простейшим текстовым редактором. Это SDI-приложение, т. е. приложение обеспечивает работу только с одним документом. Мы приводим текст только одного файла APIWIN.C. Файл описания идентификаторов RESOURCE.H содержит идентификаторы команд меню, а файл ресурсов — ссылки на значки, шаблон меню и таблицу акселераторов. При желании все это можно написать самостоятельно, основываясь, прежде всего, на содержимом функции окна приложения WndProc.

При ознакомлении с текстом обратите внимание на отличие операции регистрации окна в зависимости от системы, в которой создается исполняемый модуль приложения, и от системы, в рамках которой выполняется приложение.

```
/************************************************************
Модуль APIWin.c
Copyright (c) 1999 Мешков А., Тихомиров Ю.
************************************************************/
/* имя приложения */
#define APPNAME "APIWIN"
#include <windows.h>          /* Описание функций Win32 API */
#include "resource.h"         /* Определение идентификаторов */
```

```
/* Макрос для определения версии системы */
#if defined (WIN32)
  #define IS_WIN32 TRUE
#else
  #define IS_WIN32 FALSE
#endif
#define IS_NT IS_WIN32 && (BOOL)(GetVersion() < 0x80000000)
#define IS_WIN32S IS_WIN32 && (BOOL)(!(IS_NT) && \
                  LOBYTE(LOWORD(GetVersion()))<4)
#define IS_WIN95 (BOOL)(!(IS_NT) && !(IS_WIN32S)) && IS_WIN32
/* Глобальные переменные */
HINSTANCE hInst;                /* Идентификатор приложения */
HWND hwndEdit;              /* Идентификатор текстового поля */
char szAppName[] = APPNAME;               /* Имя приложения */
/* Заголовок главного окна */
char szTitle[] = APPNAME": Приложение-пример";
CHAR szFileName[_MAX_PATH];
CHAR szFileTitle[_MAX_PATH];
/* Определение используемых функций */
ATOM APIWinRegisterClass(CONST WNDCLASS *);
BOOL InitApplication(HINSTANCE);
BOOL InitInstance(HINSTANCE, int);
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
BOOL LoadFile(HANDLE hwnd, BOOL bFileNew);
BOOL SaveFile(HANDLE hwnd, BOOL bSaveAs);
/*
   Главная функция приложения — инициализирует приложение и
   выполняет цикл обработки сообщений
*/
int WINAPI WinMain(
  HINSTANCE hInstance,
  HINSTANCE hPrevInstance,
  LPSTR lpCmdLine,
  int nCmdShow)
{
  MSG msg;
  HANDLE hAccelTable;
  if (!hPrevInstance)
  {
```

```
    /* Инициализируем приложение -
       подготавливаем данные класса окна и регистрируем его
    */
    if (!InitApplication(hInstance))
      return (FALSE);
  }
  // Завершаем создание копии приложения -
  // создаем главное окно приложения
  if (!InitInstance(hInstance, nCmdShow))
    return (FALSE);
  hAccelTable = LoadAccelerators(hInstance, szAppName);
  /* Цикл обработки сообщений */
  while (GetMessage(&msg, NULL, 0, 0))
  {
    if (!TranslateAccelerator (msg.hwnd, hAccelTable, &msg))
    {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
    }
  }
  return (msg.wParam);
}
/*
   Функция регистрации оконного класса
*/
ATOM APIWinRegisterClass(CONST WNDCLASS *lpwc)
{
  HANDLE  hMod;
  FARPROC proc;
  WNDCLASSEX wcex;
  hMod = GetModuleHandle("USER32");
  if (hMod != NULL)
  {
#if defined (UNICODE)
    proc = GetProcAddress(hMod, "RegisterClassExW");
#else
    proc = GetProcAddress(hMod, "RegisterClassExA");
#endif
    if (proc != NULL)
```

```
    {
      wcex.style         = lpwc->style;
      wcex.lpfnWndProc   = lpwc->lpfnWndProc;
      wcex.cbClsExtra    = lpwc->cbClsExtra;
      wcex.cbWndExtra    = lpwc->cbWndExtra;
      wcex.hInstance     = lpwc->hInstance;
      wcex.hIcon         = lpwc->hIcon;
      wcex.hCursor       = lpwc->hCursor;
      wcex.hbrBackground = lpwc->hbrBackground;
      wcex.lpszMenuName  = lpwc->lpszMenuName;
      wcex.lpszClassName = lpwc->lpszClassName;
      /* добавляем элемент, необходимый для Windows 95/98 */
      wcex.cbSize = sizeof(WNDCLASSEX);
      wcex.hIconSm = LoadIcon(wcex.hInstance, "APIWIN_SMALL");
      return (*proc)(&wcex);
    }
  }
  return (RegisterClass(lpwc));
}
/*
   Функция инициализации данных окна и его регистрации
*/
BOOL InitApplication(HINSTANCE hInstance)
{
  WNDCLASS  wc;
  HWND hwnd;
  /* в Win32 API параметр hPrevInstance всегда равен NULL,
     поэтому для нахождения ранее запущенного экземпляра
     приложения используем функцию поиска окна
     (если нужно предотвратить повторный запуск приложения)
 */
  hwnd = FindWindow (szAppName, NULL);
  if (hwnd)
  {
    /* Нашли копию приложения —
       восстанавливаем его на экране и активизируем
    */
    if (IsIconic(hwnd))
      ShowWindow(hwnd, SW_RESTORE);
```

```
    SetForegroundWindow (hwnd);
    /* Данная копия должна быть завершена */
    return (FALSE);
  }
  wc.style         = CS_HREDRAW | CS_VREDRAW;
  wc.lpfnWndProc   = (WNDPROC)WndProc;
  wc.cbClsExtra    = 0;
  wc.cbWndExtra    = 0;
  wc.hInstance     = hInstance;
  wc.hIcon         = LoadIcon(hInstance, szAppName);
  wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
  wc.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
  wc.lpszMenuName  = szAppName;
  wc.lpszClassName = szAppName;
  if (IS_WIN95)
    return APIWinRegisterClass(&wc);
  else
    return RegisterClass(&wc);
}
/*
   Функция создания главного окна
*/
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
  HWND hWnd;
  hInst = hInstance;
  hWnd = CreateWindow(
    szAppName, szTitle, WS_OVERLAPPEDWINDOW,
    CW_USEDEFAULT, 0, CW_USEDEFAULT, 0,
    NULL, NULL, hInstance, NULL);
  if (!hWnd)
    return (FALSE);
  ShowWindow(hWnd, nCmdShow);
  UpdateWindow(hWnd);
  return (TRUE);
}
/*
   Функция главного окна
   обрабатывает следующие сообщения:
```

```
     WM_CREATE   — создание окна
     WM_SIZE     — изменение размеров окна
     WM_COMMAND  — обработка команд пользователя
     WM_DESTROY  — команда на завершение приложения
*/
LRESULT CALLBACK WndProc(
  HWND hWnd,
  UINT message,
  WPARAM wParam,
  LPARAM lParam)
{
  int wmId, wmEvent;
  RECT rc;
  switch (message)
  {
    case WM_CREATE:
      // Создаем текстовое поле
      hwndEdit = CreateWindow(
        "edit", NULL,
        WS_CHILD|WS_HSCROLL|WS_MAXIMIZE|WS_VISIBLE|WS_VSCROLL|
        ES_AUTOHSCROLL|ES_AUTOVSCROLL|ES_MULTILINE,
        0, 0, 0, 0,
        hWnd, GetMenu(hWnd),
        hInst, NULL);
      SetFocus (hwndEdit);
      break;
    case WM_SIZE:
       /* При изменении размеров главного окна
          изменяем и текстовое поле
       */
       GetClientRect(hWnd, &rc);
       MoveWindow(
         hwndEdit,
         rc.left, rc.top,
         rc.right-rc.left,
         rc.bottom-rc.top, TRUE);
       break;
    case WM_COMMAND:
      wmId = LOWORD(wParam);
```

```
      wmEvent = HIWORD(wParam);
      // Анализ команды меню
      switch (wmId)
      {
        case IDM_EXIT:
          DestroyWindow (hWnd);
          break;
        case IDM_NEW:
          LoadFile(hWnd, TRUE);
          break;
        case IDM_OPEN:
          LoadFile(hWnd, FALSE);
          break;
        case IDM_SAVE:
          SaveFile(hWnd, FALSE);
          break;
        case IDM_SAVEAS:
          SaveFile(hWnd, TRUE);
          break;
        // Команды, которые пока не обрабатываем
        default:
          return (DefWindowProc(
            hWnd, message, wParam, lParam));
      }
      break;
    case WM_DESTROY:
      PostQuitMessage(0);
      break;
    default:
      return (DefWindowProc(hWnd, message, wParam, lParam));
  }
  return (0);
}
// Функция для загрузки содержимого файла или очистки поля
BOOL LoadFile(HANDLE hwnd, BOOL bFileNew)
{
  OPENFILENAME ofn;
  DWORD dwFileSize;
  HGLOBAL hBuffer;
```

```
LPSTR lpBuffer;
HANDLE hFile;
if (bFileNew)
{
  szFileName[0] = '\0';
  SendMessage(hwndEdit, WM_SETTEXT, 0, (LPARAM)");
}
else
{
  memset(&ofn, 0, sizeof(ofn));
  ofn.lStructSize       = sizeof(OPENFILENAME);
  ofn.hwndOwner         = GetActiveWindow();
  ofn.lpstrFilter       = "Все файлы(*.*)\0*.*\0";
  ofn.lpstrCustomFilter = NULL;
  ofn.nMaxCustFilter    = 0;
  ofn.nFilterIndex      = 0;
  ofn.lpstrFile         = szFileName;
  ofn.nMaxFile          = _MAX_PATH;
  ofn.lpstrInitialDir   = NULL;
  ofn.lpstrFileTitle    = szFileTitle;
  ofn.nMaxFileTitle     = _MAX_PATH;
  ofn.lpstrTitle        = "Открытие файла";
  ofn.lpstrDefExt       = NULL;
  ofn.Flags             = OFN_FILEMUSTEXIST |
                          OFN_HIDEREADONLY;
  // Используем стандартный диалог
  GetOpenFileName ((LPOPENFILENAME)&ofn);

  /* Открываем файл по имени */
  hFile = CreateFile(szFileName, GENERIC_READ,
    0, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

  /* Определяем длину файла */
  dwFileSize = GetFileSize(hFile, NULL);

  /* Создаем буфер для чтения из файла */
  hBuffer = GlobalAlloc(GHND, dwFileSize+1);
  lpBuffer = (LPSTR)GlobalLock(hBuffer);
```

```
    /* Читаем содержимое файла */
    ReadFile(hFile, lpBuffer, dwFileSize, &dwFileSize, NULL);
    lpBuffer[dwFileSize] = '\0';

    /* Заполняем текстовое поле */
    SendMessage(hwndEdit, WM_SETTEXT, 0, (LPARAM)lpBuffer);

    /* Освобождаем буфер */
    GlobalUnlock(lpBuffer);
    GlobalFree(hBuffer);

    /* Закрываем файл */
    CloseHandle(hFile);
  }

  return TRUE;
}

// Функция для сохранения содержимого поля в файле
BOOL SaveFile(HANDLE hwnd, BOOL bSaveAs)
{
  OPENFILENAME ofn;
  DWORD dwFileSize;
  HGLOBAL hBuffer;
  LPSTR lpBuffer;
  HANDLE hFile;

  if (bSaveAs)
  {
    memset(&ofn, 0, sizeof(ofn));

    ofn.lStructSize       = sizeof(OPENFILENAME);
    ofn.hwndOwner         = hwnd;
    ofn.lpstrFilter       = "Все файлы(*.*)\0*.*\0";
    ofn.lpstrCustomFilter = NULL;
    ofn.nMaxCustFilter    = 0;
    ofn.nFilterIndex      = 0;
    ofn.lpstrFile         = szFileName;
    ofn.nMaxFile          = _MAX_PATH;
```

```
    ofn.lpstrInitialDir   = NULL;
    ofn.lpstrFileTitle    = szFileTitle;
    ofn.nMaxFileTitle     = _MAX_PATH;
    ofn.Flags             = OFN_OVERWRITEPROMPT |
                            OFN_HIDEREADONLY;

    /* Используем стандартный диалог */
    GetSaveFileName((LPOPENFILENAME)&ofn);
  }

  // По имени открываем или создаем файл, если он не существует
  hFile = CreateFile(szFileName, GENERIC_WRITE,
    0, NULL, OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);

  /* Удаляем содержимое файла */
  SetEndOfFile(hFile);

  /* Определяем размер текста в поле */
  dwFileSize = SendMessage(hwndEdit, WM_GETTEXTLENGTH, 0, 0);

  /* Создаем буфер для чтения из буфера поля */
  hBuffer = GlobalAlloc(GHND, dwFileSize+1);
  lpBuffer = (LPSTR)GlobalLock(hBuffer);

  /* Читаем содержимое буфера */
  SendMessage(hwndEdit, WM_GETTEXT,
    dwFileSize+1, (LPARAM)lpBuffer);

  /* Записываем буфер в файл */
  WriteFile(hFile, lpBuffer, dwFileSize, &dwFileSize, NULL);

  /* Освобождаем буфер */
  GlobalUnlock(lpBuffer);
  GlobalFree(hBuffer);

  /* Закрываем файл */
  CloseHandle(hFile);
  return TRUE;
}
```

Как видите, все, что касается собственно оконной функции нашего текстового поля, ненамного сложнее кода, приведенного ранее. Наибольший интерес представляют функции, активизирующие некоторые стандартные диалоги и функции работы с файлами. Рассмотрим их поподробнее.

Забегая немного вперед, скажем, что практически все стандартные диалоги поддерживаются библиотекой классов MFC. Настройка таких диалогов осуществляется примерно одинаково, как при использовании функций Win32 API, так и при использовании соответствующих классов библиотеки. Функции GetOpenFileName и GetSaveFileName очень показательны в этом смысле. И та и другая управляются при помощи подготавливаемой структуры OPENFILENAME. В части VII подробно рассказано о поддержке библиотекой классов MFC стандартных диалогов, где в числе прочих рассмотрена и эта структура.

Что касается функций ввода/вывода (в примере используются функции CreateFile, GetFileSize, SetEndOfFile, ReadFile, WriteFile и универсальная функция освобождения/закрытия объекта CloseHandle), то их в ряде случаев приходится использовать и при программировании MFC-приложений. Важно отметить тот факт, что файловый ввод/вывод в рамках Win32 API коренным образом отличается от Win16 API. (Например, для открытия файла вызывается функция CreateFile с флагом OPEN_EXISTING, а не функция OpenFile). Win32 API унифицирует работу с потоками данных различной, если хотите, физической природы. Это могут быть файлы, каналы, порты, стандартные устройства ввода/вывода и т. д. Кроме того, использование именно этого интерфейса задействует новые, более эффективные средства обмена с дисковыми накопителями, сетями и т. п.

Примечание

Очень хорошо о файловом вводе/выводе, включая асинхронный, рассказано в книге Джеффри Рихтера "Windows для профессионалов".

На этом мы заканчиваем знакомство с основными чертами программирования для Windows и переходим к основной теме книги.

БИБЛИОТЕКА КЛАССОВ MICROSOFT FOUNDATION CLASSES (MFC)

Глава 5

Обзор библиотеки MFC

Для выполнения какой-либо работы на профессиональном уровне недостаточно поверхностного знакомства с тем инструментарием, которым собираешься пользоваться. Необходимо ясно представлять себе внутреннюю структуру и логику работы используемого средства. В полной мере это относится к инструменту построения (создания) программных продуктов. Мы подходим к библиотеке классов MFC именно как к инструменту, который, взяв на себя очень большую часть черновой работы, требует не простого знакомства с ним, а глубокого изучения. Только знание всех нюансов построения и возможностей этой библиотеки позволит быстро и легко создавать программы любой степени сложности. В противном случае нет уверенности, что созданное приложение будет при любых условиях работать правильно, — слишком много скрыто непосредственно от программиста, функции вызываются не только из недр Windows, но и из недр библиотеки. Ни о каком последовательном вызове функций речь не идет. Данные готовятся неизвестно где и кем, и неизвестно, когда обрабатываются. Какая уж тут уверенность!

Что же представляет собой библиотека Microsoft Foundation Classes (MFC)? Это множество классов, охватывающее большую часть функциональных возможностей Windows и предоставляющее разработчику не только комплекс очень мощных дополнительных классов, но и целые механизмы, которые, по сути, не нарушая идеологию операционной системы, существенно ее расширяют и... упрощают.

Перед создателями библиотеки стояла задача разработки объектно-ориентированного интерфейса для работы в среде Windows, который удовлетворял бы следующим основным целям проектирования программных продуктов:

- Значительное уменьшение усилий при программировании приложений для Windows
- Скорость выполнения программ, написанных с использованием библиотеки, должна быть сопоставима со скоростью выполнения программ, написанных на языке C с использованием Win32 API

- Размер вспомогательного кода должен быть минимальным
- Способность напрямую вызывать любую С-функцию Win32 API
- Легкость использования Win32 API в С++ должна быть такая же, что и при использовании традиционного языка С

Надо сказать, что поставленная задача была решена очень профессионально. Созданная библиотека классов предоставляет окна, блоки диалога, контексты устройств, общие объекты GDI, такие как битовые образы и кисти, элементы управления и многие другие стандартные элементы Windows. Основа программирования для Windows — обработка событий — предоставлена программистам в удобном и привычном виде. Классы библиотеки полностью вобрали в себя многочисленные операторы switch, которые так загромождают программы, написанные на языке С. Наряду с этим вы можете совершенно свободно использовать как вызовы библиотеки классов, так и прямые вызовы Win32 API. Кроме того, за сравнительно небольшой промежуток времени корпорацией Microsoft было разработано несколько версий библиотеки MFC, которая становилась все мощнее и удобнее. Есть все основания считать, что развитие не остановится и на версии MFC 4.3. К тому же поддержка другими компаниями в их компиляторах и средствах разработки программного обеспечения именно библиотеки MFC, а не OWL фирмы Borland International или собственных дублирующих продуктов позволяет сконцентрироваться именно на MFC. Но давайте рассмотрим все по порядку. Начнем с общего обзора библиотеки.

Иерархия классов MFC

На рис. 5.1 показана иерархия основных категорий классов библиотеки MFC. Каждый новый производный класс обладает как свойствами, унаследованными им от родительских классов, так и приобретает новые, характерные только для данного класса. На вершине иерархии находится единственный базовый класс CObject. Все остальные классы библиотеки MFC можно условно разбить на две группы, в зависимости от их отношения к классу CObject: производные и не производные от него.

Для того чтобы составить представление о структуре и возможностях библиотеки классов MFC, приведем краткое описание классов, составляющих эту библиотеку. Начнем с вершины представленной иерархии — класса CObject.

CObject — основа всех классов

Большинство классов в библиотеке MFC являются производными от единственного класса CObject — корня иерархической структуры. Основное назначение этого класса заключается в предоставлении всем производным от него классам следующих возможностей:

- Хранение информации о классе времени выполнения
- Поддержка сериализации и диагностики объекта

Рис. 5.1. Иерархия классов по категориям

К этим понятиям мы еще вернемся, а сначала рассмотрим данные и функции этого класса и тесно связанной с ним структуры CRuntimeClass, определенные в файле AFX.H.

```
class CObject
{
protected:
  CObject();
public:
  virtual ~CObject();
```

```
  void* PASCAL operator new(size_t nSize);
  void* PASCAL operator new(size_t, void* p);
  void  PASCAL operator delete(void* p);
public:
  BOOL IsSerializable() const;
  BOOL IsKindOf(const CRuntimeClass* pClass) const;
// Переопределяемые функции-члены
  virtual CRuntimeClass* GetRuntimeClass() const;
  virtual void Serialize(CArchive& ar);
  virtual void AssertValid() const;
  virtual void Dump(CDumpContext& dc) const;
public:
  static AFX_DATA CRuntimeClass classCObject;
};
```

Единственной переменной этого класса является статическая переменная classObject типа CRuntimeClass, которая хранит информацию об объекте времени выполнения, ассоциированном с классом CObject.

Краткое описание функций-членов приведено ниже.

CObject::**CObject( )** —

защищенный конструктор, вызывается при сериализации и динамическом создании объектов.

CRunTimeClass* CObject::**GetRuntimeClass( )** —

виртуальная функция, которая возвращает указатель на структуру CRuntimeClass, ассоциированную с данным объектом; в нашей работе нам еще не приходилось сталкиваться с необходимостью переопределения этой функции.

Прежде чем давать комментарии к следующей функции, необходимо пояснить широко используемый в библиотеке MFC термин "сериализация". *Сериализация* (serialize) — это процесс сохранения текущего состояния объекта на каком-либо устройстве постоянного хранения (чаще всего в файле на диске) с возможностью последующего восстановления этого состояния.

Примечание

Serialize дословно переводится как "преобразование в последовательную форму представления", слишком громоздко, поэтому будем использовать термин "сериализация", тем более, что с момента выхода предыдущего издания нашей книги этот термин уже стал использоваться достаточно широко.

Ключевым моментом здесь является то, что объект сам отвечает за способ и формат записи и чтения своего состояния. Для этого ему достаточно пере-

определить базовую операцию сериализации, которую поддерживает класс CObject, а именно функцию Serialize:

void CObject::**Serialize**(CArchive& ar) —

виртуальная функция, которая служит для чтения объекта из архива или его записи в архив; если вы хотите полностью использовать возможности, которые дает процесс сериализации, необходимо переопределить эту функцию для каждого класса, информацию о котором вы хотите сохранить, при этом сначала должна вызываться функция базового класса; более подробно к описанию этого процесса и тесно связанного с ним класса CArchive мы вернемся позже.

void CObject::**AssertValid**( ) —

виртуальная функция, которая проверяет внутреннее состояние объекта на истинность; в отладочной версии этой функции отрицательный результат проверки позволяет завершить программу выдачей сообщения о номере строки и имени файла, где эта ошибка произошла; при работе со своим классом следует переопределить эту функцию, причем в переопределенной версии первой должна вызываться функция базового класса; поскольку функция объявлена как const, нельзя изменять состояние объекта во время тестирования.

void CObject::**Dump**(CDumpContext& dc) —

виртуальная функция, которая сохраняет содержимое объекта в специальном объекте класса CDumpContext, в качестве которого чаще всего используется afxDump — предопределенный объект класса CDumpContext, позволяющий послать информацию в окно отладки; и функция и предопределенный объект доступны только в отладочной версии приложения, создаваемой библиотекой.

BOOL CObject::**IsSerializable**( ) —

позволяет проверить возможность проведения сериализации объекта; возвращает значение TRUE, если в областях описания и реализации объекта использованы макросы DECLARE_SERIAL и IMPLEMENT_SERIAL.

BOOL CObject::**IsKindOf**(const CRuntimeClass* pClass) —

позволяет проверить, соответствует ли объект классу, переданному ей в качестве параметра, или какому либо производному от него; реализация этой функции нарушает принцип полиморфизма языка C++, поэтому не следует часто ее применять, лучше пользоваться имеющимися виртуальными функциями.

Класс CObject не поддерживает множественное наследование, т. е. классы, которые вы создаете, могут иметь только один CObject в качестве базового, и он должен быть "самым левым" в иерархии. Однако он позволяет иметь "справа от него", в качестве базовых, несколько структур и классов, не являющихся производными от него. Например,

```
class A : public Cobject        class B       class C
{ };                            { };          { };

class D : public A, B, C

{ };
```

Как уже отмечалось, каждый производный от CObject класс ассоциируется со структурой CRuntimeClass, которую можно использовать для получения информации времени выполнения об объекте или его базовом классе. Структура CRuntimeClass имеет следующее объявление:

```
struct CRuntimeClass
{
LPCSTR m_lpszClassName;
   int m_nObjectSize;
  UINT m_wSchema;
  CObject* (PASCAL* m_pfnCreateObject)();
#ifdef _AFXDLL
   CRuntimeClass* (PASCAL* m_pfnGetBaseClass)();
#else
   CRuntimeClass* m_pBaseClass;
#endif
  CObject* CreateObject();
  BOOL IsDerivedFrom(const CRuntimeClass* pBaseClass) const;
  ...
};
```

Ниже приведено описание ее членов.

LPCSTR CRuntimeClass::**m_lpszClassName** —

строка, содержащая имя класса.

int CRuntimeClass::**m_nObjectSize** —

размер объекта в байтах.

UINT CRuntimeClass::**m_wSchema** —

номер схемы ("версии") сериализации, которая позволяет идентифицировать и обрабатывать данные, созданные более ранней версией программы.

CObject* CRuntimeClass::(PASCAL* **m_pfnCreateObject**)( ) —

указатель на конструктор по умолчанию, который создает объект вашего класса (если класс не поддерживает динамическое создание, то здесь хранится значение NULL).

CRuntimeClass* CRuntimeClass::(PASCAL* **m_pfnGetBaseClass**)( ) —

если вы работаете с динамической версией MFC, то здесь хранится указатель на функцию, которая возвращает указатель на структуру CRuntimeClass базового класса.

CRuntimeClass* CRuntimeClass::**m_pBaseClass** —

используется, если вы работаете со статической версией MFC, и содержит указатель на структуру CRuntimeClass базового класса.

CObject* CRuntimeClass::**CreateObject**( ) —

эта функция используется для создания объектов классов, которые поддерживают динамическое создание.

BOOL CRuntimeClass::**IsDerivedFrom**(const **CRuntimeClass*** pBaseClass) —

позволяет определить, является ли класс, функция которого вызывается, производным от класса, указатель на который передается в качестве параметра.

Использование этой структуры должно сопровождаться обязательным включением в качестве информации времени выполнения одного из макросов DECLARE_DYNAMIC, DECLARE_DYNCREATE, DECLARE_SERIAL в области объявления и соответствующих им макросов IMPLEMENT_DYNAMIC, IMPLEMENT_DYNCREATE, IMPLEMENT_SERIAL в области реализации. Описание макросов приведено ниже в этой главе.

Наряду с рассмотренными функциями для получения структуры класса времени выполнения широко используется макрос RUNTIME_CLASS, который позволяет получить эту структуру по имени класса C++.

В заключение обзора корня иерархической структуры библиотеки MFC — класса CObject — рассмотрим четыре уровня функциональных возможностей, которые он нам предоставляет:

1. Базовые функциональные возможности — отсутствует поддержка информации о классе времени выполнения и сериализации, но включается управление диагностикой памяти.
2. Базовые функциональные возможности плюс поддержка информации о классе времени выполнения.
3. Базовые функциональные возможности плюс поддержка информации о классе времени выполнения и динамическое создание.
4. Базовые функциональные возможности плюс поддержка информации о классе времени выполнения, динамическое создание и сериализация.

Уровень функциональных возможностей задается вашим выбором определенных макросов объявления и реализации при создании классов, производных от CObject.

Информацию о классе времени выполнения CObject поддерживает посредством своей функции-члена IsKindOf, которая позволяет определить, относится ли объект к определенному классу или, может быть, является производным от него. Эта функция позволяет осуществить безопасное приведение типа в производном классе. Аргументом этой функции является объект класса CRuntimeClass, который можно получить, используя макрос

RUNTIME_CLASS с именем класса. Функция IsKindOf не поддерживает множественное наследование или виртуальные классы, хотя при необходимости можно использовать множественное наследование для ваших классов, производных от классов библиотеки MFC.

CObject также поддерживает динамическое создание — создание объекта определенного класса во время выполнения. Объект создается функцией-членом CreateObject структуры CRuntimeClass, как показано в примере:

```
CRuntimeClass* pRuntimeClass = RUNTIME_CLASS(CMyClass);
CObject* pObject = pRuntimeClass->CreateObject();
ASSERT(pObject->IsKindOf(RUNTIME_CLASS(CMyClass)));
```

Классы, определяющие архитектуру приложения

Классы этой категории поддерживают общие функциональные возможности большинства приложений. После создания каркаса приложения с использованием средства AppWizard, ваша задача — наполнить его функциональными возможностями, специфичными для решения конкретной задачи. Обычно для этого создаются новые классы из классов архитектуры путем добавления новых и/или переопределением существующих функций-членов.

AppWizard генерирует несколько типов приложений, все из которых используют каркас в различных вариантах. Приложения, реализованные на базе SDI (однодокументного интерфейса) и MDI (многодокументного интерфейса), позволяют в полной мере использовать часть каркаса, называемую *архитектурой "документ/представление"* (Document/View architecture). Другие типы приложений, такие как приложения на базе диалогового окна, формы или DLL, используют только некоторые особенности этой архитектуры.

Архитектура "документ/представление" включает один или несколько документов, их представлений и соответствующих фреймов окон. Объект шаблона документа ассоциируется с классами для каждого множества "документ/представление/фрейм окна".

Все приложения на базе MFC имеют, по крайней мере, два объекта: объект "приложение", производный от класса CWinApp, и некоторый объект "главное окно", производный (часто не напрямую) от класса CWnd. (Наиболее часто главное окно порождается от классов CFrameWnd, CMDIFrameWndalog или CDialog, каждый из которых произведен от CWnd).

Приложения, которые используют архитектуру "документ/представление", содержат дополнительные объекты:

- Объект-приложение, производный от класса CWinApp, как упоминалось выше

- Один или больше объектов-документов, произведенных от класса CDocument. Эти объекты служат для внутреннего представления данных, обрабатываемых в представлении. Они могут быть ассоциированы с файлом данных
- Один или больше объектов-представлений, произведенных от класса CView. Каждое представление является окном, которое присоединено к документу и ассоциировано с фреймом окна. Представление изображает и манипулирует данными, содержащимися в объекте-документе

Кроме того, такие приложения содержат фреймы окон (произведенных от CFrameWnd) и шаблонов документов (произведенных от CDocTemplate).

Классы приложения и поддержки потоков

Каждое приложение имеет один и только один объект-приложение, который координирует работу других объектов при запуске программы.

Библиотека MFC поддерживает множество потоков, выполняемых внутри приложения. Но все приложения обязаны иметь, по крайней мере, один поток, называемый *главным*, который использует объект класса CWinApp.

Класс CWinThread инкапсулирует часть возможностей операционной системы для работы с потоками. Кроме того, MFC содержит классы объектов синхронизации, предоставляя интерфейс C++ объектам синхронизации Win32.

Классы приложения и потоков

CWinApp	Инкапсулирует код инициализации, запуска и завершения приложения
CWinThread	Базовый класс для всех потоков; его можно использовать непосредственно, или произвести от него свой класс, если необходимо выполнение каких-либо своих функций
COLEControlModule	Предоставляет функции для инициализации модуля управления

Классы приложений ISAPI

CHttpFilter	Определяет фильтры, посылающие выбранные запросы HTTP на сервер ISAPI; см. раздел "Классы ISAPI" этой главы
CHttpServer	Расширяет функциональные возможности сервера ISAPI; см. раздел "Классы ISAPI" этой главы

Классы поддержки синхронизации

CSyncObject	Базовый класс для всех классов поддержки синхронизации
CCriticalSection	Обеспечивает исключительный доступ к некоторому объекту только одному потоку процесса

CSemaphore	Обеспечивает заданное количество одновременных обращений к объекту
CMutex	Обеспечивает исключительный доступ к объекту только для одного потока любого количества процессов
CEvent	Служит для уведомления приложения о наступлении заданного события

Связанные классы

CCommandLineInfo	Анализирует командную строку приложения
CWaitCursor	Меняет обычный курсор на "песочные часы"; используется для индикации выполнения длительных операций
CDockState	Содержит дескрипторы постоянного хранилища состояния данных для панелей элементов управления
CRecentFileList	Поддерживает список файлов, используемых в последнее время
CMultiLock	Обеспечивает ожидание сигнализации одного или всех заданных объектов классов CCriticalSection, CSemaphore, CMutex или Cevent
CSingleLock	Обеспечивает ожидание сигнализации объекта одного из классов CCriticalSection, CSemaphore, CMutex или CEvent

Классы маршрутизации команд

Когда пользователь взаимодействует с приложением, выбирая при помощи мыши элементы меню или кнопки панели инструментов, приложение посылает сообщения от объекта интерфейса пользователя соответствующему объекту-получателю команды. Классы-получатели команд, производные от CCmdTarget, включают CWinApp, CWnd, CDocTemplate, CDocument, CView и классы, производные от них. Каркас приложения поддерживает автоматическую маршрутизацию команд, так что они могут быть обработаны большинством соответствующих активных объектов в приложении.

CCmdTarget	Базовый класс для всех классов объектов, которые могут отправлять и получать сообщения
CCmdUI	Предоставляет программный интерфейс для обновления (разрешение, запрещение, контроль и/или очистка) объектов интерфейса пользователя, таких как элементы меню или кнопки панелей инструментов; объект класса CCmdUI посылает команду ON_UPDATE_COMMAND_UI на обновление интерфейса пользователя вашим обработчикам команд; этот процесс одинаков как для элементов меню, так и для кнопок

Классы документов

Объекты-документы, созданные из объектов-шаблонов документов, управляют данными приложения. Они взаимодействуют с объектами-представлениями, которые представляют клиентскую область окна, изображают данные документа и позволяют пользователю взаимодействовать с ними.

CDocument	Базовый класс для документов, определяющих приложение
COLEDocument	Используется для реализации составного документа OLE; см. раздел "Контейнерные классы OLE" этой главы
COLELinkingDoc	Предоставляет инфраструктуру для организации связывания объектов; см. раздел "Контейнерные классы OLE" этой главы
CRichEditDoc	Содержит список элементов клиентов OLE; см. раздел "Контейнерные классы OLE" этой главы
COLEServerDoc	Используется как базовый класс для классов документов приложения сервера; см. раздел "Серверные классы OLE" этой главы

Связанные классы

Объекты-документы могут записывать свое состояние в устройство хранения и считывать его из этого устройства. Библиотека MFC предоставляет класс CArchive, имеющий средства передачи данных документов в хранилище и обратно:

Carchive	Совместно с классом CFile реализует механизм непрерывного хранения объектов посредством сериализации

Документы могут также содержать OLE-объекты, абстрактным базовым классом для которых является класс CDocItem (и производные от него):

CDocItem	Абстрактный базовый класс; объекты классов, производных от него, представляют части документов

Классы шаблонов документов

Объекты-шаблоны документов координируют создание объектов-документов, представлений и фреймов окон, когда создается новый документ и/или представление.

CDocTemplate	Базовый класс для шаблонов документов; нет необходимости использовать этот класс непосредственно, лучше воспользоваться одним из его производных классов
CMultiDocTemplate	Шаблон для документов в MDI
CSingleDocTemplate	Шаблон для документов в SDI

Связанные классы

CCreateContext	Структура, посылаемая из шаблона документа в функции создания окна для координации создания объектов-документов, представлений и фреймов окон

Окна, блоки диалога и элементы управления

Класс CWnd (базовый класс для всех окон) и его производные классы инкапсулируют объекты — окна Windows. Этот класс может использоваться как самостоятельно, так и в качестве базового для производства новых классов. Производные классы, имеющиеся в библиотеке классов, представляют различные свойства окон.

CWnd	Базовый класс всех окон

Классы фреймов окон

В архитектуре "документ/представление" фреймы окон являются окнами, которые содержат представления. Они также поддерживают панели элементов управления, присоединенные к ним.

В приложениях с многодокументным интерфейсом главное окно является производным от CMDIFrameWnd, которое содержит фреймы документов, являющихся объектами CMDIChildWnd.

В приложениях с однодокументным интерфейсом главное окно является производным от CFrameWnd и содержит представление текущего документа.

CFrameWnd	Базовый класс для главного окна однодокументного интерфейса; кроме того, является базовым для всех других классов фреймов окон
CMDIFrameWnd	Базовый класс для главного окна многодокументного интерфейса
CMDIChildWnd	Базовый класс для окна документа многодокументного интерфейса
CMiniFrameWnd	Окно с заголовком половинной высоты, обычно показывающее границу плавающей панели инструментов
COLEIPFrameWnd	Предоставляет окно для представления, когда документ-сервер редактируется по месту; см. раздел "Серверные классы OLE" этой главы

Связанные классы

CMenu	Предоставляет интерфейс для доступа к меню приложения; позволяет динамически манипулировать меню во время выполнения, например, когда необходимо добавить или удалить элемент меню в зависимости от контекста; несмотря на то что наиболее часто меню используются с фреймами окон, можно также использовать их в блоках диалога и в дочерних окнах; этот класс инкапсулирует системный объект-меню

Классы представлений

Класс CView и его производные классы являются дочерними окнами, которые представляют клиентскую область фрейма окна. Представления отображают данные и позволяют пользователю вводить, редактировать или выбирать данные документа. Этот класс является базовым для представления специфических данных документа.

CView	Базовый класс представлений
CScrollView	Базовый класс для представления с возможностью автоматической прокрутки данных

Форматное представление и просмотр записей

Формы также обеспечивают прокрутку, но строятся на основе шаблонов блоков диалога.

Классы просмотра записей баз данных основываются на формах и имеют непосредственную связь с таблицами базы данных.

CFormView	Прокручиваемое представление, расположение которого определяется шаблоном блока диалога
CDaoRecordView	Отображает записи базы данных в элементах управления; см. раздел "Классы для работы с DAO" этой главы
CRecordView	Предоставляет вид формы, непосредственно соединенной с объектом класса CRecordSet; см. раздел "Классы для работы с ODBC" этой главы
CHtmlView	Предоставляет функциональные возможности элемента управления "Web-браузер" в контексте архитектуры "документ/представление"
COleDBRecordView	Отображает записи базы данных в элементах управления

Форматное представление на основе элементов управления

CCtrlView	Базовый класс для всех представлений, ассоциированных с элементами управления Windows

CEditView	Представление, которое содержит общий элемент управления Windows — поле, поддерживающее возможности редактирования, поиска, замены и прокрутки
CRichEditView	Представление, которое в дополнение к возможностям элемента управления EDIT поддерживает настройку шрифтов, цветов, форматирование абзацев и внедрение OLE-объектов
CListView	Представление, которое содержит элемент управления Windows — LIST BOX
CTreeView	Представление, позволяющее изображать значки со строками текста, иерархически упорядоченные в виде дерева

Связанные классы

CSplitterWnd	Окно, которое пользователь может разделять на несколько частей, их размеры при необходимости можно изменять; этот класс позволяет иметь несколько представлений внутри единственного фрейма окна
CPrintInfo	Содержит информацию о печати или задании предварительного просмотра
CRichEditDoc	Поддерживает список элементов-клиентов OLE, которые имеются в CRichEditView; см. раздел "Контейнерные классы OLE" этой главы
CRichEditCntrItem	Предоставляет доступ к элементам OLE, хранящимся в CRichEditView; см. раздел "Контейнерные классы OLE" этой главы

Примечание

Два последних класса совместно с CRichEditView реализуют возможности контейнера OLE.

Классы блоков диалога

Класс CDialog и его производные классы инкапсулируют функциональные возможности блока диалога. Поскольку блок диалога является специальным типом окна, то очевидно, что класс CDialog является производным от CWnd, и служит в качестве базового для всех блоков диалога, как модальных, так и немодальных. Кроме того, от него образованы классы стандартных блоков диалогов, таких как диалог для открытия или сохранения файла, выбора шрифта или цвета, инициализации операций поиска и замены или выполнения различных операций с OLE-объектами.

CDialog	Базовый класс блоков диалога
CDataExchange	Поддерживает обмен данными и проверку информации для блоков диалога

Стандартные диалоги

CCommonDialog	Базовый класс для всех общих блоков диалога
CFileDialog	Стандартный блок диалога для открытия и сохранения файла
CColorDialog	Стандартный блок диалога для выбора цвета
CFontDialog	Стандартный блок диалога для выбора шрифта
CFindReplaceDialog	Стандартный блок диалога для операций поиска и замены
CPrintDialog	Стандартный блок диалога для печати файла
CPageSetupDialog	Инкапсулирует сервис, предоставляемый блоком диалога Windows, обеспечивающий задание параметров страницы с дополнительной поддержкой установки и модификации границ печати

Классы стандартных блоков диалога OLE

Эти классы обеспечивают решение общих задач OLE, используя для этого стандартные блоки диалога OLE. Они также предоставляют составляющие интерфейса пользователя для функциональных возможностей OLE.

COLEDialog	Используется в рабочем окне и содержит общую реализацию для всех блоков диалога OLE; все классы блоков диалога в категории интерфейса пользователя произведены от этого базового класса; он не может использоваться непосредственно
COLEInsertDialog	Отображает блок диалога **Insert Object** (вставка объекта) для вставки нового связанного или внедренного объекта OLE
COLEPasteSpecialDialog	Отображает блок диалога **Paste Special** (специальная вставка) для реализации одноименной команды меню **Edit** (правка)
COLELinksDialog	Отображает блок диалога **Edit Links** (редактировать связи) для модификации информации о связанных элементах
COLEChangeIconDialog	Отображает блок диалога **Change Icon** (смена значка) для изменения значка, ассоциированного со связанным или внедренным элементом OLE
COLEConvertDialog	Отображает блок диалога **Convert** (конвертировать) для преобразования элементов OLE из одного типа в другой
COLEPropertiesDialog	Инкапсулирует общий блок диалога **OLE Properties** (свойства OLE), который предоставляет удобный подход для отображения и модификации свойств элемента OLE, совместимого со стандартом Windows

COLEUpdateDialog	Отображает блок диалога **Update** (обновить) для обновления всех связей в документе; блок диалога содержит линейный индикатор для отображения степени завершения процесса обновления
COLEChangeSourceDialog	Изображает блок диалога **Change Source** (изменить источник), для изменения связей приемника или источника
COLEBusyDialog	Представляет стандартные блоки диалога **Server Busy** (сервер занят) и **Server Not Responding** (сервер не отвечает), для обработки вызовов занятых приложений; обычно изображается автоматически при реализации COLEMessageFilter

Классы свойств и мастеров

Окна свойств, известные также как диалоги со вкладками, наиболее эффективны, когда нужно объединить вместе большое число элементов управления в одном блоке диалога.

CPropertyPage и CPropertyPageEx	Представляют отдельную страницу (вкладку) внутри окна свойств
CPropertySheetEx и CPropertySheet	Представляют фрейм для отображения и координации множества вкладок
COlePropertyPage	Изображает свойства элемента управления OLE с использованием графического интерфейса, похожего на блок диалога

Классы элементов управления

Классы элементов управления инкапсулируют ряд стандартных элементов управления Windows, а также включают дополнительные элементы — кнопки с битовыми образами и панели элементов управления.

Статический элемент управления

CStatic	Статически отображаемое окно; используется в качестве метки, рамки или разделителя других элементов управления в блоках диалога или окнах; может также содержать графические образы

Элементы редактирования текста

CEdit	Окно простейшего текстового редактора
CRichEditCtrl	Элемент управления, в котором пользователь может вводить и редактировать текст; в отличие от CEdit поддерживает форматирование символов и абзацев, а также объекты OLE

Элементы представления числовой информации

CSliderCtrl	Элемент управления, содержащий ползунок, который пользователь может перемещать, выбирая одно из возможных значений
CSpinButtonCtrl	Счетчик (пара кнопок со стрелками, нажимая которые пользователь может увеличивать или уменьшать некоторое значение)
CProgressCtrl	Индикатор — полоса, дискретно проградуированная слева направо и отображающая процесс выполнения операции
CScrollBar	Полоса прокрутки

Кнопки

CButton	Программный интерфейс для кнопок (BUTTON), флажков (CHECKBOX) или переключателей (RADIO BUTTON) в блоке диалога или окне
CBitmapButton	Кнопка с битовым образом вместо текста

Списки

CListBox	Список элементов, которые пользователь может просматривать и выбирать
CDragListBox	Дополнительно к возможностям, предоставляемым классом CListBox, позволяет пользователю перемещать элементы внутри списка
CCheckListBox	Список элементов с флажками, каждый из которых пользователь может установить или снять
CComboBox и CComboBoxEx	Элемент управления, объединяющий текстовое поле и список
CListCtrl	Набор элементов, содержащих значок и текст
CTreeCtrl	Иерархический (древовидный) список, где каждый элемент может содержать значок и текст

Панель инструментов и строка состояния

CToolBarCtrl	Представляет функциональные возможности общего элемента управления Windows — панель инструментов; в большинстве программ с применением MFC вместо этого класса используется класс CToolBar
CStatusBarCtrl	Горизонтальное окно, обычно разделенное на области, в которых приложение может отображать информацию о состоянии; в большинстве программ с использованием MFC вместо этого класса используется класс CStatusBar

Другие элементы управления

CAnimateCtrl	Отдельный видеоклип
CToolTipCtrl	Подсказка — небольшое всплывающее окно, в котором отображается короткий текст, описывающий назначение указанного объекта приложения
CHeaderCtrl	Заголовок или метка для столбца
CTabCtrl	Элемент управления, реализующий вкладки, например для смены отображаемых страниц (по аналогии с закладками в блокноте)
CHotKeyCtrl	Поддерживает создание горячей клавиши, которую пользователь может нажать для быстрого выполнения определенного действия
CDateTimeCtrl	Поддерживает отображения даты и/или времени
CMonthCalCtrl	Поддерживает создание календаря
CRebarCtrl	Поддерживает создание расширенной панели управления Windows

Связанные классы

CImageList	Предоставляет функциональные возможности списка образов Windows; в основном используется с классами CListCtrl и CTreeCtrl, реже с CTabCtrl; позволяет хранить множество битовых массивов или значков одного размера

Классы панелей элементов управления

Панели элементов управления присоединяются к фрейму окна. Они содержат кнопки, области отображения состояния или основываются на шаблонах диалога. Плавающие панели элементов реализуют присоединение их к объекту CMiniFrameWnd.

CControlBar	Базовый класс для панелей элементов управления, которые представляют собой окно, выровненное по углу фрейма окна; может содержать как оконные дочерние элементы управления, так и не оконные, например, кнопки панели инструментов
CToolBar	Представляет функциональные возможности общего элемента управления Windows — панель инструментов в более удобном для программиста виде, чем класс CToolBarCtrl
CStatusBar	Горизонтальное окно, обычно разделенное на области, в которых приложение может отображать информацию состояния; класс более удобен в использовании, нежели CStatusBarCtrl

CDialogBar	Предоставляет возможности, аналогично другим классам панелей элементов управления, но основывается на шаблонах диалога, и, соответственно, может содержать любые элементы управления
CRebar	Представляет функциональные возможности общего элемента управления Windows — расширенная панель инструментов в более удобном для программиста виде, чем класс CRebarCtrl

Классы для вывода на экран и печать

В Windows весь графический вывод осуществляется при помощи контекстов устройства (DC — device context). Библиотека MFC инкапсулирует классы различных типов контекстов устройств. Кроме того, имеются классы графических объектов, таких как битовые образы, кисти, палитры, перья и др.

Классы контекста устройства

Контекст устройства — это объект Windows, который содержит информацию об атрибутах рисования. Все вызовы для рисования осуществляются с использованием объекта контекста устройства, который представлен в библиотеке MFC базовым классом CDC. Дополнительные классы, производные от CDC, инкапсулируют специализированные функциональные возможности контекста устройства, включая поддержку метафайлов Windows.

CDC	Базовый класс для контекстов устройств; используется непосредственно для доступа ко всем экранным и неэкранным контекстам устройств, таких как принтеры
CPaintDC	Контекст экрана, используемый в функции OnPaint оконных классов; в конструкторе автоматически вызывается функция BeginPaint, а в деструкторе — EndPaint
CClientDC	Контекст экрана для клиентской области окна; используется, например, при рисовании в ответ на события от мыши
CWindowDC	Контекст экрана для всего окна, включая клиентскую и неклиентскую области
CMetaFileDC	Контекст устройства для метафайла Windows, который содержит последовательность команд GDI, необходимых для создания образа

Классы графических объектов

Эти классы определяют графические объекты, используемые для рисования в контексте устройства.

CGdiObject	Базовый класс для объектов GDI
CBrush	Инкапсулирует кисть GDI, которая может быть установлена текущей в контексте устройства; кисти используются для заливки замкнутых объектов
CPen	Инкапсулирует карандаш GDI; карандаши используются для рисования ограничивающих линий объектов
CFont	Инкапсулирует шрифт GDI
CBitmap	Инкапсулирует битовый образ GDI, предоставляя интерфейс для манипулирования им
CPalette	Инкапсулирует палитру цветов GDI для использования в качестве интерфейса между приложением и цветом выходного устройства, такого как экран
CRgn	Инкапсулирует регион GDI для манипулирования эллиптическими, многоугольными или нерегулярными областями внутри окна; используется совместно с функциями отсечения класса CDC

Связанные классы

CRectTracker	Изображение и обработка интерфейса пользователя для изменения размеров и перемещения прямоугольных областей

Классы простых типов данных

Эти классы инкапсулируют координаты рисования, строки символов и информацию о дате и времени, поддерживая удобный синтаксис языка C++. Они широко используются в качестве параметров функций-членов классов Windows в библиотеке MFC.

Классы CPoint, CSize и CRect соответствуют структурам POINT, SIZE RECT, представленным в Win32 SDK (возможно использование как объектов этих классов C++, так и структур языка C).

CPoint	Хранит пару координат (x,y)
CSize	Хранит размеры или относительные координаты позиции
CRect	Хранит координаты прямоугольной области
CString	Хранит строку символов
CTime	Хранит значение абсолютного времени и даты
COLEDateTime	Интерфейс к объектам OLE типа DATE; представляет значения времени и даты
CTimeSpan	Хранит относительные значения времени и даты

COLEDateTimeSpan	Хранит относительные значения времени и даты для объектов OLE типа DATE
COLECurrency	Интерфейс к объектам OLE типа CURRENCY, арифметического типа с фиксированной точкой (15 знаков до десятичной точки и 4 после)
COLEVariant	Интерфейс к объектам OLE типа VARIANT; данные этого типа могут храниться во многих форматах

Массивы, списки и ассоциативные списки

Для обработки агрегированных данных библиотека классов предоставляет группу классов *коллекций* — массивы, простые и ассоциативные списки, которые могут хранить различные объекты и предопределенные типы. Коллекции динамически изменяют свой размер. Эти классы могут использоваться в любом приложении, написанном как для Windows, так и нет. Однако они очень полезны для реализации структур данных, которые определяют классы документов в каркасе приложения. На основе этих классов можно создавать различные специальные классы коллекций.

- *Массивы* — это одномерная структура данных, занимающая непрерывное место в памяти. Они поддерживают достаточно быстрый прямой доступ к заданному индексом элементу. Однако массивы неэффективны при возникновении необходимости вставить элемент. Размер массива может изменяться.
- *Списки* похожи на массивы, но хранятся по-другому. Каждый элемент списка включает указатель на предыдущий и следующий элементы, делая его двусвязным списком. Он позволяет очень быстро осуществлять удаление и вставку элементов, т. к. для этого необходимо только изменить несколько указателей. К недостаткам следует отнести большие издержки на операции поиска в списке.
- *Ассоциативные списки* обеспечивают взаимосвязь между ключевым значением и значением элемента данных. На базе ассоциативных списков библиотека классов MFC поддерживает список, называемый *картой сообщений*, который хранит указатели на функции их обработки.

Классы шаблонов массивов, списков и ассоциативных списков

Эти классы коллекций являются шаблонами, параметры которых определяют типы хранящихся объектов. Классы CArray, CList и CMap используют также глобальные вспомогательные функции, которые предварительно должны настраиваться.

CArray	Шаблон класса для создания массивов произвольных типов
CList	Шаблон класса для создания списков произвольных типов
CMap	Шаблон класса для создания ассоциативных списков произвольных типов
CTypedPtrArray	Шаблон класса для массивов указателей
CTypedPtrList	Шаблон класса для списков указателей
CTypedPtrMap	Шаблон класса для ассоциативных списков указателей

Готовые к использованию классы массивов

CByteArray	Массив элементов типа BYTE
CWordArray	Массив элементов типа WORD
CDWordArray	Массив элементов типа DWORD
CUIntArray	Массив элементов типа UINT
CObArray	Массив указателей на объекты класса CObject или производных от него
CPtrArray	Массив указателей типа void
CStringArray	Массив объектов CString

Готовые к использованию классы списков

CObList	Связанный список указателей на объекты класса CObject или производных от него
CPtrList	Связанный список указателей типа void
CStringList	Связанный список объектов CString

Готовые к использованию классы ассоциативных списков

CMapPtrToPtr	Использует указатель типа void в качестве ключа для нахождения другого указателя типа void
CMapPtrToWord	Использует указатель типа void в качестве ключа для нахождения данных типа WORD
CMapStringToOb	Использует объект CString в качестве ключа для нахождения указателей на объекты класса CObject или производных от него
CMapStringToString	Использует объект CString в качестве ключа для нахождения другого объекта CString

CMapStringToPtr	Использует объект CString в качестве ключа для нахождения указателя типа void
CMapWordToOb	Использует данные типа WORD для поиска указателя на объект класса CObject или производных от него
CMapWordToPtr	Использует данные типа WORD для поиска указателя типа void

Классы для работы с файлами и базами данных

Эти классы позволяют хранить информацию в базе данных или в файле на диске. Они включают две группы классов для работы с базами данных на основе механизмов DAO и ODBC, которые имеют сходные функциональные возможности. Группа DAO реализована с использованием механизма Data Access Object — объект доступа к данным, группа ODBC использует механизм Open Database Connectivity — открытый доступ к базам данных. Кроме того, библиотека снабжена множеством классов для работы со стандартными файлами, потоками OLE и HTML.

Классы файлов ввода/вывода

Эти классы предоставляют интерфейс для традиционных дисковых файлов, потоков OLE и сокетов Windows. Все классы, производные от CFile, могут использоваться совместно с объектом класса CArchive для выполнения сериализации. Обычно нет необходимости образовывать из них свои классы. Если для создания приложения используется средство AppWizard, то по умолчанию предоставляется реализация обработчиков команд **Open** (открыть), **Save** (сохранить) и **Save as** (сохранить как) меню **File** (файл) для организации ввода/вывода (с использованием класса CArchive). В дальнейшем, возможно переопределить функцию Serialize для реализации необходимой сериализации документа.

CFile	Предоставляет интерфейс для бинарных файлов на диске
CStdioFile	Предоставляет интерфейс класса CFile с буферизацией потоков файлов на диске, обычно в текстовом режиме
CMemFile	Предоставляет интерфейс класса CFile для файлов в памяти
CSharedFile	Предоставляет интерфейс класса CFile для разделяемых файлов в памяти
COLEStreamFile	Использует интерфейс OLE IStream, предоставляя доступ класса CFile к составным файлам; этот класс (производный от CFile) разрешает сериализацию MFC с использованием структурированного хранилища OLE
CSocketFile	Предоставляет интерфейс класса CFile для сокетов Windows

Связанные классы

CHtmlStream	Обеспечивает кэширование вывода HTML; функциональные возможности похожи на предоставляемые классом CMemFile
CFileStatus	Структура, содержащая информацию о файле, такую как имя, атрибуты, длина и т. д.

Классы для работы с DAO

Эти классы, работая с другими классами приложения, дают легкий доступ к базам данных, поддерживающим механизм DAO, основанный на тех же принципах, что и Microsoft Visual Basic и Microsoft Access. Кроме того, классы DAO обеспечивают доступ к разнообразным базам данных, использующим драйверы ODBC. Приложениям, которые используют базы данных DAO, следует, по крайней мере, иметь в своем составе объекты классов CDAODatabase и CDAORecordset.

CDAOWorkspace	Управляет наименованием и парольной защитой сеанса работы с базой данных от начала и до завершения
CDAODatabase	Обеспечивает соединение с базой данных, посредством которого возможно выполнять операции над данными
CDAORecordset	Представляет множество записей, выбранных из источника данных
CDAORecordView	Изображает записи базы данных в элементах управления
CDAOQueryDef	Представляет определение запроса, обычно хранящегося в базе данных
CDAOTableDef	Представляет определение таблицы базы данных или присоединенной таблицы
CDaoFieldExchange	Обеспечивает обмен полей записей DAO, которые используются классами базы данных DAO; этот класс обычно непосредственно не используется

Связанные классы

CLongBinary	Инкапсулирует дескриптор хранилища для бинарного объекта большого объема, такого как битовая карта; объекты этого класса используются для управления большими объемами данных, хранящимися в таблицах базы данных

Классы для работы с ODBC

Эти классы, работая с другими классами приложения, обеспечивают простой доступ к различным базам данных, использующим драйверы ODBC.

Приложениям, которые используют базы данных ODBC, следует иметь в своем составе, по крайней мере, объекты классов CDatabase и CRecordset.

CDatabase	Инкапсулирует соединение с источником данных, посредством которого можно манипулировать данными
CRecordSet	Объединяет множество записей, выбранных из источника данных; этот класс поддерживает переход от записи к записи, обновление записей (добавление, редактирование и удаление), ограничение выбора некоторым фильтром, сортировку и параметризованный выбор с информацией, имеющейся или рассчитываемой во время выполнения
CRecordView	Предоставляет форму, непосредственно соединенную с объектом класса CRecordSet; механизм DDX (обмен данными с блоком диалога) позволяет обновлять данные; так же, как и все формы, представление записи базируется на ресурсах шаблона блока диалога; он поддерживает также перемещение между записями, их обновление и закрытие соответствующего объекта класса CRecordSet
CFieldEchange	Обеспечивает обмен данными в обоих направлениях между источником данных и их представлением в приложении; требуется только для пользовательских типов данных

Классы для работы в Internet и сетях

Классы этой категории обеспечивают обмен информацией с другими компьютерами, используя механизмы ISAPI (Internet Server API) или сокетов. Они включают также множество классов для создания ISAPI-расширений и множество классов для манипулирования сокетами Windows.

Классы ISAPI

Механизм ISAPI реализует интерфейс для сервера Internet. Например, сервер ISAPI является сервером Windows NT для запуска информационного сервера Internet.

Фильтры HTTP обрабатывают запросы сервера. Они могут использоваться для следующих типов операций:

- Аутентификация
- Сжатие данных
- Шифрование
- Регистрация

Классы фильтров

CHttpFilter	Определяет фильтры, посылающие выбранные запросы HTTP на сервер ISAPI
CHttpFilterContext	Управляет содержимым для фильтров HTTP; это вспомогательный класс для множественной обработки конкурирующих запросов объекта класса CHttpFilter

Серверные классы

Классы обеспечивают обслуживание запросов сервера, включая CGI (Common Gateway Interface).

CHttpServer	Расширяет функциональные возможности сервера ISAPI, обрабатывая запросы клиента, и включает в себя CGI
CHttpServerContext	Управляет содержимым расширенного сервера ISAPI; это вспомогательный класс для множественной обработки конкурирующих запросов объекта класса CHttpServer

Классы сокетов Windows

Сокеты Windows предоставляют независимый сетевой протокол коммуникаций между двумя компьютерами. Сокеты могут быть синхронными (приложение ждет, когда освободится коммуникация) или асинхронными (приложение продолжает выполняться, пока действует коммуникация).

CAsyncSocket	Инкапсулирует сокеты Windows API
CSocket	Реализует то же, что и CAsyncSocket, но на более высоком уровне абстракции; используется для синхронных операций
CSocketFile	Предоставляет интерфейс класса CFile для сокетов Windows

Классы для работы с OLE

Классы OLE работают с другими классами окна приложения, предоставляют легкий доступ к функциям OLE API. Используя OLE, можно:

- Создавать документы OLE, которые позволяют пользователям создавать и редактировать документы, содержащие данные, которые созданы другими приложениями, включая текст, графику, звук или другие типы данных
- Создавать объекты OLE, которые можно внедрять в документы OLE
- Использовать механизм drag-and-drop для копирования данных между приложениями
- Использовать автоматизацию OLE для контроля одной программы из другой

- Создавать элементы управления OLE и контейнеры из таких элементов управления

Контейнерные классы OLE

Эти классы используются в приложениях, использующих контейнеры. Оба класса COLELinkingDoc и COleDocument управляют коллекциями объектов класса COLEClientItem, которые используются для представления каждого элемента OLE в документе, внедренного в него или связаного с ним.

COLEDocument	Используется для реализации составного документа OLE; может использоваться как базовый для поддержки документов с контейнерами
COLELinkingDoc	Предоставляет инфраструктуру для организации связывания объектов
CRichEditDoc	Содержит список элементов OLE; используется совместно с CRichEditView и CRichEditCntrItem
COleClientItem	Класс элемента, который представляет сторону клиента при связывании и внедрении элементов OLE; свои элементы со стороны клиентов следует образовывать из этого класса
CRichEditCntrItem	Представляет доступ к стороне клиента OLE, хранящегося в элементе управления RICH EDIT, когда используется совместно с CRichEditView и CRichEditDoc

Серверные классы OLE

Эти классы используются в серверных приложениях. Документы сервера могут являться контейнерами, которые поддерживают связывание.

COLEServerDoc	Используется как базовый класс для классов документов приложения сервера; объекты этого класса представляют собой основу поддержки сервера через взаимодействие с объектами класса COLEServerItem; визуальные возможности становятся доступными при использовании архитектуры "документ/представление"
COLEServerItem	Используется для представления документа или его части, которые могут быть внедрены в другой документ или связаны с ним
COLEIPFrameWnd	Является окном для представления, когда документ-сервер редактируется по месту
COLEResizeBar	Предоставляет стандартный интерфейс пользователя для изменения размеров по месту; объекты этого класса всегда используются совместно с объектами класса COLEIPFrameWnd

COLETemplateServer	Используется для создания документов на базе архитектуры документ/представление; объект этого класса передает большую часть своей работы ассоциированному с ним объекту класса CDocTemplate

Классы OLE для передачи данных и drag-and-drop

Эти классы используются при передаче данных OLE. Они поддерживают передачу данных между приложениями с использованием буфера обмена и посредством механизма drag-and-drop ("перетаскивание").

COLEDropSource	Управляет перетаскиванием от начала до конца; позволяет определить, когда начинается и когда заканчивается операция перетаскивания; он также изображает курсор обратной связи во время перетаскивания
COLEDataSource	Используется, когда приложение предоставляет данные для перетаскивания; может быть представлен как объектно-ориентированный объект буфера обмена
COLEDropTarget	Представляет цель операции drag-and-drop; объект этого класса передается в окно, определяет, имеются ли в нем какие-либо перетаскиваемые данные, и реализует перетаскивание
COLEDataObject	Используется как получающая сторона для объекта класса COLEDataSource; он предоставляет доступ к данным, хранящимся в объекте класса COLEDataSource

Классы автоматизации OLE

Эти классы поддерживают автоматизацию *клиентов* (приложений, которые могут управляться другими приложениями). Автоматизация *серверов* (приложений, которые управляют другими приложениями) поддерживается посредством карт диспетчеризации.

COLEDispatchDriver	Используется при вызове сервера-автомата из сервера-клиента; ClassWizard использует этот класс в качестве базового при создании классов для сервера-автомата, предоставляемого библиотекой типов

Классы, связанные с OLE

Это основные классы, которые используются, когда создаются элементы управления OLE. Класс COLEControlModule в модуле элементов управления OLE похож на класс CWinApp в приложении. Каждый модуль содержит один или больше элементов управления OLE, которые представлены объектами класса COLEControl. Эти элементы управления связываются со своими

контейнерами, используя объекты CConnectionPoint. Классы CPictureHolder и CFontHolder инкапсулируют интерфейс OLE для рисунков и шрифтов, в то время как классы COLEPropertyPage и CPropertyExchange помогают реализовать страницы свойств для созданных элементов управления.

COLEControlModule	Заменяет класс CWinApp для модуля элемента управления OLE; предоставляет функции-члены для инициализации модуля элементов управления
COLEControl	Наследует все функциональные возможности объекта окна Windows и дополнительные возможности, специфичные для OLE, такие как запуск события и доступ к методам поддержки и свойствам
CConnectionPoint	Определяет специальный тип интерфейса, используемого для соединения с другими объектами OLE, называемый *точкой соединения*; точка соединения реализует интерфейс внешнего выхода, который в состоянии инициировать действия других объектов, таких как запуск событий и изменение уведомлений
CPictureHolder	Инкапсулирует функциональные возможности рисованных объектов Windows и интерфейса OLE IPicture; используется для реализации общих свойств элемента управления OLE Picture
CFontHolder	Инкапсулирует функциональные возможности шрифтовых объектов Windows и интерфейса OLE IFont; используется для реализации имеющихся свойств элемента управления OLE Font
CPropExchange	Поддерживает реализацию постоянных свойств для элемента управления OLE, аналогично классу CDataExchange для блоков диалога
COLEObjectFactory	Используется для создания элементов, которые запрашиваются из других OLE-контейнеров; этот класс служит базовым для большинства специфических типов "фабрик объектов", включая COLETemplateServer
COLEMessageFilter	Используется для управления соглашениями с *облегченными вызовами удаленных процедур* (Lightweight Remote Procedure Calls, LRPC)

Классы для отладки и обработки исключений

Эти классы обеспечивают поддержку обработки исключительных ситуаций в момент динамического распределения памяти при отладке и для передачи информации из функции, где произошло исключение, в функцию его обработки.

Классы поддержки отладки

CDumpContext	Служит приемником для дампа диагностики
CMemoryState	Структура, которая предоставляет мгновенный снимок используемой памяти; позволяет сравнивать ранний и поздний снимки памяти
CRunTimeClass	Класс поддержки контроля типов времени исполнения

Классы обработки исключений

Библиотека классов предоставляет механизм обработки исключений, базирующийся на классе CException. Использование объектов этих классов обеспечивает включение в код приложения обработки исключений.

CException	Базовый класс для обработки исключений
CArchiveException	Обработка исключений при работе с архивами
CDaoException	Обработка исключений при ошибках в операциях работы с базами данных на основе DAO
CDBException	Обработка исключений при ошибках в операциях работы с базами данных на основе ODBC
CFileException	Обработка исключений при работе с файлами
CMemoryException	Обработка исключений при работе с памятью
CNotSupportedException	Обработка исключений, возникающих при попытке использования неподдерживаемых возможностей
COLEException	Обработка исключений при работе с механизмом OLE; этот класс используется и для сервера и для контейнера
COleDispatchException	Обработка исключений при ошибках, возникающих во время автоматизации OLE
CResourceException	Обработка исключений, возникающих при загрузке ресурсов Windows
CUserException	Обработка исключений, используемая при остановке операций, инициированных пользователем

Глава 6

Макросы, глобальные функции и переменные

Знакомство с библиотекой будет неполным и не даст существенных преимуществ при работе с ней, если мы не рассмотрим включенные в нее макросы, глобальные функции и переменные.

Все макросы, глобальные функции и переменные можно разбить на несколько категорий: основные, работа с базами данных, работа в Internet, OLE и элементы управления OLE. Рассмотрим первую категорию (основные).

Имена практически всех глобальных функций начинаются с префикса Afx. Исключение составляют большинство функций работы с базами данных и функции, обеспечивающие обмен данными. Во всех без исключения именах глобальных переменных применяется префикс afx, а имена всех макросов записаны заглавными буквами.

Типы данных

Большинство типов данных полностью соответствует представленным в SDK (Software Development Kit), однако есть и специфичные для MFC.

BOOL	Булевское значение
BSTR	32-битный указатель на символ
BYTE	8-битное беззнаковое целое
COLORREF	32-битное значение, используемое как значение цвета
DWORD	32-битное беззнаковое целое или адрес
LONG	32-битное целое со знаком
LPARAM	32-битное значение, посылаемое в качестве параметра в оконную процедуру или функцию обратного вызова
LPCSTR	32-битный указатель на постоянную строку символов

LPSTR	32-битный указатель на строку символов
LPCTSTR	32-битный указатель на постоянную строку символов, которая переносима в Unicode и DBCS
LPTSTR	32-битный указатель на строку символов, которая переносима в Unicode и DBCS
LPVOID	32-битный указатель на неопределенный тип данных
LRESULT	32-битное значение, возвращаемое из оконной процедуры или функции обратного вызова
UINT	32-битное беззнаковое целое для Win32 и 16-битное для Windows 3.x
WNDPROC	32-битный указатель на оконную процедуру
WORD	16-битное беззнаковое целое
WPARAM	Значение, посылаемое в качестве параметра в оконную процедуру или функцию обратного вызова, 32-битное для Win32 и 16-битное для Windows 3.x
POSITION	32-битное беззнаковое целое значение, используемое для обозначения позиции элемента в коллекции
LPCRECT	32-битный указатель на немодифицируемую структуру RECT
HINSTANCE	32-битное беззнаковое целое для идентификации дескриптора экземпляра приложения

Примечание

Функция обратного вызова или *CallBack*-функция вызывается операционной системой по указателю, переданному ей ранее приложением.

Имеются также некоторые другие типы данных, которые будут описаны в соответствующих разделах.

Получение информации о приложении

Когда вы пишете приложение с использованием классов MFC, вы создаете единственный объект, производный от CWinApp. Для получения информации об этом объекте и связанных с ним параметрах из любого места программы разработчики предоставили следующие функции:

CWinApp* **AfxGetApp**( ) —

возвращает указатель на единственный объект CWinApp приложения.

LPCTSTR **AfxGetAppName**( ) —

возвращает строку, заканчивающуюся нулем, содержащую имя приложения.

HINSTANCE **AfxGetInstanceHandle**( ) —

возвращает дескриптор текущего приложения для исполняемого файла (EXE); в случае, если она вызвана из динамически подключаемой библиотеки, связанной с USRDLL-версией MFC, то возвращается HINSTANCE этой библиотеки DLL.

CWnd* **AfxGetMainWnd**( ) —

возвращает значение переменной m_pMainWnd объекта-приложения, если оно не является сервером OLE, или указатель на активное главное окно приложения в противном случае.

HINSTANCE **AfxGetResourceHandle**( ) —

возвращает дескриптор загруженных по умолчанию ресурсов приложения.

void **AfxSetResourceHandle**(HINSTANCE hInstResource) —

устанавливает дескриптор экземпляра приложения или модуля EXE или DLL, из которого будут загружены ресурсы приложения.

LPCTSTR AFXAPI **AfxRegisterWndClass**(
UINT nClassStyle,
HCURSOR hCursor = 0,
HBRUSH hbrBackground = 0,
HICON hIcon = 0) —

возвращает строку, заканчивающуюся нулем, которая содержит имя зарегистрированного класса. В качестве параметров в эту функцию передаются: nClassStyle — стиль или комбинация стилей класса окна Windows, которые будут рассмотрены при описании регистрации класса окна, hCursor — дескриптор ресурса курсора, hbrBackground — дескриптор ресурса кисти, определяющей цвет фона, hIcon — дескриптор ресурса значка.

BOOL AFXAPI **AfxRegisterClass**(WNDCLASS* lpWndClass) —

возвращает значение TRUE при успешной регистрации класса окна и FALSE при ошибке. В качестве параметра в функцию передается указатель на структуру WNDCLASS, которая должна содержать информацию, необходимую для успешной регистрации. При использовании этой функции для регистрации класса окна в модуле DLL следует иметь в виду, что имя зарегистрированного класса автоматически удаляется из системы, когда библиотека выгружается из памяти.

HINSTANCE AFXAPI **AfxLoadLibrary**(LPCTSTR lpszModuleName) —

загружает в память модуль (файл DLL или EXE), имя которого передается в качестве параметра lpszModuleName. Если загрузка прошла успешно, то возвращается дескриптор модуля и увеличивается счетчик ссылок для этой библиотеки; в противном случае возвращается NULL.

BOOL AFXAPI **AfxFreeLibrary**(HINSTANCE hInstLib) —

уменьшает счетчик ссылок для загруженного модуля DLL, определяемого параметром hInstLib. Когда счетчик ссылок становится равным нулю, модуль выгружается из адресного пространства текущего процесса.

Модель объекта времени выполнения

Как уже отмечалось в предыдущей главе, все классы, производные от CObject и CRuntimeClass приобретают определенные возможности по об-

служиванию объектов, включающие доступ к информации времени выполнения о классе, сериализацию и динамическое создание объекта.

Здесь описаны макросы, которые облегчают использование этих возможностей.

DECLARE_DYNAMIC(className) —

разрешает доступ к информации времени выполнения об объекте класса (должен использоваться при объявлении класса); className — действительное C++-имя класса без кавычек.

IMPLEMENT_DYNAMIC(className, baseClassName) —

генерирует код, необходимый для динамического получения информации времени выполнения об объекте класса, производного от CObject (должен использоваться в реализации класса).

DECLARE_DYNCREATE(className) —

разрешает динамическое создание объекта, производного от CObject, и получение доступа к информации времени выполнения об этом объекте (должен использоваться при объявлении класса).

IMPLEMENT_DYNCREATE(className, baseClassName) —

разрешает динамическое создание и доступ к информации класса времени выполнения (должен использоваться в реализации класса). Приложение использует эту возможность для динамического создания нового объекта, например, когда требуется прочитать объект из файла на диске во время процесса сериализации (должен использоваться в реализации класса).

При использовании этих макросов вы получаете возможность использования макроса RUNTIME_CLASS и функции CObject::IsKindOf для определения класса вашего объекта во время выполнения.

DECLARE_SERIAL(className) —

генерирует код C++, необходимый для сериализации и доступа к информации класса времени выполнения об объекте класса, производного от CObject (должен использоваться при объявлении класса). Определение этого макроса включает в себя все функциональные возможности, предоставляемые обоими макросами DECLARE_DYNAMIC и DECLARE_DYNCREATE.

IMPLEMENT_SERIAL(className, baseClassName, wSchema) —

генерирует код C++, необходимый для сериализации и доступа к информации класса времени выполнения об объекте класса, производного от CObject (должен использоваться в реализации класса). wSchema — переменная типа UINT, определяющая "номер версии" приложения, которой может быть закодирован архив, для возможности применения процесса десериализации. Этот номер не должен быть равен –1.

RUNTIME_CLASS(className) —

для объектов, производных от CObject, и объявленных с использованием макросов DECLARE_DYNAMIC, DECLARE_DYNCREATE или DECLARE_SERIAL возвращает указатель на структуру CRuntimeClass, которая соответствует имени класса className.

Диагностика объектов

Диагностический сервис, предоставляемый библиотекой MFC, значительно упрощает отладку приложений. Он включает макросы и глобальные функции, которые позволяют отслеживать распределение памяти, содержимое дампа объекта и печать отладочной информации во время выполнения.

Основные макросы

ASSERT(booleanExpression) —

прерывает выполнение программы, если вычисляемое выражение booleanExpression равно FALSE, и печатает сообщение об ошибке.

ASSERT_KINDOF(className, pObject) —

проверяет, является ли pObject объектом класса className, где className — имя класса, производного от класса CObject; этот макрос работает только в случае, если в области объявления класса используется макрос DECLARE_DYNAMIC или DECLARE_SERIAL.

ASSERT_VALID(pObject) —

используется для проверки внутреннего состояния pObject на доступность, проверяет pObject на NULL и вызывает затем его функцию-член AssertValid. Если хотя бы одна проверка приводит к возникновению ошибки, то макрос выводит сообщение аналогично ASSERT.

TRACE(exp) —

позволяет вывести на экран форматированную строку, определяемую выражением exp, аналогично функции printf для C-программ под MS-DOS, например:

```
TRACE("Проверка вывода строк %s и чисел %d",
      "Отладка программы", 5)
```

TRACE0,…, **TRACE3** —

упрощенные версии макроса TRACE, позволяющие вывести форматированную строку с числом аргументов от 0 до 3.

Перечисленные макросы работают только в отладочной версии приложения, создаваемой с помощью библиотеки.

VERIFY(booleanExpression) —

действует аналогично макросу ASSERT, но для рабочей версии библиотеки.

Основные глобальные переменные

CDumpContext **afxDump** —

предопределенная переменная, позволяющая послать информацию в окно отладчика.

BOOL **afxTraceEnable** —

используется для разрешения или запрещения работы макроса TRACE.

Форматирование строк и окно сообщения

Эта группа содержит три глобальные функции: две — для работы со строками и одну — для вывода произвольного сообщения в специальный блок диалога, называемый *окном сообщения*.

void **AfxFormatString1**(
CString& rString,
UINT nIDS,
LPCTSTR lpsz) —

загружает строку из ресурсов, определяемую идентификатором nIDS, в объект rString с одновременной заменой шаблона "%1" на строку, указанную в lpsz. Например, если в ресурсах определена строка "Для профессионального программирования 32-разрядных приложений читайте книгу %1", а lpsz указывает на строку "<Программирование для Windows с использованием библиотеки MFC>", то после выполнения этой функции rString будет содержать строку "Для профессионального программирования 32-разрядных приложений читайте книгу <Программирование для Windows с использованием библиотеки MFC>".

void **AfxFormatString2**(
CString& rString,
UINT nIDS,
LPCTSTR lpsz1,
LPCTSTR lpsz2) —

аналогична функции AfxFormatString1, но использует два шаблона ("%1" и "%2") и соответственно две строки для подстановки.

Последняя функция этой группы имеет две формы: со строкой текста и с идентификатором ресурса:

int **AfxMessageBox**(
LPCTSTR lpszText,
UINT nType = MB_OK,
UINT nIDHelp = 0)

или

int **AfxMessageBox**(
UINT nIDPrompt,
UINT nType = MB_OK,
UINT nIDHelp = (UINT) −1) —

выводят на экран окно сообщения. Первая форма этой функции выводит в окно сообщения строку lpszText и использует идентификатор nIDHelp для перехода к соответствующей теме справки при нажатии клавиши <F1>. Вторая форма функции использует для определения текста идентификатор строки ресурса, задаваемый параметром nIDPrompt. Если для nIDHelp задано значение по умолчанию –1, то nIDPrompt задает тему справки.

Обе эти функции возвращают одно из следующих значений:

- IDABORT — выбрана кнопка **Abort** (стоп)
- IDCANCEL — выбрана кнопка **Cancel** (отмена) или нажата клавиша <Esc>

- IDIGNORE — выбрана кнопка **Ignore** (пропустить)
- IDNO — выбрана кнопка **No** (нет)
- IDOK — выбрана кнопка **OK**
- IDRETRY — выбрана кнопка **Retry** (повтор)
- IDYES — выбрана кнопка **Yes** (да)
- 0 — окно сообщения не может быть создано

Параметр nType определяет тип окна сообщения и может принимать одно из значений:

- MB_ABORTRETRYIGNORE — окно содержит кнопки **Abort**, **Retry** и **Ignore**
- MB_OK — окно содержит только кнопку **OK**
- MB_OKCANCEL — окно содержит две кнопки **OK** и **Cancel**
- MB_RETRYCANCEL — окно содержит кнопки **Retry** и **Cancel**
- MB_YESNO — окно содержит кнопки **Yes** и **No**
- MB_YESNOCANCEL — окно содержит кнопки **Yes**, **No** и **Cancel**

Кроме того, можно использовать следующие константы для определения степени модальности блока сообщений:

- MB_APPLMODAL — модальность на уровне пользователя
- MB_SYSTEMMODAL — модальность на уровне системы
- MB_TASKMODAL — модальность на уровне задачи

Более подробно эти вопросы будут рассмотрены при описании блоков диалога.

Помимо кнопок, в окне сообщения можно разместить один из предопределенных (системных) значков:

- MB_ICONEXCLAMATION — восклицательный знак

- MB_ICONINFORMATION — символ "i" (информация)

- MB_ICONQUESTION — знак вопроса

- MB_ICONSTOP — значок "стоп"

При использовании в окне сообщения более одной кнопки можно также установить кнопку по умолчанию:

- MB_DEFBUTTON1 — первая кнопка по умолчанию
- MB_DEFBUTTON2 — вторая кнопка по умолчанию
- MB_DEFBUTTON3 — третья кнопка по умолчанию

ПРОГРАММИРОВАНИЕ С ИСПОЛЬЗОВАНИЕМ БИБЛИОТЕКИ MFC

Глава 7

Основные составляющие приложения на базе библиотеки классов MFC

Цель этой главы — проанализировать общую структуру программы для Windows, написанной с использованием возможностей, предоставляемых библиотекой классов MFC, и показать тесную взаимосвязь между классами библиотеки и интерфейсом прикладного программирования Win32 API.

Начнем с рассмотрения полного текста первого минимального приложения First, создающего простое окно с заголовком, бордюром и системным меню.

```
/////////////////////////////////////////////////////////
// Файл First.h
// Copyright (c) 1999 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////
#ifndef __AFXWIN_H__
  #error сначала надо включить 'Stdafx.h' для поддержки PCH
#endif
class CFirstApp : public CWinApp
{
public:
  CFirstApp();
// Переопределяемые функции
  //{{AFX_VIRTUAL(CFirstApp)
  public:
  virtual BOOL InitInstance();
  //}}AFX_VIRTUAL
};
/////////////////////////////////////////////////////////
// Файл First.cpp
// Copyright (c) 1999 Мешков А., Тихомиров Ю.
```

```
/////////////////////////////////////////////////////////
#include "Stdafx.h"
#include "First.h"
#include "MainFrm.h"
// CFirstApp
CFirstApp::CFirstApp()
{
}
// Может быть создан один и только один объект CFirstApp
CFirstApp theApp;

// Инициализация объекта CFirstApp
BOOL CFirstApp::InitInstance()
{
  const char *szWndClass = AfxRegisterWndClass(
    CS_HREDRAW | CS_HREDRAW,
    LoadStandardCursor(IDC_ARROW),
    (HBRUSH)COLOR_BACKGROUND + 1,
    LoadIcon(IDR_MAINFRAME));
  CMainFrame *pWnd = new CMainFrame;
  pWnd->CreateEx(
    0,                          // дополнительные стили
    szWndClass,                 // класс окна
    "Первая программа",         // заголовок
    WS_OVERLAPPEDWINDOW,        // стиль
    CW_USEDEFAULT, CW_USEDEFAULT, // левый верхний угол (x,y)
    CW_USEDEFAULT, CW_USEDEFAULT, // размеры (cx,cy)
    NULL,                       // родительское окно
    NULL);                      // идентификатор меню

  pWnd->ShowWindow(m_nCmdShow);
  m_pMainWnd = pWnd;
  return TRUE;
}
/////////////////////////////////////////////////////////
// Файл MainFrm.h
// Copyright (c) 1999 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////
```

```
class CMainFrame : public CWnd
{
public:
  CMainFrame();
public:
virtual ~CMainFrame();
};
/////////////////////////////////////////////////////////
// Файл MainFrm.cpp
// Copyright (c) 1999 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////
#include "Stdafx.h"
#include "First.h"
#include "MainFrm.h"
// Конструктор/деструктор CMainFrame
CMainFrame::CMainFrame()
{
}
CMainFrame::~CMainFrame()
{
}
/////////////////////////////////////////////////////////
// Файл Stdafx.h
// Copyright (c) 1999 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////
// Stdafx.h : заголовочный файл, подключающий стандартные,
// наиболее часто используемые заголовочные файлы
#define VC_EXTRALEAN // Исключает редко используемые части
                     // заголовочных файлов Windows
#include <Afxwin.h>  // Сердцевина MFC и стандартные компоненты
/////////////////////////////////////////////////////////
// Файл Stdafx.cpp
// Copyright (c) 1999 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////
// Stdafx.cpp : исходный файл, который подключает только
//              стандартные включаемые файлы
// First.pch : файл прекомпилированных заголовков
// Stdafx.obj : файл, содержащий прекомпилированные типы информации
#include "Stdafx.h"
```

```
/////////////////////////////////////////////////////////
// Файл Resource.h
// Copyright (c) 1999 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////
//{{NO_DEPENDENCIES}}
// Microsoft Developer Studio generated include file.
// Used by Template.rc
//
// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE        129
#define _APS_NEXT_COMMAND_VALUE         32771
#define _APS_NEXT_CONTROL_VALUE         1010
#define _APS_NEXT_SYMED_VALUE           103
#endif
#endif
/////////////////////////////////////////////////////////
// Файл First.rc
// Copyright (c) 1999 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////
...
/////////////////////////////////////////////////////////
// Icon
// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDR_MAINFRAME           ICON    DISCARDABLE     "res\\First.ico"
#endif    // Russian resources
...
```

Рис. 7.1. Главное окно программы First

Программа First строит обычное, традиционное окно, хорошо знакомое всем пользователям системы Windows (рис. 7.1). Его можно переместить по экрану, закрыть, свернуть (уменьшить до размера значка) или развернуть (увеличить до размеров экрана), а также изменить его размер.

Основное назначение этого примера в том, что он позволяет ознакомиться с принципами построения программ на базе библиотеки классов MFC.

Программа имеет два класса объектов: CFirstApp и CMainFrame. Первый позволяет создать объект "приложение", а второй — объект "главное окно". Каждое приложение, которое использует библиотеку MFC, может содержать только один объект "приложение", который конструируется вместе с другими глобальными объектами C++ (определение — CFirstApp theApp) и уже доступен, когда Windows вызывает функцию WinMain. На данном этапе знакомства это самый существенный момент для успешного программирования для Windows на базе MFC, и если вы его усвоили, то можно переходить к обсуждению имен, используемых библиотекой для классов, функций и переменных.

Соглашения об именах MFC

В качестве префикса, обозначающего имя класса, библиотека MFC использует заглавную букву "C" от слова "class" (класс), за которой идет имя, характеризующее его назначение. Например, CWinApp — класс, определяющий приложение, CWnd — базовый класс всех оконных объектов, CDialog — класс блоков диалога и т. д. Мы также будем придерживаться этого соглашения в приводимых примерах. Например, программа First содержит определения двух классов: CFirstApp и CMainFrame.

Для имен функций-членов классов используется три способа. При одном имя объединяет глагол и существительное, например, LoadIcon (загрузить пиктограмму) или DrawText (нарисовать текст). При втором имя функции-члена состоит только из существительного, например, DialogBox (блок диалога). Для функций, предназначенных для преобразования одного типа в другой, обычными являются такие имена, как XtoY (из X в Y).

Для членов классов библиотеки MFC принят следующий способ назначения имен: обязательный префикс m_ (от class member — член класса), за которым идет префикс, характеризующий тип данных, и завершается все содержательным именем переменной, например, m_pMainWnd, где p — префикс, описывающий указатель (для переменных, которые не являются компонентами класса, префикс m_ не используется, чтобы можно было отличить обычные переменные от членов любых классов, а не только классов библиотеки MFC). В своих примерах мы придерживаемся аналогичного способа формирования имен и рекомендуем вам поступать так же.

Включаемые файлы

Вы, наверное, уже обратили внимание на то, что одной из первых в нашей программе стоит строка #include "Stdafx.h". Этот включаемый файл (или include-файл) служит для подключения к программе необходимых ей библиотечных файлов. При написании больших приложений вам обязательно потребуется использовать заранее откомпилированные заголовочные файлы, что поможет существенно сократить требуемое время компиляции. Прекомпилированные заголовки полезны, если есть несколько заголовков, которые должны быть включены во все или в большинство ваших исходных файлов и не меняются между компиляциями. Для того чтобы продемонстрировать способ сокращения времени компиляции, мы использовали прекомпилируемый файл STDAFX.H, который должен включать в себя все наиболее часто используемые заголовочные файлы. В данном случае это единственный большой (143 Кб) файл AFXWIN.H, который содержит описания основных классов библиотеки и сводит воедино все включаемые файлы, необходимые для базового функционирования MFC: AFX.H, содержащий описания классов общего назначения, макросы, базовые типы данных MFC и подключающий стандартные библиотеки времени выполнения языка C, и файл AFXRES.H, куда включены стандартные идентификаторы ресурсов. По мере изложения материала мы ближе рассмотрим классы, определяемые во включаемых файлах MFC. Здесь же отметим изменения, которые претерпел основной включаемый файл Windows — WINDOWS.H. Как и раньше, каждое приложение Windows должно содержать этот файл. Отличие заключается в том, что в нынешнем виде он уже не похож на огромного "монстра", а разбит на четыре сравнительно небольших файла: WINDEF.H (содержит определения базовых типов), WINBASE.H (описывает базовые функции 32-разрядного API), WINGDI.H и WINUSER.H (включают объявления процедур GDI и USER, а также определения констант и макросов для соответствующих модулей системы).

В заключение обсуждения стандартных включаемых файлов скажем несколько слов об определении #define VC_EXTRALEAN, использование которого позволяет исключить из процесса компиляции редко используемые части заголовочных файлов Windows (для C- и C++-программ, не использующих MFC, используется определение #define WIN32_EXTRA_LEAN).

Функция WinMain

А где же функция WinMain, которая является точкой входа в любую программу для Windows? Пусть вас не удивляет, что в представленном листинге вы не найдете даже упоминания об этой функции. Дело в том, что создатели библиотеки MFC проделали большую работу и избавили вас от написания многих строк вспомогательного кода, предоставив возможность заниматься

только решением конкретной задачи. Однако при этом библиотека скрывает от программиста и часть того, что заставляет Windows работать. По этой причине мы будем углубляться в Windows немного больше, чем может показаться необходимым. Мы уверены, что приобретенное понимание позволит вам с большей легкостью использовать возможности библиотек MFC и SDK. Сказанное в полной мере относится к функции WinMain. Что же предлагают нам разработчики MFC в плане ее реализации?

Примечание

На самом деле функция WinMain операционной системой не вызывается. Вместо этого происходит обращение к стартовой функции из библиотеки времени выполнения C/C++. Компоновщик Visual C++ знает, что она называется _WinMainCRTStartup и отвечает за выполнение таких операций, как:

- Поиск указателя на полную командную строку нового процесса
- Поиск указателя на переменные окружения нового процесса
- Инициализация глобальных переменных из библиотеки времени выполнения языка C, доступ к которым обеспечивается включением файла STDLIB.H
- Инициализация блока памяти (heap), используемого функциями выделения памяти и процедурами низкоуровневого ввода/вывода языка C
- Вызов функции WinMain библиотеки MFC

Давайте подробно рассмотрим отрывок исходного текста из библиотеки:

```
int AFXAPI WinMain(
  HINSTANCE hInstance,
  HINSTANCE hPrevInstance,
  LPTSTR lpCmdLine,
  int nCmdShow)
{
  int nReturnCode = -1;
  CWinApp* pApp = AfxGetApp();
// Внутренняя инициализация
  if (!AfxWinInit(hInstance, hPrevInstance,
    lpCmdLine, nCmdShow))
    goto InitFailure;
// Глобальная инициализация приложения
  if (!pApp->InitApplication())
    goto InitFailure;
// Специфическая инициализация экземпляра приложения
  if (!pApp->InitInstance())
```

```
  {
    if (pApp->m_pMainWnd != NULL)
    {
      TRACE0("Warning: Destroying non-NULL m_pMainWnd\n");
      pApp->m_pMainWnd->DestroyWindow();
     }
    nReturnCode = pApp->ExitInstance();
   goto InitFailure;
  }
  nReturnCode = pApp->Run();
  InitFailure:
  AfxWinTerm();
  return nReturnCode;
}
```

Мы не будем повторять описание параметров, которые система Windows передает в программу. Этот вопрос рассматривался нами достаточно подробно в первой части. Вместо этого еще раз подчеркнем, что объект нашего приложения (класса CFirstApp) является глобальным, единственным и уже создан к моменту вызова WinMain. Глобальная функция AfxGetApp, реализованная в библиотеке MFC, как раз и занимается тем, что возвращает указатель на объект "приложение" (точнее, на его базовый класс):

```
CWinApp* pApp = AfxGetApp();
```

Для лучшего понимания тех задач, которые решает WinMain, рассмотрим структуру этой функции, упрощенное условное изображение которой представлено на рис. 7.2.

Рис. 7.2. Упрощенная структура функции WinMain

Все представленные здесь функции являются функциями-членами класса CWinApp — базового для нашего объекта "приложение". Чтобы двигаться дальше, рассмотрим его подробнее.

Класс CWinApp

Класс CWinApp является базовым классом, из которого мы образуем обязательный объект-приложение Windows. Основными задачами объекта этого класса являются инициализация и создание главного окна, а затем опрос системных сообщений. Если вспомнить, что процесс — каждый выполняемый экземпляр приложения (в предыдущих версиях применялся термин "задача"), то можно сказать, что объект класса CWinApp отвечает за создание процесса в системе. Например, щелчком на значке WordPad вы запускаете процесс, который вызывает приложение WordPad. В действительности все происходит несколько сложнее (одновременно формируется первичный поток, за создание и функционирование которого отвечает класс CWinThread — базовый для CWinApp), но на данном этапе знакомства с библиотекой MFC достаточно такого упрощенного представления.

Поскольку без этого класса нельзя создать ни одно приложение на базе библиотеки MFC, начнем с рассмотрения основных членов и функций-членов этого класса, определенных в AFXWIN.H. Члены класса CWinApp:

LPTSTR CWinApp::**m_pszAppName** —

определяет имя приложения (общедоступная переменная).

HINSTANCE CWinApp::**m_hInstance** —

копия параметра hInstance, передаваемого WinMain с логическим номером (дескриптором) приложения. Может быть получен с помощью глобальной функции AfxGetInstanceHandle (общедоступная переменная).

HINSTANCE CWinApp::**m_hPrevInstance** —

копия параметра hPrevInstance, передаваемого WinMain. В отличие от предыдущих версий Windows, где этот параметр указывал на предыдущий экземпляр приложения, для Win32 он всегда устанавливается в NULL. Различные варианты идентификации запущенных экземпляров программы, кроме уже встречавшихся, будут рассмотрены позже (общедоступная переменная).

LPTSTR CWinApp::**m_lpCmdLine** —

копия параметра lpCmdLine, передаваемого WinMain для указания на командную строку приложения (общедоступная переменная).

int CWinApp::**m_nCmdShow** —

копия параметра nCmdShow, передаваемого WinMain, который определяет режим первоначального отображения главного окна на экране (общедоступная переменная).

LPTSTR CWinApp::**m_pszExeName** —

содержит имя исполняемого модуля приложения без расширения. В общем случае не совпадает с m_pszAppName (общедоступная переменная).

LPTSTR CWinApp::**m_pszHelpFilePath** —

содержит полное имя пути к файлу справки приложения. По умолчанию инициализируется именем приложения с расширением HLP. При изменении места расположения и/или имени файла справки необходимо переопределить эту переменную. Обычно это делается в переопределенной функции приложения InitInsnance (общедоступная переменная).

LPTSTR CWinApp::**m_pszProfileName** —

содержит имя INI-файла приложения (общедоступная переменная).

LPCTSTR CWinApp::**m_pszRegistryKey** —

определяет полный ключ реестра, где хранится профиль установок приложения (общедоступная переменная).

В заключение перечисления членов класса CWinApp рассмотрим один член упоминавшегося класса CWinThread, который нам понадобится раньше, чем мы займемся потоками:

CWnd* CWinThread::**m_pMainWnd** —

в самом общем случае служит для хранения указателя на объект главного окна текущего потока. Если этот поток является основным (первичным) потоком приложения, то в этой переменной хранится указатель на объект главного окна приложения. Библиотека MFC автоматически завершает поток, когда закрывается окно, на которое ссылается m_pMainWnd. В случае основного (первичного) потока завершается работа всего приложения (общедоступная переменная).

Примечание

В версиях Visual C++ до 4.0 эта переменная принадлежала классу CWinApp.

Оставшиеся члены класса CWinApp будут рассмотрены ниже, при обсуждении соответствующих тем.

Перечисление основных функций-членов класса CWinApp начнем с конструктора, поскольку создание объекта начинается с него:

CWinApp::**CWinApp**(LPCTSTR lpszAppName = NULL) —

параметр lpszAppName — указатель на текстовую строку, которая содержит имя приложения, используемое системой Windows. Если этот аргумент отсутствует или равен NULL (по умолчанию), то для формирования имени приложения конструктор использует строку из файла ресурсов с идентификатором AFX_IDS_APP_TITLE (в ней вы можете указать имя главного окна вашего приложения), а при ее отсутствии — имя исполняемого файла. Этот параметр конструктор запоминает в члене класса m_pszAppName.

Следующая группа отвечает за инициализацию и содержит функции, отвечающие за внешний вид приложения и получение информации из INI-файла или системного реестра. Сюда входят:

void CWinApp::**LoadStdProfileSettings**(
UINT nMaxMRU = _AFX_MRU_COUNT) —

загружает список последних использованных (MRU — most recently used) файлов и последнее состояние предварительного просмотра. Если nMaxMRU = 0, то список файлов не поддерживается. Параметр _AFX_MRU_COUNT определяется в файле AFXWIN.H директивой препроцессора #define _AFX_MRU_COUNT 4.

void CWinApp::**SetDialogBkColor**(
COLOREF clrCtlBk = RGB(192, 192, 192),
COLOREF clrCtlText = RGB(0, 0, 0)) —

устанавливает цвет фона (clrCtlBk) и текста (clrCtlText), для блоков диалога и окон сообщений, используемых в приложении. Макрос RGB комбинирует интенсивности (в диапазоне от 0 до 255) трех цветов (красного, зеленого и синего) для получения требуемого цвета и определен следующим образом:

COLORREF **RGB**(int red, int green, int blue)
BOOL CWinApp::**Enable3dControlsStatic**( )

и

BOOL CWinApp::**Enable3dControls**( ) —

загружают CTL3D32.DLL для того, чтобы можно было выводить в блоках диалога и окнах элементы управления, которые имеют трехмерный вид. В этом случае отпадает необходимость в вызове функции SetDialogBkColor. Первый вариант функции используется при работе со статической, а второй — с динамической библиотекой. MFC автоматически поддерживает трехмерное представление для объектов следующих классов: CDialog, CDialogBar, CFormView, CPropertyPage, CPropertySheet, CControlBar и CToolBar. При необходимости получения трехмерного эффекта для других элементов управления следует непосредственно вызывать функции библиотеки API CTL3D32.DLL.

void CWinApp::**SetRegistryKey**(
LPCTSTR lpszRegistryKey)

и

void CWinApp::**SetRegistryKey**(
UINT nIDRegistryKey) —

заставляют приложение сохранять свои установки в системном реестре вместо INI-файла, устанавливая для этого член класса m_pszRegistryKey в значение, передаваемое в качестве параметра функции. При вызове этой функции список MRU-файлов также сохраняется в реестре. Параметры: lpszRegistryKey — указатель на строку, содержащую имя ключа; nIDRegistryKey — идентификатор/индекс ключа в реестре. Ключ реестра обычно хранится в следующем формате:

`HKEY_CURRENT_USER\Software\< названиеОрганизации>\`

`<имяПриложения>\<имяСекции>\<имяПеременной>`

void CWinApp::**EnableShellOpen**( ) —

разрешает пользователям приложения открывать файлы данных, обрабатываемые им, двойным щелчком на имени файла. Используется совместно с функцией RegisterShellFileTypes.

void CWinApp::**RegisterShellFileTypes**( ) —

регистрирует все типы документов вашего приложения в регистрационной базе данных. Эту функцию следует вызывать после формирования всех типов шаблонов документов, с

которыми вы будете работать. Если регистрационная база данных уже ассоциирует расширение имени файла с другим типом файла, то новое соответствие не устанавливается. Используется совместно с функцией EnableShellOpen.

Функции следующей группы (операции) позволяют определить курсор и значок приложения, получить из INI-файла или записать в него необходимую информацию, произвести анализ параметров, передаваемых из командной строки, и выполнить некоторые другие операции:

HCURSOR CWinApp::**LoadCursor**(
LPCTSTR lpszResourceName)

и

HCURSOR CWinApp::**LoadCursor**(
UINT nIDResource) —

загружают ресурс курсора, определяемый строковой константой lpszResourceName или номером идентификатора ресурса nIDResource, из текущего выполняемого файла. Курсор загружается в память, если он не был предварительно туда загружен; в последнем случае функции возвращают дескриптор существующего ресурса. Для загрузки предопределенных курсоров Windows следует использовать функции LoadStandardCursor или LoadOEMCursor.

HICON CWinApp::**LoadIcon**(
LPCTSTR lpszResourceName)

и

HICON CWinApp::**LoadIcon**(
UINT nIDResource) —

действие этих функций и параметры аналогичны LoadCursor, но для определения значка приложения. Для загрузки предопределенных значков Windows следует использовать функции LoadStandardIcon или LoadOEMIcon.

void CWinApp::**ParseCommandLine**(
CCommandLineInfo& rCmdInfo) —

осуществляет анализ командной строки и посылает параметры в член ParseParam класса CCommandLineInfo. Перед вызовом следует создать локальный объект класса CCommandLineInfo и передать ссылку на него в качестве параметра функции. При использовании AppWizard это производится автоматически.

BOOL CWinApp::**ProcessShellCommand**(
CCommandLineInfo& rCmdInfo) —

обрабатывает аргументы командной строки и флаги.

UINT CWinApp::**GetProfileInt**(
LPCTSTR lpszSection,
LPCTSTR lpszEntry,
int nDefault) —

при успешном завершении возвращает целое значение, заданное текстовой строкой lpszEntry внутри секции, определяемой строкой текста lpszSection, из INI-файла приложения или реестра; если функция не нашла запись, то возвращается значение параметра nDefault. Эта функция поддерживает шестнадцатеричную нотацию для значений в INI-файле или реестре.

BOOL CWinApp::**WriteProfileInt**(
LPCTSTR lpszSection,
LPCTSTR lpszEntry,
int nValue) —

записывает целое значение nValue в INI-файл приложения или реестр, в секцию, заданную строкой lpszSection, на место записи, определяемой текстовой строкой lpszEntry. Если секция и/или запись не определены, то они создаются этой функцией.

CString CWinApp::**GetProfileString**(
LPCTSTR lpszSection,
LPCTSRT lpszEntry,
LPCTSTR lpszDefault = NULL) —

возвращает значение строки из INI-файла приложения или реестра или lpszDefault, если строка не найдена. Максимальная поддерживаемая длина строки определяется параметром _MAX_PATH и составляет 256 символов, включая завершающий нуль. Если lpszDefault = NULL, то возвращается пустая строка. По принципу действия аналогична функции GetProfileInt.

BOOL CWinApp::**WriteProfileString**(
LPCTSTR lpszSection,
LPCTSRT lpszEntry,
LPCTSTR lpszValue) —

записывает строку lpszValue в определенную секцию INI-файла приложения или реестра, аналогично WriteProfileInt для целых чисел.

BOOL CWinApp::**GetProfileBinary**(
LPCTSTR lpszSection,
LPCTSTR lpszEntry,
LPBYTE* ppData,
UINT* pBytes) —

используется для чтения бинарного массива данных из INI-файла приложения или реестра, аналогично функции GetProfileInt для целых чисел. В случае успеха возвращает TRUE, параметр ppData содержит указатель на прочитанный массив данных, длина которого содержится по адресу pBytes. Память под данные, отведенную функцией GetProfileBinary, необходимо самостоятельно освободить после использования.

BOOL CWinApp::**WriteProfileBinary**(
LPCTSTR lpszSection,
LPCTSTR lpszEntry,
LPBYTE pData,
UINT nBytes) —

используется для записи в INI-файл приложения или реестр бинарного массива данных, заданного адресом (pData) и длиной (nBytes).

Следующая важная группа членов-функций, которую мы должны рассмотреть для того, чтобы двигаться дальше, — переопределяемые (виртуальные) функции. В документации на MFC обращается внимание на следующие функции этой группы: InitInstance, Run, ExitInstance и OnIdle.

virtual BOOL CWinApp::**InitInstance**( ) —

отвечает за инициализацию каждого нового экземпляра приложения.

Если посмотреть поставляемые исходные тексты, то можно увидеть, что в своем первозданном виде эта функция "ничего не делает":

```
BOOL CWinApp::InitInstance()
{
  return TRUE;
}
```

и наполнить ее содержанием — целиком задача программиста. Обычно это то место, где конструируется объект "главное окно" приложения, загружаются стандартные настройки из INI-файла или из реестра Windows, обрабатывается командная строка текущего экземпляра приложения, создается новый или открывается существующий документ, регистрируются шаблоны документов, создаются сами документы, их представления (view) и ассоциированные с ними окна (этот процесс будет подробно рассмотрен в одной из следующих глав). После создания главного окна переменная CWinThread::m_pMainWnd устанавливается равной значению указателя на это окно. Это единственная функция-член CWinApp, которую вы должны обязательно переопределить для того, чтобы ваше приложение "что-нибудь делало".

virtual BOOL CWinApp::**ExitInstance**( ) —

вызывается только из функции Run для завершения работы текущего экземпляра приложения. Реализованная версия этой функции записывает сохраняемые настройки в INI-файл приложения. Если необходимо провести специальную обработку при завершении приложения, такую как возврат памяти, распределенной во время работы, очистка контекста устройства или других ресурсов, то следует переопределить эту функцию. Освобождение стандартных элементов, таких как документы и их представления, производится стандартными средствами библиотеки классов MFC и не должно беспокоить разработчиков программного обеспечения.

virtual int CWinApp::**Run**( ) —

запускает цикл обработки сообщений, в котором сообщения Windows получаются и распределяются до тех пор, пока не поступит сообщение WM_QUIT, инициирующее завершение работы приложения. Все поступающие сообщения направляются в функцию PreTranslateMessage и затем в стандартную функцию Windows TranslateMessage для поддержки работы с клавиатурой. В заключение вызывается функция Windows DispatchMessage. Если очередь сообщений пуста, то Run вызывает виртуальную функцию OnIdle для выполнения фоновой обработки. Эта функция не требует переопределения, хотя и допускает его.

virtual BOOL CWinApp::**OnIdle**(LONG lCount) —

вызывается в цикле обработки сообщений, когда очередь сообщений приложения пуста. Функция, выполняемая по умолчанию, обновляет команды объектов пользовательского интерфейса, такие как элементы меню и кнопки панелей инструментов, и очищает внутренние структуры данных. Если вы переопределяете эту функцию, то в новой версии необходимо сначала вызвать базовую функцию CWinApp::OnIdle с параметром lCount. Параметр lCount увеличивается перед каждым вызовом функции, т. е. всякий раз, когда очередь сообщений приложения пуста, и сбрасывается в 0 при обработке нового сооб-

щения. Этот параметр можно использовать для определения различных режимов обработки или относительного времени "простоя" приложения.

virtual void CWinApp::**HideApplication**( ) —

скрывает приложение до закрытия открытых документов.

virtual void CWinApp::**DoWaitCursor**(int nCode) —

эта функция вызывается приложением при создании объекта класса CWaitCursor, выполнении функций CCmdTarget::BeginWaitCursor, CCmdTarget::EndWaitCursor и CCmdTarget::RestoreWaitCursor. Реализация по умолчанию выводит курсор в виде песочных часов и поддерживает счетчик ссылок. Если его значение положительно, то курсор отображается, а если отрицательно, то нет. Параметр nCode определяет отображение или скрытие курсора "песочные часы": nCode = 1 — курсор появляется, nCode = 0 — курсор восстанавливается без увеличения счетчика ссылок и nCode = –1 — курсор убирается с экрана. При переопределении функции можно изменить вид курсора, добавить дополнительную обработку на время его изображения и т. д.

virtual int CWinApp::**DoMessageBox**(
LPCTSTR lpszPrompt,
UINT nType,
UINT nIDPrompt) —

вызывается глобальной функцией AfxMesageBox для вывода на экран окна сообщений; непосредственно вызывать эту функцию не рекомендуется. Параметр lpszPrompt определяет адрес строки текста, выводимой в окне; nType — стиль окна сообщений; nIDPrompt — индекс строки контекстной справки, задаваемый в файле ресурсов. Можно переопределить эту функцию для настройки широкого круга процессов, вызывающих AfxMessageBox.

Мы не затронули некоторые имеющиеся виртуальные функции-члены, которые определяют работу с документами. Этой большой и во многом основной для работы с библиотекой классов MFC теме посвящена отдельная глава.

Теперь, после подробного знакомства с классом CWinApp, можно вернуться к нашей первой программе.

Минимальная программа для Windows

Напомним еще раз основные этапы создания и работы нашей первой программы:

1. Создается единственный глобальный объект класса приложения CFirstApp — theApp. Программа First использует класс CWinApp как базовый для собственного класса CFirstApp, который имеет две функции — конструктор и функцию, переопределяющую аналогичную функцию родительского класса:

```
class CFirstApp : public CWinApp
{
public:
  CFirstApp();
```

```
// Переопределяемые
// ClassWizard описывает здесь виртуальные функции
  //{{AFX_VIRTUAL(CFirstApp)
  public:
  virtual BOOL InitInstance();
  //}}AFX_VIRTUAL
};
```

2. После создания объекта theApp операционная система вызывает функцию WinMain — точку входа в приложение Windows. Реализация этой функции в библиотеке MFC выполняет следующие действия:
   - Определяет указатель на объект "приложение"

     ```
     CWinApp* pApp = AfxGetApp();
     ```

   - Вызывает функцию, отвечающую за инициализацию первого экземпляра вашей программы

     ```
     pApp->InitApplication();
     ```

   - Вызывает функцию, отвечающую за инициализацию текущего экземпляра приложения

     ```
     pApp->InitInstance();
     ```

   - Запускает цикл обработки сообщений

     ```
     pApp->Run();
     ```

   - Завершает функцию WinMain при завершении работы приложения

При всей своей лаконичности функция WinMain имеет все составляющие, необходимые для выполнения программы в Windows. Из них наибольший интерес представляют инициализация текущего экземпляра и цикл обработки сообщений, поэтому рассмотрим их подробнее. Любая программа, написанная на базе библиотеки классов MFC, переопределяет функцию InitInstance. В примере First это выглядит так:

```
BOOL CFirstApp::InitInstance()
{
  const char *szWndClass = AfxRegisterWndClass(
    CS_HREDRAW | CS_HREDRAW,
    LoadStandardCursor(IDC_ARROW),
    (HBRUSH)COLOR_BACKGROUND + 1,
    LoadIcon(IDR_MAINFRAME));
  CMainWnd *pWnd = new CMainWnd;
  pWnd->CreateEx(
    0,                              // дополнительные стили
```

```
    szWndClass,                    // класс окна
    "Первая программа",            // заголовок
    WS_OVERLAPPEDWINDOW,           // стиль
    CW_USEDEFAULT, CW_USEDEFAULT,  // левый верхний угол (x, y)
    CW_USEDEFAULT, CW_USEDEFAULT,  // размеры (cx, cy)
    NULL,                          // родительское окно
    NULL);                         // идентификатор меню

  pWnd->ShowWindow(m_nCmdShow);
  m_pMainWnd = pWnd;
  return TRUE;
}
```

Основное назначение этой функции, как вы помните, заключается в проведении некоторых инициализирующих действий (в нашей функции отсутствуют) и создание главного окна приложения, для чего необходимо, прежде всего, зарегистрировать класс нашего окна (подробно к вопросу регистрации мы вернемся в следующем разделе):

```
const char *szWndClass = AfxRegisterWndClass(
  CS_HREDRAW | CS_HREDRAW,
  LoadStandardCursor(IDC_ARROW),
  (HBRUSH)COLOR_BACKGROUND + 1,
  LoadIcon(IDR_MAINFRAME));
```

а затем создать экземпляр одного из оконных классов MFC:

```
CMainWnd *pWnd = new CMainWnd;
```

и создать собственно окно.

После того как объект окна MFC создан, указатель на этот объект присваивается переменной CWinThread::m_pMainWnd (как уже отмечалось, такое присвоение необходимо для того, чтобы иметь возможность корректно завершить приложение, когда будет закрыто его главное окно):

```
m_pMainWnd = pWnd;
```

можно создавать окно Windows и выводить его на экран

```
m_pMainWnd->ShowWindow(m_nCmdShow);
```

По завершении инициализации основная работа объекта "приложение" состоит в том, чтобы обеспечить передачу сообщений между пользователем и оконным объектом.

Примечание

Мы не описываем здесь те действия, которые необходимо проделать, чтобы посмотреть на результаты проделанной работы. Дело в том, что идти по описанному пути при создании приложений не следует — иначе в дальнейшем потребуется слишком много дополнительной работы. В Visual C++ 6 можно пользоваться "услугами" AppWizard для создания простейшего каркаса приложения *без* поддержки механизма архитектуры "документ/представление", что было невозможно в предыдущих версиях.

Тем не менее, нам показался более правильным именно такой подход, который позволяет рассмотреть внутренние механизмы библиотеки, не перегружая дополнительной информацией.

К этим вопросам мы еще вернемся, а пока рассмотрим процессы регистрации оконного класса и создания окна.

Регистрация класса окна

Любое окно Windows принадлежит одному из существующих в системе классов, который должен быть определен до того, как окно будет отображено на экране. Класс окна задает наиболее общие свойства окон, например форму курсора при перемещении его в область окна или имя меню, определенного для окон этого класса.

В библиотеке MFC для выполнения операции регистрации определены две глобальные функции — AfxRegisterClass и AfxRegisterWndClass. Первая требует заполнения полей структуры WNDCLASS, а вторая делает это сама, запрашивая только некоторые параметры. Но обо всем по порядку.

В Win32 для регистрации класса окна используются два типа структур: привычная WNDCLASS и новая WNDCLASSEX, которая отличается наличием двух полей — cbSize (определяет размер структуры в байтах и вычисляется как sizeof(WNDCLASS)) и hIconSm (задает идентификатор маленького значка, который будет отображаться в области системного меню; в Windows NT это поле не используется и должно быть установлено в NULL). Введение новой структуры пока не привело к ее широкому использованию и в MFC по-прежнему используется WNDCLASS, которая однако сменила "место прописки" и теперь располагается в WINUSER.H:

```
typedef struct tagWNDCLASSA {
  UINT style;
  WNDPROC lpfnWndProc;
  int cbClsExtra;
  int cbWndExtra;
  HINSTANCE hInstance;
  HICON hIcon;
```

```
  HCURSOR hCursor;
  HBRUSH hbrBackground;
  LPCSTR lpszMenuName;
  LPCSTR lpszClassName;
} WNDCLASS;
```

Рассмотрим поля этой структуры:

- lpszClassName — указатель на строку, содержащую имя класса. Поскольку после регистрации класс становится доступен всем другим приложениям Windows, имя класса должно быть уникальным
- lpfnWndProc — определяет адрес функции окна, которая обрабатывает сообщения для окон данного класса. В программах на базе MFC эта функция следит за вызовом правильной функции-обработчика сообщений
- hInstance — сообщает Windows о том, кто создает определение класса. Это необходимо для служебных действий внутри Windows. Когда завершается последний экземпляр программы, Windows удаляет все связанные определения классов
- cbClsExtra — задает число байт, которое необходимо дополнительно запросить у Windows под эту структуру для хранения собственных данных, присоединенных к классу
- cbWndExtra — задает число байт, которое необходимо дополнительно запросить у Windows для размещения всех структур, создаваемых совместно с данным классом, для хранения собственных данных, присоединенных к окну
- hIcon — определяет значок, который будет использоваться для изображения приложения, например на панели задач. Вы можете создать значок сами или использовать какой-либо из предопределенных:

IDI_APPLICATION	Стандартный значок для приложения
IDI_HAND	"Стоп"
IDI_QUESTION	"Вопросительный знак"
IDI_EXCLAMATION	"Восклицательный знак"
IDI_ASTERISK	"Информация"

- hCursor — идентифицирует курсор мыши, используемый в данном окне по умолчанию. Если нет желания создавать свой курсор, то можно воспользоваться одним из предоставляемых системой:

IDC_ARROW	Стрелка
IDC_IBEAM	Вертикальная черта
IDC_WAIT	"Песочные часы"

IDC_CROSS	Перекрестье
IDC_UPARROW	Вертикальная стрелка
IDC_SIZE	Четыре стрелки, указывающие в разные стороны
IDC_ICON	Курсор, используемый при переносе файлов
IDC_SIZENWSE	Двунаправленная стрелка "северо-запад/юго-восток"
IDC_SIZENESW	Двунаправленная стрелка "северо-восток/юго-запад"
IDC_SIZEWE	Двунаправленная стрелка "восток-запад"
IDC_SIZENS	Двунаправленная стрелка "север-юг"

- hbrBackground — задает цвет фона для окна. Для его определения можно использовать любую из 20 системных констант цвета, которые определены в файле WINUSER.H следующим образом:

```
#define COLOR_BACKGROUND     1
...
#define COLOR_APPWORKSPACE   12
```

  При использовании константы к ее значению обязательно нужно добавлять единицу, т. к. первое значение для них равно нулю, а ноль является недействительным значением для логического номера кисти. Кроме того, необходимо выполнить приведение к типу HBRUSH, иначе компилятор выдаст ошибку

- lpszMenuName — указатель на имя меню окна, определенное в файле ресурсов

- style — определяет свойства окна, которые могут комбинироваться при помощи оператора | (ИЛИ). При программировании для Windows этот термин относится к набору параметров, каждый из которых управляется одним или двумя битами. Если этому полю присвоить значение NULL, то Windows автоматически установит значения по умолчанию. Чтобы больше не возвращаться к этому вопросу, приведем здесь список допустимых стилей:

CS_BYTEALIGNCLIENT	Оказывает влияние на ширину окна, задавая выравнивание клиентской области по границе байтов; такое выравнивание позволяет увеличить производительность операций рисования в окне
CS_BYTEALIGNWINDOW	Устанавливает выравнивание окна по границе байтов; такое выравнивание позволяет увеличить производительность некоторых типов операций (перемещение и изменение размеров окна, рисование пунктов меню)
CS_CLASSDC	Обеспечивает контекст устройства, используемый совместно всеми окнами в классе

CS_DBLCLKS	Служит для установки таймера после получения первого сообщения о нажатии кнопки мыши; сообщение о двойном щелчке будет сформировано только в том случае, если второе нажатие произойдет до истечения определенного времени таймера; обеспечивает всем окнам класса возможность воспринимать сообщения о двойном щелчке одной из кнопок мыши
CS_GLOBALCLASS	Позволяет приложению создавать окно этого класса, не считаясь со значением параметра hInstance; если вы не определите этот стиль, то параметр hInstance, передаваемый в функции создания окна, должен быть таким же, как передаваемый в функцию RegisterClass; другими словами, установка этого стиля позволяет создавать оконные классы, предназначенные для совместного использования несколькими программами
CS_HREDRAW	Определяет, что окно будет перерисовываться, как только изменится его горизонтальный размер
CS_VREDRAW	Определяет, что окно будет перерисовываться, как только изменится его вертикальный размер
CS_NOCLOSE	Запрещает пункт **Close** (закрыть) системного меню; этот стиль следует использовать с окнами, имеющими системное меню, которые не могут быть закрыты пользователем
CS_OWNDC	Предоставляет частный контекст устройства для каждого окна в данном классе; этот тип контекста устройства наиболее расточителен в смысле использования памяти, однако обеспечивает самую высокую скорость реакции
CS_PARENTDC	Устанавливает, что дочерние окна не могут рисовать в своем родительском окне; этот стиль обычно применяется для предварительно определенных классов, необходимых для создания элементов управления блоков диалога
CS_SAVEBITS	Предписывает системе сохранять копию части экрана, поврежденной окном. После удаления окна поврежденная часть может быть восстановлена. При этом Windows не посылает сообщения WM_PAINT для перерисовки области, которую занимало окно. Этот стиль применяется для небольших окон (например, меню или блоков диалога), которые появляются на экране на короткие промежутки времени

Теперь, когда вы ознакомились с полями структуры WNDCLASS, можно их осмысленно заполнить и, передав адрес структуры в качестве параметра в функцию AfxRegisterClass, произвести регистрацию класса окна Windows.

Данный подход ничем не отличается от того, который применяется при программировании с использованием Win32 API. Совершенно естественно, что в библиотеке есть функция, которая выполняет большую часть работы по регистрации класса окна. Это описанная ранее глобальная функция AfxRegisterWndClass, которая позволяет пользователю определить стиль, цвет фона, курсор и значок для окон, создаваемых на базе этого класса. Остальные поля структуры класса заполняются автоматически. Обе функции возвращают имя зарегистрированного класса.

Однако чаще всего вообще не нужно думать о методе регистрации. Дело в том, что в библиотеке MFC созданы предопределенные классы окон, достаточные практически во всех возможных случаях:

"AfxWnd"	Используется для всех дочерних окон, создаваемых при помощи функции CWnd::Create; для этого класса определен курсор в виде стандартной стрелки и стиль CS_DBLCLKS | CS_HREDRAW | CS_VREDRAW
"AfxFrameOrView"	Используется для стандартных окон, созданных на базе классов CFrameWnd и CMDIChildWnd, и всех представлений; для этого класса определен курсор в виде стандартной стрелки, стиль CS_DBLCLKS | CS_HREDRAW | CS_VREDRAW, цвет фона WINDOW_COLOR и значок для окон SDI
"AfxMDIFrame"	Используется для окон MDI, созданных при помощи функции CMDIFrame::Create; для этого класса определен курсор в виде стандартной стрелки, стиль CS_DBLCLKS и значок для окон MDI
"AfxControlBar"	Используется для стандартных элементов управления; для этого класса определен курсор в виде стандартной стрелки и серый цвет фона (COLOR_BTNFACE)

Примечание

Если для окон многодокументного интерфейса вы захотите использовать различные значки, то следует зарегистрировать дополнительные классы окон или использовать функцию CFrameWnd::LoadFrame, которая автоматически регистрирует класс, используя стандартные атрибуты "AfxFrameOrView". Параметр, передаваемый в эту функцию, определяет идентификатор ресурса, содержащего значок. Более подробно мы расскажем об этом при описании класса CFrameWnd.

Примечание

Для клиентской области окна, созданного на базе класса CMDIFrameWnd, значения курсора и цвета фона не определяются. Дело в том, что эта область "покрывается" специальным окном MDICLIENT, которое использует только стандартные цвета и типы курсора. Более того, фирма Microsoft не поощряет создание каких-либо подклассов MDICLIENT.

Теперь, когда класс окна зарегистрирован, можно переходить непосредственно к созданию главного окна нашего приложения.

Создание окна

Каждая программа на базе MFC, которую вы создаете, будет содержать, по крайней мере, один класс, определяющий оконный объект. Приложение First определяет свой класс CMainFrame, который базируется на одном из предопределенных классов, а именно CWnd, и полностью использует тот прочный фундамент, который заложила библиотека MFC, не добавляя ничего своего:

```
class CMainFrame : public CWnd
{
public:
  CMainFrame();
  virtual ~CMainFrame();
};
```

В действительности все еще проще, поскольку этот класс можно было вообще не определять, т. к. обе его функции-члены пустые. Мы включили его только для наглядности. Всю работу выполняет базовый класс нашего окна CWnd. Еще раз повторим — окно создается в два-три шага: сначала определяется оконный класс, затем создается объект "окно", и наконец окно делается видимым.

Теперь о создании собственно окна Windows. Мы использовали для этой цели функцию CWnd::CreateEx:

```
    0,                          // дополнительные стили
    szWndClass,                 // класс окна
    "Первая программа",         // заголовок
    WS_OVERLAPPEDWINDOW,        // стиль
    CW_USEDEFAULT, CW_USEDEFAULT, // левый верхний угол (x,y)
    CW_USEDEFAULT, CW_USEDEFAULT, // размеры (cx,cy)
    NULL,                       // родительское окно
    NULL);                      // идентификатор меню
```

В дополнение к CreateEx класс CWnd имеет вторую функцию для создания окон — Create. Эти функции практически идентичны и имеют два отличия: во-первых, функция Create позволяет создавать лишь дочерние окна, в то время как CreateEx — также перекрывающиеся и вспомогательные, и, во-вторых, CreateEx содержит один дополнительный параметр (Ex означает extended — расширенный). Ввиду того что нам надо было создать главное окно программы, которое не может быть дочерним, мы воспользовались функцией CreateEx. Параметры обеих функций подробно описаны в главе 9, где речь идет о классах окон библиотеки MFC.

После возврата из CreateEx система Windows создает у себя всю внутреннюю информацию, необходимую для сопровождения окна, т. е. вносит запись в базу, содержащую информацию об окнах. Но в большинстве случаев окно на экране еще не появляется — требуется вызвать функцию ShowWindow, которая также принадлежит классу CWnd:

```
m_pMainWnd->ShowWindow(m_nCmdShow)
```

В зависимости от значения m_nCmdShow программа может при вызове иметь свернутое, обычное или развернутое окно. Вызовом ShowWindow завершается инициализация оконного объекта. Такая последовательность действий при инициализации типична для всех приложений Windows.

Глава 8

Обработка сообщений в библиотеке классов MFC

Цикл обработки сообщений MFC

При рассмотрении нашей первой программы мы совершенно не затронули такой важный вопрос, как обработка сообщений. И дело тут, как вы уже догадались, в том, что библиотека MFC опять взяла на себя решение этой важнейшей задачи. Ниже представлена функция Run из класса CWinThread, который отвечает за работу с потоками:

```
int CWinThread::Run()
{
  // Переменные для отслеживания состояния времени ожидания
  BOOL bIdle = TRUE;
  LONG lIdleCount = 0;
  // Получаем и распределяем сообщения, пока не получено
  // сообщение WM_QUIT
  for (;;)
  {
    // Фаза1: проверяем возможность работы в период ожидания
    while(bIdle &&
      !::PeekMessage(&m_msgCur, NULL,
      NULL, NULL, PM_NOREMOVE))
    {
      // Вызываем OnIdle, пока установлена переменная bIdle
      if (!OnIdle(lIdleCount++))
        bIdle = FALSE; // Присваиваем состояние "нет ожидания"
    }
    // Фаза2: подкачка сообщений, пока возможно
    do
```

```
    {
      // подкачка сообщений, и выход по WM_QUIT
      if (!PumpMessage())
        return ExitInstance();
      // сброс состояния "нет ожидания" после подкачки
      // "нормального" сообщения
      if (IsIdleMessage(&m_msgCur))
      {
        bIdle = TRUE;
        lIdleCount = 0;
      }
    }
    while(::PeekMessage(&m_msgCur, NULL,
      NULL, NULL, PM_NOREMOVE));
  }
}
```

Хотя показанный цикл внешне и отличается от стандартного (см. главу 4), но практически по всем компонентам они совпадают. Сообщения выбираются из очереди при помощи функции PeekMessage, а работу по поддержке клавиатуры, выбору оконной процедуры и передаче в нее сообщений выполняет недокументированная функция PumpMessage из класса CWinThread (вы можете в этом убедиться, если изучите ее исходный текст в файле ..\MFC\SRC\THRDCORE.CPP).

Основное отличие заключается в существовании специального механизма, позволяющего перехватить у системы циклы фоновой обработки путем вызова переопределяемой функции OnIdle класса вашего приложения. Такая возможность появляется, когда приложение принимает сообщения с помощью PeekMessage. Если очередь сообщений пуста, то функция возвращает FALSE. Отметим, что цикл обработки сообщений MFC требует специальной обработки сообщения WM_QUIT, поскольку PeekMessage не реагирует на это сообщение так, как это делает функция GetMessage. Эту работу выполняет упомянутая функция PumpMessage.

И еще несколько слов в заключение. Представленный цикл включает две фазы: первая предназначена для внутреннего употребления библиотекой MFC, а во время второй вы получаете возможность выполнять любые задачи. Более того, если очередь приложения остается пустой "достаточно длительное время", то вы можете получить дополнительные вызовы функции OnIdle в течение одного цикла фоновой обработки. При этом параметр lIdleCount позволяет отслеживать номер вызова. Но следует иметь в виду, что при поступлении очередного сообщения lIdleCount сбрасывается в нуль.

Конечно, не всякая программа в системе Windows требует фоновой обработки, и можно просто игнорировать эту возможность. Тем более, что реализация функции OnIdle в классе CWinApp обновляет объекты интерфейса пользователя и очищает внутренние структуры данных, т. е. выполняет за вас всю черновую работу.

Однако рассмотренный цикл обработки сообщений не объясняет, каким образом организуется обработка сообщений. Неужели необходимо переопределять оконную процедуру? Для ответа на этот вопрос необходимо рассмотреть, как организована обработка сообщений в библиотеке классов MFC.

Категории сообщений

В библиотеке MFC принята классификация сообщений, отличная от рассмотренной ранее. Все возможные сообщения разделены на три основные категории:

- Сообщения Windows
- Извещения элементов управления
- Командные сообщения (команды)

В первую категорию входят сообщения, имена которых начинаются с префикса WM_, за исключением WM_COMMAND. Все сообщения этой категории предназначены для обработки окнами и представлениями и часто содержат параметры, которые определяют алгоритм обработки того или иного сообщения. Сюда входят, например, аппаратные сообщения, сообщения обслуживания окна и т. д.

Вторая категория включает *извещения* (notification messages) от элементов управления и других дочерних окон, направляемые своим родительским окнам. Например, элемент управления LISTVIEW посылает своему родительскому окну сообщение WM_COMMAND, содержащее код извещения LVN_SETDISPINFO, когда требуется обновить информацию об элементах списка. Оконная процедура отвечает на полученное извещение заполнением структуры LV_DISPINFO и передачей ее обратно в элемент управления. Механизмы передачи извещений и других WM_-сообщений аналогичны, с одним единственным исключением. Извещение BN_CLICKED, посылаемое элементом управления BUTTON, когда пользователь щелкает на нем, трактуется как командное сообщение и передается аналогично другим командам.

Сообщения этих двух категорий предназначены для объектов классов, образованных от базового класса CWnd, т. е. имеющих дескриптор HWND окна Windows. Их обработка осуществляется в соответствующей оконной процедуре.

Наконец, третья категория охватывает все сообщения WM_COMMAND, называемые *командами* (или *командными сообщениями*), от объектов интер-

фейса пользователя, который включает меню, кнопки панелей инструментов и акселераторы. Обработка команд отличается от обработки других сообщений и может производиться множеством объектов (документы, шаблоны документов и сам объект "приложение").

Но независимо от способа разделения сообщений по тем или иным категориям, они существуют для того, чтобы их обрабатывали, т. е. при получении сообщения некоторая часть программы должна выполнить определенную последовательность действий. При рассмотрении оконной процедуры говорилось, что поступающие в нее сообщения обрабатываются индивидуально. Там же мы упомянули о существовании специального типа функций, каждая из которых отвечает за обработку одного и только одного сообщения. Аналогичный подход реализован и в библиотеке классов MFC, где для обработки каждого отдельного сообщения используется специальная функция-обработчик. Все функции-обработчики являются функциями какого-либо класса, и для них используются названия: *функция-член обработчик сообщения*, *функция обработчик сообщения* или просто *обработчик сообщения*. Принципы работы обработчиков сообщений не изменяются от того, к какой категории относится сообщение, однако механизмы их вызова различны. Но прежде чем рассматривать механизмы отправки и получения сообщений, необходимо понять, каким образом сопоставляются сообщение и соответствующий ему обработчик.

Карта сообщений

Библиотека классов MFC предоставляет альтернативу многочисленным операторам switch, используемым в традиционных Windows-программах для обработки сообщений, посылаемых окну. Взаимосвязь сообщений и их обработчиков может быть определена таким образом, что когда сообщение поступает в оконную процедуру, автоматически вызывается соответствующий обработчик. Реализация такой взаимосвязи основана на понятии *карты сообщений* (message map).

Примечание

С появлением файла заголовков WINDOWSX.H, а, точнее, ряда макроопределений, стал возможным отказ от операторов switch в традиционных программах на языке C.

Карта сообщений представляет собой механизм пересылки сообщений и команд Windows в окна, документы, представления и другие объекты приложения, реализованного на базе MFC. Такие карты преобразуют сообщения Windows, извещения элементов управления, а также команды меню, кнопок панелей инструментов, акселераторов в функции соответствующих классов, которые их обрабатывают. Эта отличительная черта карты сообще-

ний реализована по аналогии с виртуальными функциями C++, но имеет дополнительные преимущества, не доступные для них. Каждый класс, который может получить сообщение, должен иметь свою карту сообщений, чтобы иметь возможность соответствующим образом обрабатывать сообщения. При этом следует иметь в виду, что карта сообщений должна определяться вне какой-либо функции или объявления класса. Она также не может размещаться внутри C-блока.

Для определения карты сообщений необходимо использовать три макроса: BEGIN_MESSAGE_MAP, END_MESSAGE_MAP и DECLARE_MESSAGE_MAP.

Макрос DECLARE_MESSAGE_MAP располагается в конце объявления класса, использующего карту сообщений:

```
class CTheClass : public CBaseClass
{
// объявление компонентов класса
protected:
  //{{AFX_MSG_MAP(TheClass)
  afx_msg void OnPaint();
  //}}AFX_MSG_MAP,
DECLARE_MESSAGE_MAP()
};
```

Примечание

Если вы захотите объявить какие-либо компоненты класса после этого макроса, то для них нужно определить новый тип доступа (public, private или protected).

Он используется для объявления трех специальных членов класса:

- Частного массива компонентов AFX_MSGMAP_ENTRY — _messageEntries
- Защищенной структуры AFX_MSGMAP — messageMap, которая указывает на массив _messageEntries
- Защищенной виртуальной функции — GetMessageMap, которая возвращает адрес messageMap

Структура карты сообщений достаточно проста и представляет собой набор макросов, заключенных в специальные "операторные скобки":

Примечание

При использовании специальных средств автоматической генерации программного кода AppWizard и ClassWizard все три описанных макроса, а также специальные скобки для ClassWizard (//{{AFX_MSG_MAP(CTheClass) и //{{AFX_MSG_MAP) формируются автоматически при создании оконного класса.

```
BEGIN_MESSAGE_MAP(CTheClass, CBaseClass)
  //{{AFX_MSG_MAP(CTheClass)
   ON_WM_CREATE()
   ON_WM_DESTROY()
   ON_COMMAND(ID_CHAR_BOLD, OnCharBold)
   ON_UPDATE_COMMAND_UI(ID_CHAR_BOLD, OnUpdateCharBold)
  //{{AFX_MSG_MAP
  ON_NOTIFY(FN_GETFORMAT, ID_VIEW_FORMATBAR, OnGetCharFormat)
  ON_NOTIFY(FN_SETFORMAT, ID_VIEW_FORMATBAR, OnSetCharFormat)
END_MESSAGE_MAP()
```

Как видите, начинается она с выполнения макроса:

```
BEGIN_MESSAGE_MAP(CTheClass, CBaseClass)
```

который имеет следующие параметры: CTheClass — задает имя класса, владельца карты сообщений; CBaseClass — определяет имя базового класса.

Заканчивается карта сообщений вызовом макроса END_MESSAGE_MAP, имеющего еще более простой формат:

```
END_MESSAGE_MAP()
```

Между этими двумя вызовами располагаются специальные макросы, называемые *компонентами карты сообщений*. Вот они-то, собственно, и позволяют сопоставить сообщение с конкретным обработчиком.

Компоненты карты сообщений

Для каждого стандартного сообщения Windows определен свой макрос в форме:

```
ON_WM_XXX    // XXX — имя сообщения, например, ON_WM_PAINT
```

Имена обработчиков определяются при распаковке параметров каждого сообщения Windows на основе простого соглашения; имя всегда начинается с префикса On, за которым следует имя соответствующего сообщения Windows (без префикса WM_). Например, для сообщения WM_PAINT в классе CWnd определен обработчик

```
afx_msg void OnPaint();
```

а для сообщения WM_LBUTTONUP — обработчик

```
afx_msg void OnLButtonUp(
  UINT nFlags,
  CPoint point);
```

За формирование параметров, таких как nFlags и point, необходимых для каждого конкретного сообщения, отвечает соответствующий макрос. Описа-

ние всех обработчиков стандартных сообщений Windows можно найти в файле AFXWIN.H в объявлении класса CWnd.

Командные сообщения Windows от меню, акселераторов и кнопок панелей инструментов обрабатываются макросом

```
ON_COMMAND(id, memberFn)
```

Этот макрос использует в качестве параметров идентификатор команды id и произвольное имя обработчика команды memberFn. Прототип обработчика должен быть описан в соответствующем классе и иметь вид:

```
afx_msg void memberFn();
```

Макрос ON_COMMAND_EX, определяющий расширенную форму обработчика, является надмножеством над функциональными возможностями макроса ON_COMMAND. Такие обработчики команд принимают единственный параметр типа UINT, содержащий идентификатор команды, и возвращают значение типа BOOL: если возвращается FALSE, то команда передается для обработки другому объекту "получателю" команды; в противном случае обработка завершается.

Команды обновления выполняются посредством того же механизма, что и для "обычных" команд, только при этом используется макрос:

```
ON_UPDATE_COMMAND_UI(id, memberFn)
```

Обработчики таких команд имеют вид:

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

В обоих этих случаях, обработчик команды — memberFn — будет вызван только в случае, если в оконную процедуру поступит сообщение, которое в параметре wParam содержит идентификатор команды, совпадающий с id.

Следующая группа макросов позволяет сопоставить команды, посылаемые элементами управления и другими дочерними окнами своим родителям, с соответствующим обработчиком.

Для обработки извещений от элементов управления применяется макрос:

```
ON_CONTROL(wNotifyCode, id, memberFn)
```

Этот макрос использует в качестве параметров код извещения wNotifyCode от элемента управления, идентификатор элемента управления id и произвольное имя обработчика этой команды memberFn, прототип которого должен быть описан в соответствующем классе и иметь вид:

```
afx_msg void memberFn();
```

Так же как и для обработчиков команд, для каждого извещения необходимо использовать один, и только один макрос ON_CONTROL. Обработчик memberFn вызывается только в том случае, если код извещения от элемента управления wNotifyCode, такой как BN_CLICKED, совпадает с кодом изве-

щения, определенным в компоненте карты сообщений, и значение параметра id совпадает с идентификатором элемента управления.

Помимо рассмотренного, общего для всех элементов управления и дочерних окон, макроса, для кнопок (BUTTON), текстовых полей (EDIT), списков (LISTBOX) и комбинированных списков (COMBOBOX) используются специальные макросы, например:

```
ON_BN_CLICKED(id, memberFn)
ON_EN_SETFOCUS(id, memberFn)
ON_LBN_DBLCLK(id, memberFn)
```

и многие другие. Полный список макросов этой группы можно найти в документации или в файле AFXMSG.H. Принцип их действия полностью совпадает с принципом действия макроса ON_CONTROL.

С появлением в Windows 95 и Windows NT более сложных общих элементов управления (common controls), имеющихся макросов оказалось недостаточно, т. к. для них вместе с сообщением зачастую должны быть посланы некоторые данные. В связи с этим общие элементы управления Windows используют более мощное сообщение WM_NOTIFY, а в версиях Visual C++ начиная с 4.0, имеют прямую поддержку для этого сообщения на базе макроса:

```
ON_NOTIFY(wNotifyCode, id, memberFn)
```

Параметры этого макроса полностью аналогичны параметрам уже рассмотренного макроса ON_CONTROL. Единственное, но существенное отличие заключается в параметрах обработчика memberFn:

```
afx_msg void memberFn(
  NMHDR *pNotifyStruct,
  LRESULT *result);
```

В качестве параметров обработчика используются: pNotifyStruct — указатель на структуру типа NMHDR, или большую, включающую ее в себя; result — указатель на переменную, в которую должно быть записано возвращаемое значение.

Вот список извещений, общих для всех дополнительных элементов управления:

NM_CLICK	Пользователь нажал левую кнопку мыши в элементе управления
NM_DBLCLK	Пользователь сделал двойной щелчок левой кнопкой мыши на элементе управления
NM_RCLICK	Пользователь нажал правую кнопку мыши на элементе управления
NM_RDBLCLK	Пользователь сделал двойной щелчок правой кнопкой мыши на элементе управления
NM_RETURN	Пользователь нажал клавишу <Enter>, когда элемент управления имел фокус ввода

NM_SETFOCUS	Элемент управления получил фокус ввода
NM_KILLFOCUS	Элемент управления потерял фокус ввода
NM_OUTOFMEMORY	Элемент управления не может завершить какую-либо операцию из-за нехватки памяти

Примечание

Извещения, определенные для каждого конкретного элемента управления, будут описаны при обсуждении соответствующего элемента управления.

Если необходимо использовать один обработчик для нескольких команд или извещений от элементов управления, то можно воспользоваться соответствующими макросами:

```
ON_COMMAND_RANGE(idFirst, idLast, memberFn)
ON_CONTROL_RANGE(wNotifyCode, idFirst, idLast, memberFn)
ON_NOTIFY_RANGE(wNotifyCode, idFirst, idLast, memberFn)
```

Эти макросы используют в качестве параметров два идентификатора idFirst и idLast, которые составляют непрерывный ряд. Обработчик memberFn будет вызываться всякий раз, когда в оконную процедуру поступит сообщение, которое в параметре wParam содержит идентификатор команды, лежащий в границах, определяемых idFirst и idLast. Прототип обработчика должен иметь вид:

```
afx_msg void memberFn(UINT nID);
```

для первых двух и

```
afx_msg void memberFn(
  UINT nID,
  NMHDR *pNotifyStruct,
  LRESULT *result);
```

для последнего макроса.

Аналогичный макрос существует и для команд обновления:

```
ON_UPDATE_COMMAND_UI_RANGE(idFirst, idLast, memberFn)
```

Обработчик, как и для макроса ON_UPDATE_COMMAND_UI, имеет единственный параметр — указатель на объект CCmdUI:

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

Для извещений, по аналогии с макросом ON_COMMAND_EX, определены и макросы:

```
ON_NOTIFY_EX(nCode, id, memberFn)
ON_NOTIFY_EX_RANGE(wNotifyCode, idFirst, idLast, memberFn)
```

Команды, определяемые пользователем, включаются в карту сообщений при помощи макроса:

```
ON_MESSAGE(WM_NAMEMSG, OnNameMsg)
```

В качестве параметров этот макрос использует номер (WM_NAMEMSG) и имя (OnNameMsg) обработчика сообщения. Очевидно, что необходимо определить как номер сообщения, так и функцию его обработки:

```
#define WM_NAMEMSG (WM_USER + 100)
afx_msg LRESULT OnNameMsg(
 WPARAM wParam,
 LPARAM lParam
);
```

Ясно, что в этом случае всю ответственность по использованию параметров wParam и lParam берет на себя программист.

Сообщения, определяемые пользователем, включаются в карту при помощи макроса:

```
ON_REGISTERED_MESSAGE(nMessageVariable, memberFn)
```

В качестве параметров этот макрос использует идентификатор зарегистрированного сообщения nMessageVariable и имя обработчика memberFn. Обработчик должен иметь следующий прототип:

```
afx_msg LRESULT memberFn(
  WPARAM wParam,
  LPARAM lParam);
```

Для получения уникального идентификатора сообщения необходимо вызвать функцию Windows RegisterWindowMessage, описанную ниже в этой главе.

Теперь, ознакомившись с общей структурой карты сообщений и ее компонентами, можно переходить к вопросу о том, в какой последовательности и кем осуществляется поиск обработчика сообщения или команды. Если внимательно посмотреть на иерархию классов библиотеки MFC, то можно заметить, что все классы, способные получать сообщения, являются производными от класса CCmdTarget.

Класс CCmdTarget

Подобно тому как класс CObject является базовым для практически всех классов библиотеки MFC, класс CCmdTarget выступает в качестве базового для всех классов, которые могут работать с сообщениями.

Начнем с рассмотрения функций, входящих в состав класса CCmdTarget, и определенных в файле AFXWIN.H:

void CCmdTarget::**BeginWaitCursor( )** —

изображает курсор в виде песочных часов. Обычно вызывается, чтобы показать пользователю, что выполняется длительная операция, такая как загрузка файла или сохранение документа в файле на диске.

void CCmdTarget::**EndWaitCursor( )** —

восстанавливает предыдущее изображение курсора, которое было изменено на "песочные часы" при вызове функции BeginWaitCursor.

void CCmdTarget::**RestoreWaitCursor( )** —

восстанавливает соответствующий курсор в виде песочных часов, после того как он был заменен на системный.

virtual BOOL CCmdTarget::**OnCmdMsg(**
UINT nID,
int nCode,
void* pExtra,
AFX_CMDHANDLERINFO *pHandlerInfo) —

устанавливает маршрут и распределяет командные сообщения. Если рассмотреть поставляемые исходные тексты, то видно, что это единственная функция (не считая оконной процедуры по умолчанию), которая вызывается в оконной процедуре. Это означает, что именно она занимается поиском соответствующих обработчиков сообщений.

В качестве параметров функция OnCmdMsq принимает: идентификатор команды — nID; код извещения — nCode; значение параметра pExtra определяется значением параметра nCode; pHandlerInfo — обычно этот параметр равен NULL, в противном случае функция сама заполняет элементы pTarget и pmf структуры pHandlerInfo.

Если функция нашла своего адресата, который обработает сообщение, то она возвращает значение TRUE, в противном случае FALSE.

Таким образом, основным назначением функции OnCmdMsg является распределение команд по другим объектам. В принципе, можно переопределить эту функцию. Но при этом не следует забывать, что вам потребуется указать следующего получателя команды, чтобы не прервалась цепочка передачи сообщений.

К практическим аспектам обработки сообщений Windows и извещений элементов управления мы приступим, когда ознакомимся с имеющимися в библиотеке MFC классами, отвечающими за работу с окнами, а пока рассмотрим достаточно важный вопрос — каким образом осуществляется поиск обработчика команды.

Стандартный маршрут команды

Когда Run — функция класса CWinThread — получает сообщение, она направляет его в соответствующее окно-получатель сообщения, где оно будет обрабатываться. Несколько иначе все выглядит для команд. Поскольку они

формируются в результате взаимодействия пользователя с приложением, то, обычно, команда начинает свой путь к получателю от главного окна приложения. Напомним, что каждый объект, способный получать сообщения, располагает своей картой сообщений. Что же происходит, когда какой-либо объект получает сообщение? А все очень просто. Имея свою карту сообщений, получатель ищет в ней поступившую команду (или сообщение) и в случае успеха запускает соответствующий обработчик. Сразу же возникает вопрос: а если не найдет? Что ж, в этом случае поиск переносится в карту сообщений базовых классов. А если и там нет ассоциированного обработчика, то главное окно передает эту команду следующему кандидату на обработку. Для его определения в библиотеке MFC определен стандартный маршрут, по которому передается команда. Этот процесс продолжается до тех пор, пока команда не будет выполнена или отправлена оконной процедуре, действующей по умолчанию.

Порядок, в котором адресаты передают команды, "зашит" в библиотеке классов MFC и представляет собой следующую последовательность шагов:

1. Команду получает фрейм MDI (объект класса CMDIFrameWnd или производного от него). Порядок получателей команды:
   - Активное окно класса CMDIChildWnd
   - Сам объект класса CMDIFrameWnd
   - Приложение-объект класса CWinApp или производного от него
2. Команду получает фрейм документа (объект класса CFrameWnd, CMDIChildWnd или производного от одного из них). Порядок получателей команды:
   - Активное представление (view)
   - Текущий фрейм документа
   - Приложение-объект класса CWinApp или производного от него
3. Команду получает представление. Порядок получателей команды:
   - Текущее представление
   - Документ, присоединенный к представлению
4. Команду получает документ. Порядок получателей команды:
   - Текущий документ
   - Шаблон документа, присоединенный к нему
5. Команду получает блок диалога. Порядок получателей команды:
   - Текущий блок диалога
   - Окно, которое владеет этим блоком диалога
   - Приложение-объект класса CWinApp или производного от него

Очевидно, что представленные здесь переходы взаимозависимы. Например, если команду получает документ, присоединенный к текущему представлению, то следующим объектом, которому будет передана команда, является текущий документ, а затем его шаблон.

Чтобы сказанное стало более понятным, рассмотрим небольшой пример. Пользователь работает с MDI-приложением и хочет выделить весь текущий документ (для реализации этого действия в объекте класса документа приложения предусмотрен специальный обработчик). Он выбирает пункт меню **Edit\Select All** (правка\выделить все), с которым ассоциирована команда ID_EDIT_SELECT_ALL. Первым эту команду получает главное окно приложения, которое дает возможность обработать его текущему активному дочернему окну MDI. До того как дочернее окно просмотрит свою карту сообщений, оно дает возможность обработать команду своему представлению. Представление проверяет свою карту сообщений и, поскольку не находит там обработчика, направляет команду в ассоциированный с ним документ. Теперь уже документ ищет обработчик в своей карте сообщений, находит его, и начинает обрабатывать команду. Если бы мы не реализовали обработчик в нашем объекте-документе, то команда была бы направлена в шаблон, и в случае неудачи поиск продолжался бы в объекте класса CMDIFrameWnd, а затем в объекте класса приложения. Наглядно описанная последовательность действий представлена на рис. 8.1. Для команды, совершающей этот или любой другой маршрут, вызывается метод OnCmdMsg следующего объекта.

Рис. 8.1. Пример обработки действия пользователя

Стандартное сообщение Windows, в отличие от команд, не использует какой-либо маршрут для поиска обработчика, т. к. оно посылается непосредственно в оконную процедуру того объекта, который должен соответствующим образом отреагировать на него. Но и в этом случае каждый оконный класс, образованный непосредственно или косвенно от класса CWnd, имеет собственную карту сообщений, которая устанавливает связи между сообще-

ниями и соответствующими обработчиками. Если соответствующий обработчик не найден в карте сообщений некоторого класса, то он ищется в карте сообщений базового для него класса. При этом более эффективным является использование механизма виртуальных функций, которые определены в классе CWnd для стандартных сообщений Windows.

Команды обновления и класс CCmdUI

Когда пользователь, например, раскрывает меню, каждый его элемент "должен знать", каким он будет отображен — доступным, отмеченным или нет. Получатель команды предоставляет ему эту информацию, реализуя обработчик сообщения ON_UPDATE_COMMAND_UI, внутри которого используется специальный класс CCmdUI, определяющий элемент меню или кнопку панели инструментов, или любой другой объект интерфейса пользователя, способный формировать команды. Этот обработчик вызывается до вывода элементов интерфейса пользователя на экран (или их перерисовки), что позволяет задать их внешний вид. Маршрут, по которому ищется обработчик, совпадает с тем, который действует для "обычных" командных сообщений (рис. 8.2).

Рис. 8.2. Пример обновления элемента меню

Рассмотрим те возможности, которые заложены в классе CCmdUI для обновления элементов интерфейса пользователя. Начнем с членов класса:

UINT CCmdUI::**m_nID** —

идентификатор объекта интерфейса пользователя.

UINT CCmdUI::**m_nIndex** —

индекс объекта интерфейса пользователя.

CMenu CCmdUI::**m_pMenu** —

указатель на меню, представляемое объектом CCmdUI. Если представляется не меню, то этот параметр равен NULL.

CMenu CCmdUI::**m_pSubMenu** —

указатель на имеющееся подменю, представляемое объектом CCmdUI. Если представляется не меню, то этот параметр равен NULL. Если подменю является раскрывающимся, то параметр m_nID содержит идентификатор его первого элемента.

CWnd* CCmdUI::**m_pOther** —

указатель на объект окна, который послал извещение, например, панель элементов управления или строка состояния. Если извещение пришло от меню или не от оконного объекта, то этот параметр равен NULL.

Кроме представленных компонентов, у класса имеются 5 функций:

virtual void CCmdUI::**Enable**(BOOL bOn = TRUE) —

блокирует (bOn = FALSE) или разблокирует (bOn = TRUE) элемент интерфейса пользователя, ассоциированный с этой командой.

virtual void CCmdUI::**SetCheck**(int nCheck = 1) —

устанавливает элемент интерфейса пользователя в одно из возможных состояний в зависимости от значения передаваемого параметра: 0 — непомеченное, 1 — помеченное, 2 — неопределенное. Последнее состояние применимо только к кнопкам панелей инструментов.

virtual void CCmdUI::**SetRadio**(BOOL bOn = TRUE) —

блокирует (bOn = FALSE) или разблокирует (bOn = TRUE) группу переключателей.

virtual void CCmdUI::**SetText**(LPCTSTR lpszText) —

устанавливает текст для данного элемента интерфейса пользователя.

void CCmdUI::**ContinueRouting**( ) —

позволяет направить команду на обработку дальше по маршруту. Эта функция действует совместно с обработчиком команды ON_COMMAND_EX, который возвращает FALSE.

Практические аспекты использования перечисленных здесь возможностей будут рассмотрены при обсуждении вопросов организации интерфейса пользователя.

В заключение общего обзора принципов работы с сообщениями, заложенными в библиотеке классов MFC, и несколько забегая вперед, рассмотрим основные функции, которые предназначены для работы с сообщениями.

Функции для работы с сообщениями

Понимать работу с сообщениями крайне важно для того, чтобы писать правильные и эффективные программы для Windows. Поэтому рассмотрим основные функции, которые вы можете (и должны) широко использовать:

static const MSG* CWnd::**GetCurrentMessage**( ) —

возвращает указатель на структуру MSG, которая содержит информацию о текущем сообщении. Может быть вызвана только из обработчика, заданного макросом ON_MESSAGE.

LRESULT CWnd::**Default**( ) —

служит для вызова оконной процедуры, выполняемой по умолчанию.

virtual BOOL CWnd::**PreTranslateMessage**(MSG *pMsg) —

используется классом CWinApp для фильтрации оконных сообщений до того, как они будут распределены функциями Windows TranslateMessage и DispatchMessage.

LRESULT CWnd::**SendMessage**(
UINT msg,
WPARAM wPrm = 0,
LPARAM lPrm = 0) —

посылает сообщение в объект CWnd, непосредственно вызывая оконную процедуру, и не выходит из нее, пока та не обработает сообщение.

BOOL CWnd::**PostMessage**(
UINT msg,
WPARAM wPrm = 0,
LPARAM lPrm = 0) —

помещает сообщение в очередь приложения и заканчивает работу без ожидания его обработки. Из очереди сообщения извлекаются функциями Windows GetMessage или PeekMessage.

BOOL CWnd::**SendNotifyMessage**(
UINT msg,
WPARAM wPrm = 0,
LPARAM lPrm = 0) —

посылает в окно определенное извещение. Если окно было создано текущим потоком, то функция для обработки извещения вызывает оконную процедуру и не завершается, пока оно не будет обработано. Если же окно было создано другим потоком, то, послав извещение в оконную процедуру, функция завершает свою работу.

UINT ::**RegisterWindowMessage**(LPCTSTR lpString) —

позволяет получить идентификатор (в диапазоне от 0xC000 до 0xFFFF) нового уникального сообщения, который может затем использоваться при вызовах функций SendMessage или PostMessage. В качестве параметра lpString она использует указатель на строку, определяющую имя сообщения. Эта функция должна использоваться для предотвращения возможных конфликтов в случае, когда нужно связать между собой два (или больше) приложения. Зарегистрированное сообщение действует до окончания сеанса работы Windows. Обработчик сообщения ассоциируется с ним посредством специального макроса ON_REGISTERED_MESSAGE, например, как показано ниже:

```
class CUserWnd : public CParentWnd
{
...
```

```
  //{{AFX_MSG_MAP(CUserWnd)
    afx_msg LRESULT OnSample(WPARAM wParam, LPARAM lParam);
  //{{AFX_MSG_MAP
DECLARE_MESSAGE_MAP()
};
UINT WM_SAMPLE = RegisterWindowMessage("MESSAGE_SAMPLE");
BEGIN_MESSAGE_MAP(CUserWnd, CParentWnd)
  //{{AFX_MSG_MAP(CUserWnd)
    ON_REGISTERED_MESSAGE(WM_SAMPLE, OnSample)
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

Наиболее интересными из перечисленных являются две функции: SendMessage и PostMessage (рис. 8.3):

Рис. 8.3. Посылка сообщений функциями SendMessage и PostMessage

Основная цель этих функций — посылка сообщения в окно своего или некоторого другого приложения. Но этим их сходство и ограничивается. Дело в том, что функция SendMessage явно вызывает процедуру того окна, которое ассоциировано с объектом, вызывающим ее, и не выходит из процедуры, пока та не обработает сообщение. Другими словами, выполнение программы не будет продолжено до тех пор, пока сообщение не будет обработано.

В отличие от нее, функция PostMessage не вызывает явно оконную процедуру, а посылает сообщение в очередь сообщений приложения, откуда оно выбирается с помощью функции GetMessage или PeekMessage. Выполнение самой функции PostMessage при этом завершается и работа приложения может быть продолжена.

Пора переходить от теории, пусть даже и очень важной, к практике. Неоднократное упоминание классов оконных объектов (и даже некоторое знакомство с ними) — одной из важнейших составляющих библиотеки классов MFC — диктует необходимость их скорейшего рассмотрения.

СОЗДАНИЕ SDI- И MDI-ПРИЛОЖЕНИЙ НА БАЗЕ БИБЛИОТЕКИ MFC

Глава 9

Классы окон библиотеки MFC

Вспомним, что Windows — это операционная система, которая, прежде всего, работает с окнами. Пришло время подробно познакомиться с ними.

Объекты оконных классов библиотеки MFC, естественно, отличаются от окон Windows, но тем не менее тесно с ними связаны. Полное и правильное понимание этой взаимосвязи — залог успеха при создании приложений для Windows.

Как любой объект C++, объект "окно" создается и уничтожается, соответственно, конструктором и деструктором класса. Это с одной стороны. А с другой, окно Windows — это внутренняя структура данных операционной системы, которая формируется функцией интерфейса прикладного программирования Win32 API Create, а разрушается функцией DestroyWindow.
В системе Windows каждое окно идентифицируется своим дескриптором, значение которого хранится в члене m_hWnd объекта CWnd (рис. 9.1).

Рис. 9.1. Объект класса CWnd связан с окном Windows через дескриптор m_hWnd

Переходить непосредственно к созданию и работе с окнами нельзя, на наш взгляд, не рассмотрев подробно типы окон, имеющиеся в Windows.

Окна, определенные в ОС Windows

Существует всего три основных типа окон:

- Перекрывающиеся (overlapped)

- Всплывающие или вспомогательные (pop-up)
- Дочерние (child)

Из них программист может создать множество самых разнообразных объектов, комбинируя предопределенные биты стиля, имена которых имеют префикс WS_ (от window style — стиль окна). Рассмотрим более подробно эти биты, их влияние на внешний вид и некоторые свойства окон:

WS_BORDER	Окно имеет рамку без заголовка
WS_CAPTION	Окно имеет заголовок и рамку; как правило, этот стиль используется для перекрывающихся окон и не может применяться совместно со стилем WS_DLGFRAME
WS_CHILD	Дочернее окно; не может использоваться совместно со стилем WS_POPUP
WS_CLIPCHILDREN	Исключает область, занятую дочерним окном, из области рисования родительского окна; используется только для родительских окон
WS_CLIPSIBLINGS	Исключает все другие дочерние окна из своей области рисования; другими словами, если дочерние окна перекрываются, а этот стиль не указан, то при изменении рабочей области одного из окон могут быть испорчены рабочие области других дочерних окон; этот стиль используется только вместе со стилем WS_CHILD
WS_DISABLED	Неактивное окно, т. е. сразу после его создания оно недоступно для ввода с клавиатуры или с помощью мыши
WS_DLGFRAME	Окно имеет двойную рамку и не имеет заголовка
WS_GROUP	Определяет первый элемент управления группы окон, к которым пользователь может переходить при помощи клавиш со стрелками; все элементы управления, определенные с этим стилем, заканчивают текущую и начинают новую группу (одна группа заканчивается там, где начинается другая)
WS_HSCROLL	Окно имеет горизонтальную полосу прокрутки
WS_VSCROLL	Окно имеет вертикальную полосу прокрутки
WS_MAXIMIZE	Окно будет отображено в развернутом виде
WS_MAXIMIZEBOX	Окно имеет кнопку "Развернуть"; если окно является дочерним окном элемента управления, то этот флаг стиля используется под другим именем — WS_TABSTOP
WS_MINIMIZE	Окно будет отображено в виде значка; используется только со стилем WS_OVERLAPPED
WS_MINIMIZEBOX	Окно имеет кнопку "Свернуть"; если окно является дочерним окном элемента управления, то этот флаг стиля используется под другим именем — WS_GROUP

WS_OVERLAPPED	Окно является перекрывающимся; обычно имеет заголовок и рамку
WS_POPUP	Всплывающее окно; не может использоваться совместно со стилем WS_CHILD
WS_SYSMENU	Окно имеет значок системного меню в полосе заголовка
WS_TABSTOP	Назначается одному или нескольким элементам управления для того, чтобы между ними можно было перемещаться с помощью клавиши табуляции (<Tab>)
WS_THICKFRAME	Окно имеет утолщенную рамку, при помощи которой можно изменять размер окна
WS_VISIBLE	Окно становится видимым сразу после создания; этот стиль в основном используется для окон диалога

Примечание

Полосы прокрутки, определяемые как свойства окон, отличаются от полос прокрутки — элементов управления: так, если установить нулевой диапазон полосы прокрутки окна, то Windows автоматически уберет ее! Более подробно полосы прокрутки рассматриваются в главах, посвященных элементам управления.

Помимо перечисленных стилей окон Windows, в файле WINUSER.H определены стили, являющиеся комбинацией наиболее часто используемых стилей.

WS_OVERLAPPEDWINDOW	Комбинация стилей WS_OVERLAPPED, WS_CAPTION, WS_SYSMENU, WS_THICKFRAME, WS_MINIMIZEBOX и WS_MAXIMIZEBOX
WS_POPUPWINDOW	Комбинация стилей WS_POPUP, WS_BORDER и WS_SYSMENU

При работе с окнами часто также используются дополнительные или расширенные биты стиля:

WS_EX_ABSPOSITION	Определяет, что позиция окна задается в абсолютных единицах относительно левого верхнего угла экрана
WS_EX_ACCEPTFILES	Определяет, что окно допускает перетаскивание файлов (drag-and-drop)
WS_EX_CLIENTEDGE	Придает "трехмерный эффект" границе окна, клиентская область при этом как бы вдавлена в окно
WS_EX_CONTEXTHELP	Включает знак вопроса в заголовок окна; когда пользователь нажимает этот знак, курсор принимает вид вопросительного знака; если теперь выбрать какое-либо дочернее окно, то оно получит сообщение WM_HELP

WS_EX_CONTROLPARENT	Позволяет пользователю перемещаться по дочерним окнам родительского окна, используя для этого клавишу табуляции <Tab>
WS_EX_DLGMODALFRAME	Окно имеет удвоенную границу, что при использовании совместно со стилем WS_CAPTION позволяет создавать окно с полосой заголовка
WS_EX_LEFT	Устанавливает для окна свойство выравнивания влево (по умолчанию)
WS_EX_LEFTSCROLLBAR	Если имеется полоса прокрутки, то она располагается с левой стороны рабочей (клиентской) области окна
WS_EX_LTRREADING	Используется по умолчанию и определяет естественный порядок (слева направо) для большинства языков ввода и чтения текста
WS_EX_MDICHILD	Дочернее окно MDI
WS_EX_NOPARENTNOTIFY	Определяет, что дочернее окно, созданное с этим стилем, не посылает сообщение WM_PARENTNOTIFY своему родительскому окну при создании или уничтожении
WS_EX_RIGHT	В зависимости от класса окна устанавливает для него групповое свойство выравнивания вправо
WS_EX_RIGHTSCROLLBAR	Если имеется полоса прокрутки, то она располагается с правой стороны рабочей (клиентской) области окна; используется по умолчанию
WS_EX_RTLREADING	Определяет порядок ввода и чтения текста справа налево
WS_EX_SMCAPTION	Окно имеет уменьшенную высоту полосы заголовка; используется при создании "плавающих" панелей инструментов
WS_EX_STATICEDGE	Окно имеет границы с трехмерным стилем, заданным для использования в элементах, к которым пользователь не получает доступа (статические элементы управления)
WS_EX_TOOLWINDOW	Окно, которое используется для создания "плавающей" панели инструментов; такое окно имеет укороченную полосу заголовка и не отображается на панели задач
WS_EX_TOPMOST	Окно будет располагаться поверх всех окон (созданных без этого стиля); для установки или снятия этого бита можно использовать функцию SetWindowPos
WS_EX_TRANSPARENT	Окно является прозрачным, т. е. не закрывает другие окна, расположенные "под ним"; сообщение WM_PAINT такое окно получает до того, как это сообщение получают все окна того же уровня, но расположенные "под ним"
WS_EX_WINDOWEDGE	Окно имеет приподнятую границу для создания "трехмерного эффекта"

Так же как в случае основных стилей, для расширенных определены два комбинированных дополнительных стиля (WINUSER.H):

WS_EX_OVERLAPPEDWINDOW	Комбинация стилей WS_EX_CLIENTEDGE и WS_EX_WINDOWEDGE
WS_EX_PALETTEWINDOW	Комбинация стилей WS_EX_WINDOWEDGE, WS_EX_SMCAPTION и WS_EX_TOPMOST

После того как мы перечислили все возможные стили окон, осталось сказать несколько слов об упомянутых выше основных типах окон Windows. Напомним, что их всего три:

- *Перекрывающиеся окна.* Это основной, наиболее универсальный, тип окон Windows. Стиль WS_OVERLAPPEDWINDOW чаще всего используется для их создания. Главное окно приложения, как правило, имеет именно этот тип.
- *Вспомогательные или всплывающие окна.* Чаще всего этот тип окон создается с использованием стиля WS_POPUP. Обычно они отображают какую-либо информацию короткого промежутка времени. Наиболее часто окна этого типа используются для отображения диалоговых окон или окон сообщения. Основное их отличие от других окон заключается в том, что, даже если они имеют родительское окно, то все равно всегда отображаются поверх всех окон на экране, выскакивая, как поплавки, наверх даже тогда, когда пользователь делает активным другое окно. Для окон этого типа, как правило, организуется своя оконная процедура. Они могут иметь, а могут и не иметь родителя. Если такое окно не имеет родительского, то оно является совершенно независимым от создавшего его окна и по своим свойствам практически неотличимо от перекрывающихся окон. Поведение всплывающего окна, имеющего родителя, зависит от того, что происходит с последним. Когда главное окно свертывается, всплывающее окно "без родителя" скрывается, а "с родителем" остается на экране сверху. Подчеркнем еще один важный момент: когда мы говорим, что вспомогательное окно *имеет родителя*, это совсем не означает, что оно является *дочерним*.

Примечание

Наиболее полным семантическим эквивалентом английского термина "popup window" является термин "окно-поплавок". Мы будем называть такие окна *всплывающими*.

- *Дочерние окна.* Окна этого типа создаются тогда, когда у приложения уже есть главное окно. Такие окна связаны некоторыми характеристиками (как бы подчинены) с тем окном, из которого они были созданы, отсюда и название. Назначение этих окон может быть самым разнообразным, начиная от простого деления родительского окна на области, до организации многодокументного интерфейса. Все элементы управления также

являются дочерними окнами. Из основных свойств этого типа окон отметим следующие. Дочерние окна никогда не отображаются вне своего родительского окна ни в раскрытом виде, ни в виде значка — они как бы целиком принадлежат родителю. Располагаются они в родительском окне относительно верхнего левого угла его рабочей (клиентской) области. Более того, при перемещении родительского окна по экрану его дочерние окна перемещаются вместе с ним. И, наконец, дочернее окно никогда не может стать активным.

Примечание

Если вы создаете окно-элемент управления, то вместе со стилем WS_CHILD можно указать дополнительные флаги, характерные для того или иного элемента управления. С конкретными значениями вы познакомитесь при обсуждении соответствующих элементов управления.

Какие же возможности предоставляет нам библиотека MFC для создания и работы с окнами? Если обратимся к иерархии классов MFC (рис. 9.2), то вспомним, что все классы, имеющие какое-либо отношение к окнам, являются производными (прямо или косвенно) от единственного класса CWnd.

Рис. 9.2. Иерархия оконных классов

Этот класс, а также специальные производные от него классы, включая CFrameWnd, CMDIFrameWnd, CMDIChildWnd, CView и CDialog, имеющиеся в библиотеке, спроектированы таким образом, чтобы сделать за нас большую часть обязательной, но рутинной работы с различными типами окон Windows. Вначале рассмотрим возможности базового класса.

Класс CWnd

Класс CWnd — это базовый класс для всех окон, созданных на базе библиотеки MFC. Помимо предоставления всем своим производным классам большого числа функций для работы с окнами, этот класс служит для создания разнообразных дочерних окон. Начнем со знакомства с теми возможностями, которые заложены в него разработчиками.

Из имеющихся членов этого класса для нас, на данном этапе знакомства, интерес представляет один:

HWND CWnd::**m_hWnd** —

дескриптор окна Windows, присоединенного к объекту класса *CWnd*. Это достаточно важный параметр, особенно для главного окна приложения, т. к. он позволяет библиотеке взять на себя всю работу для корректного завершения приложения, когда пользователь закрывает его главное окно.

Класс CWnd содержит практически все функции, необходимые для работы с окнами. Таких функций более двухсот, их можно разбить на несколько категорий, каждая из которых включает функции, направленные на решение определенного круга задач. Приведем список категорий с краткой характеристикой входящих в них функций:

- *Конструктор*. Создает объект класса без создания окна Windows.
- *Инициализация*. В эту категорию входят функции: отвечающие за создание окна Windows (соответствующей ему внутренней структуры) и присоединение его к объекту класса, — Create и CreateEx; виртуальная функция PreCreateWindow, позволяющая зарегистрировать используемый по умолчанию класс окна "AfxWnd"; функции, позволяющие узнать текущий обычный или расширенный стиль окна — GetStyle и GetExStyle; функции, позволяющие присоединить или отсоединить окно Windows от объекта класса CWnd — Attach и Detach; функции для получения дескриптора окна Windows (GetSafeWnd) или указателя на объект по дескриптору (FromHandle и FromHandlePermanent); функции для работы с подклассами окна — PreSubclassWindow и SubclassWindow; а также некоторые другие функции.
- *Состояние окна*. Сюда входят функции, позволяющие: определить (IsWindowEnable) или изменить (EnableWindow) доступ к окну; узнать или назначить активное окно (обычное — GetActiveWnd и SetActiveWnd, окно переднего плана — GetForegroundWindow и SetForegroundWindow); получить или установить фокус ввода с клавиатуры (GetFocus и SetFocus) или от мыши (GetCapture и SetCapture), а также дескриптор значка, ассоциированного с окном (GetIcon и SetIcon); модифицировать расширенный или обычный стили окна (ModifyStyle и ModifyStyleEx); получить указатель на объект "рабочего стола" Windows.
- *Изменение размеров и позиции окна*. Включает две группы функций. Функции, служащие для определения или установки состояния окна: видимое/невидимое (GetWindowPlacement и SetWindowPlacement); свернуто (IsIconic) или развернуто (IsZoomed). И, собственно, функции, позволяющие узнать или изменить размеры и/или позицию окна: MoveWindow — перемещение окна с изменением размеров; SetWindowPos — перемещение окна с изменением размеров и расположения относительно других окон; ArrangeIconicWindows — упорядочить все свернутые дочер-

ние окна; BringWindowToTop — делает окно старшим в Z-порядке; GetWindowRect — определяет размеры всего окна; GetClientRect — определяет размеры рабочей области окна.

- *Доступ к окну.* Из функций этой категории мы пока хотели бы отметить следующие: FindWindow — позволяет определить, существует ли в системе окно, заданное именем класса и именем окна (оба параметра являются необязательными), т. е. запущено ли приложение, имеющее искомое окно; CenterWindow — центрирует положение окна относительно его родителя. Кроме представленных, категория содержит еще большую группу функций, которые используются значительно реже и поэтому будут рассмотрены не все.
- *Обновление-перерисовка.* Из функций этой категории мы представим только наиболее часто используемые: UpdateWindow — обновляет рабочую область, посылая сообщение WM_PAINT; Invalidate — задает область модификации для последующей перерисовки при обработке ближайшего сообщения WM_PAINT; ShowWindow — отображает окно на экране, режим отображения задается передаваемым параметром. Кроме вышеперечисленных, сюда же можно отнести группу функций для работы с контекстами устройств: BeginPaint, EndPaint, Print, PrintClient, GetDC, ReleaseDC и GetWindowDC, которые подробно будут описаны при обсуждении вывода в окна графической информации.
- *Преобразование координат.* Служат для взаимного преобразования рабочих и экранных координат текущего окна.
- *Работа с текстом.* В эту категорию входят функции, которые позволяют установить или получить текст, ассоциированный с окном (SetWindowText и GetWindowText) и его длину (GetWindowTextLenght), а также используемый шрифт (SetFont и GetFont).
- *Работа с сообщениями.* Функции этой категории будут рассмотрены более подробно, т. к. они важны для понимания принципов работы всей системы Windows и ваших приложений в частности.

Оставшиеся категории функций достаточно специфичны и будут обсуждаться в соответствующих разделах книги: функции для работы с полосами прокрутки, каретками (текстовыми курсорами), элементами блока диалога, меню, таймерами, всплывающими подсказками, а также функции для работы с буфером обмена и обработчики сообщений.

Класс CWnd служит для выполнения двух основных целей: предоставление интерфейса всем другим оконным классам библиотеки MFC и создание дочерних окон Windows. В связи с этим мы ограничились здесь простым перечислением основных реализованных в классе функций. Многие из них мы рассмотрим при обсуждении конкретных примеров программ.

Класс CFrameWnd

В отличие от своего базового класса, CFrameWnd служит для создания перекрывающихся или всплывающих окон и поддерживает однодокументный интерфейс Windows (SDI). С этим классом (и производными от него) тесно связано понятие *фрейма* (frame window). Под ним понимается окно, которое координирует взаимодействие приложения с документом и его представлением. Оно отражается на экране в виде рамки, ограничивающей представление, с необязательной строкой состояния и стандартными элементами управления окном. Это окно отвечает за управление размещением своих дочерних окон и других элементов рабочей области. Кроме того, фрейм переадресует команды своим представлениям и может отвечать на извещения от элементов управления окна.

Примечание

Ввиду широкого использования архитектуры "документ/представление" (document/view) мы посвятили ей отдельную часть книги.

Этот класс спроектирован таким образом, чтобы взять на себя задачу выполнения всех основных функций приложения Windows на базе интерфейса SDI:

- Передает команды и многие общие извещения в активное текущее представление
- Управляет размещением панелей элементов управления, представлений и других дочерних окон внутри своей рабочей области. Во время цикла ожидания обновляет панель инструментов и кнопки панелей элементов управления. Кроме того, имеется обработка команд по умолчанию для показа или скрытия панели инструментов и строки состояния
- Управляет полосой основного меню. Когда изображается раскрывающееся меню, фрейм использует механизм UPDATE_COMMAND_UI для установки элементов меню в состояние "доступен", "заблокирован" или "помечен", а также для добавления и удаления элементов. Когда пользователь выбирает элемент меню, фрейм обновляет строку состояния, выводя туда соответствующее заданное сообщение
- Может использовать специальную *таблицу назначений акселераторов* (accelerator table), которая автоматически транслирует нажатие комбинации клавиш в команду
- Имеет необязательный идентификатор справки, который используется для контекстно-зависимой справочной системы. Управляет самим механизмом контекстно-зависимой справочной системы (вызов комбинацией

клавиш <Shift>+<F1>) и режимом предварительного просмотра перед печатью

- Поддерживает выполнение операций drag-and-drop для работы файлами. В ответ на перетаскивание файлов зарегистрированных и ассоциированных с приложением типов он требует открыть канал динамического обмена данными (DDE)
- Если фрейм является главным окном приложения (определяется значением CWinThread::m_pMainWnd), то когда пользователь закрывает приложение, фрейм предлагает ему сохранить измененные документы

В классе определены две основных функции-члена:

BOOL CFrameWnd::**m_bAutoMenuEnable** —

ее значение, равное FALSE, приведет к автоматической блокировке тех элементов меню (и соответствующих кнопок панели инструментов), для которых не определены обработчики ON_UPDATE_COMMAND_UI или ON_COMMAND. По умолчанию устанавливается в TRUE.

static AFX_DATA const CRect CFrameWnd::**rectDefault** —

определяет константный объект класса CRect, который предписывает Windows самостоятельно определить размеры и положение создаваемого окна и определен следующим образом:

```
const AFX_DATADEF CRect CFrameWnd::rectDefault(
  CW_USEDEFAULT,
  CW_USEDEFAULT,
  0,
  0
);
```

Функции-члены этого класса сгруппированы по следующим категориям:

- *Конструктор.* Как обычно, конструктор создает объект класса, но не создает окно.
- *Инициализация.* Сюда входят: две функции, отвечающие за создание окна Windows и присоединение его к текущему объекту класса — Create и LoadFrame; функция, загружающая таблицу акселераторов — LoadAccelTable; группа функций для управления инициализацией панелей элементов управления (LoadBarState — загружает сохраненные в INI-файле или реестре установки панели, SaveBarState — сохраняет текущие установки панели в INI-файле или в реестре, SetDockState — сохраняет размещение текущего окна относительно главного, GetDockState — восстанавливает размещение текущего окна относительно главного (для плавающих панелей) и ShowControlBar — управляет режимом отображения панели элементов управления).

- *Операции.* В эту категорию входит большое число функций для работы с документами и их представлениями, с панелями элементов управления, строкой состояния и для управления модальностью фрейма окна.

Помимо своего основного назначения — поддержки SDI-приложений, — этот класс служит базовым для классов CMDIFrameWnd и CMDIChildWnd, которые созданы специально для организации приложений на базе MDI.

Еще раз подчеркнем важный момент. На базе этого класса легко и удобно создавать фреймы окон. Для того чтобы существенно расширить эти возможности, в библиотеке реализованы специальные классы, базирующиеся на понятии "представление".

Создание окна и библиотека MFC

Пора подробнее рассмотреть процесс создания окон. Рассмотрим приложение-пример Styles, показывающее 10 дочерних окон с различными комбинациями оконных стилей в каждом. Если щелкнуть левой кнопкой мыши в рабочей области окна, на экран выводится окно информационного сообщения, где перечислены стили, с которыми создано окно.

Мы приводим полностью все файлы, содержащие код этого приложения. Основа (каркас) приложения создавалась с использованием средств AppWizard и ClassWizard, которые кроме самого кода добавили и комментарии. Комментарии приведены без изменений, естественно, на английском языке. После исходного текста приведен его подробный анализ. На рис. 9.3 представлен внешний вид всех окон приложения.

Рис. 9.3. Различные типы окон приложения Styles

```
/////////////////////////////////////////////////////////////////////////////
// Файл Styles.h
// Copyright (c) 1999 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////////////////////////
// main header file for the STYLES application
#ifndef __AFXWIN_H__
  #error include 'stdafx.h' before including this file for PCH
#endif
#include "resource.h"       // main symbols
/////////////////////////////////////////////////////////////////////////////
// CStylesApp:
class CStylesApp : public CWinApp
{
public:
  CStylesApp();
// Overrides
  // ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CStylesApp)
  public:
  virtual BOOL InitInstance();
  //}}AFX_VIRTUAL
  DECLARE_MESSAGE_MAP()
};
/////////////////////////////////////////////////////////////////////////////
// Файл Styles.cpp
// Copyright (c) 1999 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////////////////////
// Defines the class behaviors for the
// application.
#include "stdafx.h"
#include "Styles.h"
#include "MainFrm.h"
/////////////////////////////////////////////////////////////////////////////
// CStylesApp
BEGIN_MESSAGE_MAP(CStylesApp, CWinApp)
  //{{AFX_MSG_MAP(CStylesApp)
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

```
/////////////////////////////////////////////////////////////////////////////
// CStylesApp construction
CStylesApp::CStylesApp()
{
}
/////////////////////////////////////////////////////////////////////////////
// The one and only CStylesApp object
CStylesApp theApp;
/////////////////////////////////////////////////////////////////////////////
// CStylesApp initialization
BOOL CStylesApp::InitInstance()
{
  CMainFrame *pMainWnd = new CMainFrame;
  m_pMainWnd = pMainWnd;
  pMainWnd->LoadFrame(IDR_MAINFRAME);
  int cx = ::GetSystemMetrics(SM_CXSCREEN)/8;
  int cy = ::GetSystemMetrics(SM_CYSCREEN)/4;
  pMainWnd->SetWindowPos(
    NULL,
    cx,
    cy,
    6*cx,
    2*cy,
    SWP_NOZORDER | SWP_SHOWWINDOW);
  return TRUE;
}
/////////////////////////////////////////////////////////////////////////////
// CStylesApp commands
/////////////////////////////////////////////////////////////////////////////
// Файл MainFrm.h
// Copyright (c) 1999 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////////////////////////
// interface of the CMainFrame class
/////////////////////////////////////////////////////////////////////////////
#include "ChildWindows.h"
class CMainFrame : public CFrameWnd
{
public:
  CMainFrame();
```

```
// Attributes
public:
  CChildWnd *apChildWnd[5];
  CPopupWnd *apPopupWnd[5];
// Overrides
  // ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CMainFrame)
  virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
  //}}AFX_VIRTUAL
// Implementation
public:
  virtual ~CMainFrame();
// Generated message map functions
protected:
  //{{AFX_MSG(CMainFrame)
  afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
  afx_msg void OnGetMinMaxInfo(MINMAXINFO FAR* lpMMI);
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()
};
/////////////////////////////////////////////////////////////////////////////
// Файл MainFrm.cpp
// Copyright (c) 1999 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////////////////////////
// implementation of the CMainFrame class
//
#include "stdafx.h"
#include "Styles.h"
#include "MainFrm.h"
/////////////////////////////////////////////////////////////////////////////
// CMainFrame
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
  //{{AFX_MSG_MAP(CMainFrame)
  ON_WM_CREATE()
  ON_WM_GETMINMAXINFO()
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()
/////////////////////////////////////////////////////////////////////////////
// CMainFrame construction/destruction
```

```
extern char szNameWnd[128];
CMainFrame::CMainFrame()
{
  for(int i = 0; i < 5; i++)
  {
    apChildWnd[i] = NULL;
    apPopupWnd[i] = NULL;
  }
}
CMainFrame::~CMainFrame()
{
  for(int i = 0; i < 5; i++)
  {
    if(apChildWnd[i] != NULL)
    {
      delete apChildWnd[i];
    }
  }
}
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
  cs.lpszName = "Главное окно приложения \"Стили"";
  return CFrameWnd::PreCreateWindow(cs);
}
/////////////////////////////////////////////////////////////
// CMainFrame message handlers
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
  if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
    return -1;
  int i;
  for(i = 0; i < 5; i++)
  {
    apChildWnd[i] = new CChildWnd;
    apChildWnd[i]->Create(NULL,
                          NULL, WS_VISIBLE,
                          CRect(0, 0, 0, 0), this, (WORD)i);
    ::SetWindowLong(apChildWnd[i]->GetSafeHwnd(),
                    GWL_USERDATA, i);
```

```
  }
  for(i = 0; i < 4; i++)
  {
    apPopupWnd[i] = new CPopupWnd;
    apPopupWnd[i]->Create(NULL, NULL, 0,
                          rectDefault, this);
    ::SetWindowLong(apPopupWnd[i]->GetSafeHwnd(),
                    GWL_USERDATA, i);
    sprintf(szNameWnd, "Окно %d", i);
    apPopupWnd[i]->SetWindowText(szNameWnd);
  }
  apPopupWnd[4] = new CPopupWnd;
  apPopupWnd[4]->Create(NULL, NULL, 0, rectDefault);
  ::SetWindowLong(apPopupWnd[4]->GetSafeHwnd(),
    GWL_USERDATA, 4);
  sprintf(szNameWnd, "Окно %d", 4);
  apPopupWnd[i]->SetWindowText(szNameWnd);
  ::SetClassLong(apChildWnd[0]->GetSafeHwnd(), GCL_HCURSOR,
    (LONG)AfxGetApp()->LoadStandardCursor(IDC_IBEAM));
  ::SetClassLong(apPopupWnd[0]->GetSafeHwnd(), GCL_HCURSOR,
    (LONG)AfxGetApp()->LoadStandardCursor(IDC_UPARROW));
  HBRUSH hNewBrush = ::CreateSolidBrush(RGB(255, 255, 220));
  ::SetClassLong(apPopupWnd[3]->GetSafeHwnd(),
    GCL_HBRBACKGROUND, (LONG)hNewBrush);
  for(i = 0; i < 5; i++)
  {
    apPopupWnd[i]->ShowWindow(SW_SHOWNORMAL);
    apPopupWnd[i]->UpdateWindow();
  }
  return 0;
}
void CMainFrame::OnGetMinMaxInfo(MINMAXINFO FAR* lpMMI)
{
  int cx = ::GetSystemMetrics(SM_CXSCREEN)/8;
  int cy = ::GetSystemMetrics(SM_CYSCREEN)/4;
  lpMMI->ptMinTrackSize.x = 6*cx;
  lpMMI->ptMinTrackSize.y = 2*cy;
  lpMMI->ptMaxTrackSize.x = 6*cx;
  lpMMI->ptMaxTrackSize.y = 2*cy;
```

```
  lpMMI->ptMaxPosition.x  = cx;
  lpMMI->ptMaxPosition.y  = cy;
  CFrameWnd::OnGetMinMaxInfo(lpMMI);
}
/////////////////////////////////////////////////////////////
// Файл ChildWindows.h
// Copyright (c) 1999 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////////
// header file
//
/////////////////////////////////////////////////////////////
// CChildWnd window
class CChildWnd : public CWnd
{
// Construction
public:
  CChildWnd();
// Overrides
  // ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CChildWnd)
  protected:
  virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
  //}}AFX_VIRTUAL
// Implementation
public:
  virtual ~CChildWnd();
// Generated message map functions
protected:
  //{{AFX_MSG(CChildWnd)
  afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
  afx_msg void OnLButtonUp(UINT nFlags, CPoint point);
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()
};
/////////////////////////////////////////////////////////////
// CPopupWnd frame
class CPopupWnd : public CFrameWnd
{
// Construction
```

```
public:
  CPopupWnd();
// Implementation
public:
  virtual ~CPopupWnd();
// Generated message map functions
  //{{AFX_MSG(CPopupWnd)
  afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
  afx_msg void OnLButtonUp(UINT nFlags, CPoint point);
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()
};
/////////////////////////////////////////////////////////////
// Файл ChildWindows.cpp
// Copyright (c) 1999 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////////
// implementation file
//
#include "stdafx.h"
#include "Styles.h"
#include "ChildWindows.h"
int nCount = 0;
DWORD adwStyles[] =
{
  WS_CAPTION | WS_CLIPSIBLINGS,
  WS_CAPTION | WS_SYSMENU | WS_HSCROLL | WS_VSCROLL |
    WS_MINIMIZEBOX | WS_MAXIMIZEBOX | WS_CLIPSIBLINGS,
  WS_THICKFRAME | WS_CLIPSIBLINGS,
  WS_CAPTION | WS_SYSMENU | WS_MINIMIZEBOX | WS_MAXIMIZEBOX |
    WS_CLIPSIBLINGS,
  WS_CAPTION | WS_SYSMENU | WS_HSCROLL | WS_VSCROLL |
    WS_CLIPSIBLINGS,
  WS_POPUP,
  WS_POPUP | WS_CAPTION | WS_THICKFRAME,
  WS_POPUP | WS_CAPTION | WS_SYSMENU |
    WS_MINIMIZEBOX | WS_MAXIMIZEBOX,
  WS_POPUP | WS_CAPTION | WS_SYSMENU |
    WS_HSCROLL | WS_VSCROLL |WS_DISABLED,
  WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU |
```

```
    WS_HSCROLL | WS_VSCROLL
};
char *aszStyles[] =
{
  "WS_CAPTION | WS_CLIPSIBLINGS",
  "WS_CAPTION | WS_SYSMENU | WS_HSCROLL | WS_VSCROLL |\
    WS_MINIMIZEBOX | WS_MAXIMIZEBOX | WS_CLIPSIBLINGS",
  "WS_THICKFRAME | WS_CLIPSIBLINGS",
  "WS_CAPTION | WS_SYSMENU | WS_MINIMIZEBOX |\
    WS_MAXIMIZEBOX | WS_CLIPSIBLINGS",
  "WS_CAPTION | WS_SYSMENU | WS_HSCROLL |\
    WS_VSCROLL | WS_CLIPSIBLINGS",
  "WS_POPUP",
  "WS_POPUP | WS_CAPTION | WS_THICKFRAME",
  "WS_POPUP | WS_CAPTION | WS_SYSMENU |\
    WS_MINIMIZEBOX | WS_MAXIMIZEBOX",
  "WS_POPUP | WS_CAPTION | WS_SYSMENU |\
    WS_HSCROLL | WS_VSCROLL",
  "WS_OVERLAPPPED | WS_CAPTION | WS_SYSMENU |\
    WS_HSCROLL | WS_VSCROLL"
};
char szNameWnd[128];
/////////////////////////////////////////////////////////////////////////////
// CChildWnd
CChildWnd::CChildWnd(){
}
CChildWnd::~CChildWnd(){
}
BEGIN_MESSAGE_MAP(CChildWnd, CWnd)
  //{{AFX_MSG_MAP(CChildWnd)
  ON_WM_CREATE()
  ON_WM_LBUTTONUP()
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()
/////////////////////////////////////////////////////////////////////////////
// CChildWnd message handlers
BOOL CChildWnd::PreCreateWindow(CREATESTRUCT& cs)
{
  cs.style |= adwStyles[nCount];
```

```
  sprintf(szNameWnd, "Окно %d", nCount);
  cs.lpszName = szNameWnd;
  return CWnd::PreCreateWindow(cs);
}
int CChildWnd::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
  if (CWnd::OnCreate(lpCreateStruct) == -1)
    return -1;
  MoveWindow(nCount++ * 118, 24, 116, 96);
  return 0;
}
void CChildWnd::OnLButtonUp(UINT nFlags, CPoint point)
{
  WORD nWnd = (WORD)::GetWindowLong(this->GetSafeHwnd(),
    GWL_USERDATA);
  sprintf(szNameWnd, "Дочернее окно: %d", nWnd);
  MessageBox(aszStyles[nWnd], szNameWnd);
  CWnd::OnLButtonUp(nFlags, point);
}
/////////////////////////////////////////////////////////////////////////////
// CPopupWnd
CPopupWnd::CPopupWnd()
{
}
CPopupWnd::~CPopupWnd()
{
}
BEGIN_MESSAGE_MAP(CPopupWnd, CFrameWnd)
  //{{AFX_MSG_MAP(CPopupWnd)
  ON_WM_CREATE()
  ON_WM_LBUTTONUP()
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()
/////////////////////////////////////////////////////////////////////////////
// CPopupWnd message handlers
int CPopupWnd::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
  if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
    return -1;
```

```
  ModifyStyle((DWORD)-1, adwStyles[nCount]);
  int n = nCount++ — 5;
  MoveWindow(n * 144, 24, 136, 120);
  return 0;
}
void CPopupWnd::OnLButtonUp(UINT nFlags, CPoint point)
{
  WORD nWnd = (WORD)::GetWindowLong(this->GetSafeHwnd(),
    GWL_USERDATA);
  sprintf(szNameWnd, "Всплывающее окно: %d", nWnd);
  MessageBox(aszStyles[nWnd + 5], szNameWnd);
  CFrameWnd::OnLButtonUp(nFlags, point);
}
/////////////////////////////////////////////////////////////
// Файл stdafx.cpp
// Copyright (c) 1999 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////////
// source file that includes just the standard
// includes
// Styles.pch will be the pre-compiled header
// stdafx.obj will contain the pre-compiled type information
#include "stdafx.h"
/////////////////////////////////////////////////////////////
// Файл stdafx.h
// Copyright (c) 1999 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////////
// include file for standard system include files,
// or project specific include files that are used
// frequently, but are changed infrequently
//
#define VC_EXTRALEAN      // Exclude rarely-used stuff from
                          // Windows headers
#include <afxwin.h>       // MFC core and standard components
/////////////////////////////////////////////////////////////
// Файл Styles.rc
// Copyright (c) 1999 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////////
//Microsoft Developer Studio generated resource script.
//
```

```
#include "resource.h"
#define APSTUDIO_READONLY_SYMBOLS
/////////////////////////////////////////////////////////////////////////////
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"
/////////////////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS
/////////////////////////////////////////////////////////////////////////////
// Russian resources
#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_RUS)
#ifdef _WIN32
  LANGUAGE LANG_RUSSIAN, SUBLANG_DEFAULT
#pragma code_page(1251)
#endif //_WIN32
#ifdef APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////////////////////
// TEXTINCLUDE
1 TEXTINCLUDE DISCARDABLE
BEGIN
  "resource.h\0"
END
2 TEXTINCLUDE DISCARDABLE
BEGIN
  "#include ""afxres.h""\r\n"
  "\0"
END
3 TEXTINCLUDE DISCARDABLE
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 9, 1\r\n"
  "#pragma code_page(1252)\r\n"
  "#endif\r\n"
```

```
  "#include ""res\\Styles.rc2""  // non-Microsoft Visual C++
                                 //edited resources\r\n"
  "#include ""afxres.rc""        // Standard components\r\n"
  "#endif\0"
END
#endif    // APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////
// Menu
IDR_MAINFRAME MENU PRELOAD DISCARDABLE
BEGIN
  POPUP "&File"
  BEGIN
   MENUITEM "E&xit",                        ID_APP_EXIT
  END
END
#ifndef _MAC
/////////////////////////////////////////////////////////////
// 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", "STYLES MFC Application\0"
      VALUE "FileVersion", "1, 0, 0, 1\0"
      VALUE "InternalName", "STYLES\0"
```

```
      VALUE "LegalCopyright", "Copyright © 1996\0"
      VALUE "LegalTrademarks", "\0"
      VALUE "OriginalFilename", "STYLES.EXE\0"
      VALUE "ProductName", "STYLES Application\0"
      VALUE "ProductVersion", "1, 0, 0, 1\0"
    END
  END
  BLOCK "VarFileInfo"
  BEGIN
    VALUE "Translation", 0x409, 1200
  END
END
#endif    // !_MAC
/////////////////////////////////////////////////////////////////////////////
// Icon
IDR_MAINFRAME    ICON    DISCARDABLE     "res\\Styles.ico"
/////////////////////////////////////////////////////////////////////////////
// String Table
STRINGTABLE PRELOAD DISCARDABLE
BEGIN
  IDR_MAINFRAME "Styles\n\nStyles\n\n\nStyles.Doc\nStyles"
END
STRINGTABLE PRELOAD DISCARDABLE
BEGIN
  AFX_IDS_APP_TITLE      "Приложение "Стили""
  AFX_IDS_IDLEMESSAGE    "Готовность"
END
STRINGTABLE DISCARDABLE
BEGIN
  ID_APP_EXIT            "Завершение работы приложения\nВыход"
END
#endif    // Russian resources
/////////////////////////////////////////////////////////////////////////////
#ifndef APSTUDIO_INVOKED
// Generated from the TEXTINCLUDE 3 resource.
//
#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 9, 1
#pragma code_page(1252)
#endif
#include "res\Styles.rc2"  // non-Microsoft Visual C++
                           // edited resources
#include "afxres.rc"       // Standard components
#endif
/////////////////////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

Создание главного окна SDI-приложения

В рассмотренной ранее программе First мы образовали класс нашего окна из базового CWnd. Это допустимый, но очень неэффективный путь. Пришло время познакомить вас с одним из возможных подходов для создания главного окна SDI-приложения.

Примечание

Второй, еще более эффективный подход будет представлен при знакомстве с архитектурой "документ/представление".

Прежде всего обратим внимание на базовый класс нашего главного окна:

```
class CMainFrame : public CFrameWnd
{
...
};
```

Почему CFrameWnd? Дело в том, что этот класс спроектирован таким образом, чтобы предоставить максимум удобств для работы с перекрывающимися и всплывающими окнами. А, как вы помните, в качестве главного окна приложения наиболее приемлемым является именно перекрывающееся окно. Рассмотрим теперь подробно функцию CStylesApp::InitInsnance где, собственно, и начинается создание главного окна:

```
BOOL CStylesApp::InitInstance()
{
  CMainFrame *pMainWnd = new CMainFrame;
  m_pMainWnd = pMainWnd;
```

```
  pMainWnd->LoadFrame(IDR_MAINFRAME);
  int cx = ::GetSystemMetrics(SM_CXSCREEN)/8;
  int cy = ::GetSystemMetrics(SM_CYSCREEN)/4;
  pMainWnd->SetWindowPos(NULL,
    cx,
    cy,
    6*cx,
    2*cy,
    SWP_NOZORDER | SWP_SHOWWINDOW);
  return TRUE;
}
```

Процесс создания окна состоит из трех шагов:

1. Создание объекта "главное окно":

   ```
   CMainFrame *pMainWnd = new CMainFrame;
   ```

2. Вызов специальной функции, которая создает окно Windows и присоединяет его к нашему объекту:

   ```
   pMainWnd->LoadFrame(IDR_MAINFRAME);
   ```

3. Отображение окна на экране, чтобы пользователь мог с ним взаимодействовать:

   ```
   pMainWnd->SetWindowPos(NULL, cx, cy, 6*cx, 2*cy,
                          SWP_NOZORDER | SWP_SHOWWINDOW);
   ```

Рассмотрим сначала процесс создания окна Windows, или, используя терминологию MFC, фрейма окна.

Из двух функций, реализованных в классе CFrameWnd, мы остановили свой выбор на виртуальной функции CFrameWnd::LoadFrame, которая выполняет регистрацию оконного класса, используя для этого предопределенный шаблон, и извлекает из файла ресурсов, задаваемого идентификатором, остальные необходимые для создания окна Windows параметры:

virtual BOOL CFrameWnd::**LoadFrame**(
UINT nIDResource,
DWORD dwDefaultStyle = WS_OVERLAPPEDWINDOW |
FWS_ADDTOTITLE,
CWnd *pParentWnd = NULL,
CCreateContext *pContext = NULL) —

возвращает значение TRUE, если создание фрейма прошло успешно, и FALSE в противном случае. Функция имеет параметры: nIDResource — идентификатор разделяемых ресурсов, ассоциированных с фреймом, который определяет меню, таблицу акселераторов, значок и строку, помещаемую в заголовок окна; pParentWnd — указатель на родительское окно данного фрейма (необязательный параметр); pContext — указатель на структу-

ру CCreateContex, которая определяет объекты, связанные с данным фреймом; dwDefaultStyle — стиль фрейма (необязательный параметр), дополнительно к стандартным могут быть использованы следующие стили:

FWS_ADDTOTITLE	В конец заголовка окна будет добавлена дополнительная текстовая строка
FWS_PREFIXTITLE	Перед заголовком окна будет помещено имя документа
FWS_SNAPTOBAR	Позволяет подогнать размеры окна под размеры панели элементов управления

Основное назначение этой функции — максимальное упрощение процесса создания фрейма окна.

Примечание

При использовании функции CFrameWnd::LoadFrame единственным обязательным ресурсом является меню. Если его не определить, то программа будет выдавать сообщение об ошибке.

Используя идентификатор ресурсов, функция извлекает из них необходимые параметры и ... вызывает функцию CFrameWnd::Create, к рассмотрению которой мы и переходим:

```
BOOL CFrameWnd::Create(
    LPCTSTR lpszClassName,
    LPCTSTR lpszWindowName,
    DWORD dwStyle = WS_OVERLAPPEDWINDOW,
    const RECT &rect = rectDefault,
    CWnd *pParentWnd = NULL,
    LPCTSTR lpszMenuName = NULL,
    DWORD dwExStyle = 0,
    CCreateContext *pContext = NULL) —
```

возвращает значение TRUE, если создание прошло успешно, и FALSE в противном случае. Многие из параметров, передаваемых в функцию, вам уже должны быть хорошо знакомы, поэтому мы их только перечислим, рассматривая только те, которые не совпадают с уже описанными для функции CFrameWnd::LoadFrame: lpszClassName — адрес символьной строки, содержащей зарегистрированное имя класса, если этот параметр равен NULL, то в качестве шаблона используются атрибуты предопределенного класса "AfxFrameOrView"; lpszWindowName — адрес символьной строки, содержащей имя конкретного окна, этот же текст помещается в заголовок окна; dwStyle — стиль окна (необязательный параметр); rect — ссылка на структуру, содержащую размеры и координаты окна относительно его родителя (необязательный параметр); lpszMenuName — адрес символьной строки, содержащей имя ресурса меню, используемого в окне, если меню задано числовым идентификатором, то для получения строки следует использовать макрос Windows MAKEINTRESOURCE (необязательный параметр); dwExStyle — определяет атрибуты расширенного стиля (необязательный параметр).

Функция загружает ресурс меню и вызывает функцию CWnd::CreateEx, которая регистрирует имя оконного класса, заполняя структуру WNDCLASS, и

вызывает виртуальную функцию PreCreateWindow. В процессе выполнения функции Windows посылает в окно сообщения WM_NCCREATE, WM_NCCALKSIZE, WM_CREATE и WM_GETMINMAXINFO. Если при создании окна был определен стиль WS_VISIBLE, то в окно дополнительно посылаются все сообщения, необходимые для активизации и отображения окна на экране. Таким образом, переопределяя функцию PreCreateWindow и обработчик любого из представленных сообщений Windows, можно изменять параметры оконного класса и собственно окна, как до, так и после создания окна Windows.

Рассмотрим, как этот механизм реализован в нашей программе. Начнем с функции CMainFrame::PreCreateWindow:

```
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
  cs.lpszName = "Главное окно приложения \"Стили\"";
  return CFrameWnd::PreCreateWindow(cs);
}
```

Единственный параметр, передаваемый в эту функцию, — ссылка на структуру CREATESTRUCT, которая определяет параметры инициализации, посылаемые в оконную процедуру, и содержит следующие поля:

```
typedef struct tagCREATESTRUCT{
  LPVOID lpCreateParams;
  HANDLE hInctance;
  HMENU hMenu;
  HWND hwndParent;
  int cy;
  int cx;
  int y;
  int x;
  LONG style;
  LPCSTR lpszName;
  LPCSTR lpszClass;
  DWORD dwExStyle;
} CREATESTRUCT;
```

Назначение полей этой структуры: lpCreateParams — указатель на данные, используемые при создании окна; hInstance — дескриптор модуля, который владеет окном; hMenu — идентификатор меню, используемого окном; hwndParent — дескриптор окна-родителя, если этот параметр равен NULL, то новое окно является окном верхнего уровня; cy, cx, y, x — целые числа, задающие соответственно ширину и высоту окна, а также его положение относительно своего владельца; lpszName — символьная строка, задающая имя окна; lpszClass — символьная строка, содержащая имя класса, которому принадлежит окно; dwExStyle — определяет атрибуты расширенного стиля окна.

Мы использовали эту функцию, чтобы не заниматься процедурой регистрации класса окна, возложив ее выполнение на библиотеку, поскольку принятые по умолчанию установки нас вполне устраивают. Единственное, что нас не устраивает, это заголовок окна, который мы и переопределили:

```
cs.lpszName = "Главное окно приложения \"Стили\"";
```

Это один из возможных способов присвоения имени окну.

Примечание

Не забывайте вызывать функцию базового класса CFrameWnd::PreCreateWindow для проведения правильных инициализирующих действий.

После того как окно Windows создано и присоединено к объекту фрейма окна, в ответ на сообщение WM_CREATE вызывается обработчик сообщения OnCreate (механизм вызова был подробно описан при обсуждении механизма работы с сообщениями в библиотеке MFC):

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
  if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
    return -1;
  return 0;
}
```

Эта функция — одно из возможных мест, где создаются и присоединяются к окну произвольные элементы пользовательского интерфейса. Для правильного завершения процесса создания окна не забывайте вызывать перед какими-либо своими действиями обработчик базового класса CFrameWnd::OnCreate.

После того как фрейм создан, его необходимо вывести на экран. Библиотека предоставляет несколько функций для реализации этого процесса. Для отображения нашего главного окна мы остановились на одной из них — функции CWnd::SetWindowPos:

BOOL CWnd::**SetWindowPos**(
const CWnd *pWndInsertAfter,
int x,
int y,
int cx,
int cy,
UINT nFlags) —

Вызов этой функции позволяет изменить как размеры, так и расположение дочерних и всплывающих окон относительно экрана и других окон. Она имеет следующие параметры: pWndInsertAfter — указывает на объект CWnd, после которого будет расположено окно, вместо этого указателя можно использовать одно из следующих предопределенных значений, задающих расположение окна относительно других окон:

wndBottom	Разместить окно под другими окнами (внизу стека Z-порядка); если окно имело статус wndTopMost, то этот статус теряется — удаляется флаг расширенного стиля WS_EX_TOPMOST
wndTop	Разместить окно над другими окнами (на вершине стека Z-порядка)
wndTopMost	Разместить окно над всеми другими окнами, имеющими статус wndTopMost; эта позиция сохраняется (флаг расширенного стиля WS_EX_TOPMOST остается установленным), даже когда окно находится в неактивном состоянии
wndNoTopMost	Перемещает окно выше всех окон, имеющих статус wndTop, но ниже окон, имеющих статус wndTopMost

x, y, cx, cy — задают новые значения, соответственно, горизонтального и вертикального расположения окна, его ширины и высоты; nFlags — определяет параметры размера и позиции окна и может принимать одно из следующих значений:

SWP_DRAWFRAME	Задает рамку вокруг окна, определенную в классе окна
SWP_HIDEWINDOW	Делает окно невидимым
SWP_NOACTIVATE	Блокирует активизацию окна
SWP_NOMOVE	Предписывает окну сохранение текущей позиции; параметры x и y игнорируются
SWP_NOREDRAW	Предписывает не перерисовывать окно; после перемещения приложение должно перерисовать окно самостоятельно
SWP_NOSIZE	Задает сохранение текущих размеров; параметры cx и cy игнорируются
SWP_NOZORDER	Задает сохранение текущего расположения окна относительно других окон, при этом параметр pWndInsertAfter игнорируется
SWP_SHOWWINDOW	Отображает окно

Примечание

Windows располагает окна на экране согласно их положению в так называемом Z-порядке, который определяет позицию окна в стеке перекрывающихся окон. Стек окна ориентирован вдоль воображаемой оси (Z-оси) внутрь экрана. Окно, расположенное на вершине стека, перекрывает все другие окна, а расположенное внизу — перекрывается всеми другими окнами. Приложение не может активизировать окно без того, чтобы не перенести его на самый верх Z-порядка. Кроме того, расположение окна относительно экрана и других окон может изменяться без каких-либо ограничений.

Мы использовали эту функцию для указания размеров и расположения окна, а также для того, чтобы сделать его видимым.

Примечание

Существуют и другие способы решения этой задачи, они описаны ниже.

Отложим пока обсуждение вопросов создания и разрушения объекта класса CStylesApp до тех пор, пока не рассмотрим, как создаются дочерние окна.

Примечание

Поскольку любое порожденное окно программисты называют "дочерним", то часто возникает некоторая путаница. Вы должны совершенно четко отличать технический термин WS_CHILD от более широкого разговорного понятия "дочерний".

Создание дочерних окон

Теперь, после создания основного окна приложения, можно переходить к созданию некоторого дополнительного пространства для отображения информации или для получения команд от пользователя. Каждому "ребенку", как в свое время и основному окну, можно приписать конкретный стиль таким образом, чтобы приспособить его внешний вид и назначение к стоящим перед ним целям. В приложении Styles мы разбили создаваемые окна на два набора. Окна первого располагаются на экране независимо от положения основного окна — все они имеют стиль WS_POPUP, а для окон из второго набора установлен стиль WS_CHILD и, следовательно, они не могут покинуть пределы рабочей области основного окна.

Для того чтобы понять принципы построения и работу рассматриваемой программы, вернемся к обработчику сообщения WM_CREATE. Повторим ту часть программы, на которую следует обратить внимание:

```
extern char szNameWnd[128];
// Создаем пять дочерних окон
  for(int i = 0; i < 5; i++)
  {
    apChildWnd[i] = new CChildWnd;
    apChildWnd[i]->Create(NULL, NULL, WS_VISIBLE,
      CRect(0, 0, 0, 0), this, (UINT)i);
  }
  // Устанавливаем новый курсор для класса дочерних окон
  ::SetClassLong(apChildWnd[0]->GetSafeHwnd(), GCL_HCURSOR,
    (LONG)AfxGetApp()->LoadStandardCursor(IDC_IBEAM));
  // Создаем четыре всплывающих окна
  for(i = 0; i < 4; i++)
  {
    apPopupWnd[i] = new CPopupWnd;
    apPopupWnd[i]->Create(NULL, NULL, 0,
```

```
      rectDefault, this);
    // Указываем номер окна в поле данных, ассоциированных с окном
    ::SetWindowLong(apPopupWnd[i]->GetSafeHwnd(),
      GWL_USERDATA, i);
    // Формируем новый заголовок окна
    sprintf(szNameWnd, "Окно %d", i);
    apPopupWnd[i]->SetWindowText(szNameWnd);
  }
  // Создаем перекрывающееся окно
  apPopupWnd[4] = new CPopupWnd;
  apPopupWnd[4]->Create(NULL, NULL, 0, rectDefault);
  ::SetWindowLon2g(apPopupWnd[i]->GetSafeHwnd(),
    GWL_USERDATA, 4);
  sprintf(szNameWnd, "Окно %d", i);
  apPopupWnd[4]->SetWindowText(szNameWnd);
  // Устанавливаем новый курсор для класса всплывающих окон
  ::SetClassLong(apPopupWnd[0]->GetSafeHwnd(), GCL_HCURSOR,
    (LONG)AfxGetApp()->LoadStandardCursor(IDC_UPARROW));
  // Создаем и устанавливаем новый цвет фона для всплывающих окон
  HBRUSH hNewBrush = ::CreateSolidBrush(RGB(255, 255, 220));
  ::SetClassLong(apPopupWnd[3]->GetSafeHwnd(),
    GCL_HBRBACKGROUND, (LONG)hNewBrush);
  // Отображаем всплывающие окна
  for(i = 0; i < 5; i++)
  {
    apPopupWnd[i]->ShowWindow(SW_SHOWNORMAL);
    apPopupWnd[i]->UpdateWindow();
  }
```

При ближайшем рассмотрении — это не более чем создание десяти окон, принадлежащих двум новым классам.

Объекты одного класса позволяют создавать дочерние окна:

```
class CChildWnd : public CWnd
{
  ...
};
```

а объекты второго — всплывающие и/или перекрывающиеся окна (хотя с тем же успехом можно создавать и дочерние):

```
class CPopupWnd : public CFrameWnd
{
  ...
};
```

Такое разбиение не продиктовано необходимостью и произведено лишь для наглядности. Дело в том, что, как мы упоминали, на базе класса CWnd можно создавать только дочерние окна (имеющие стиль WS_CHILD). Его мы и решили использовать для этих целей. На базе же класса CFrameWnd могут быть созданы окна любого из трех имеющихся типов. Поэтому для создания всплывающих окон именно он был использован в качестве базового.

В основном процесс создания окон обоих типов стандартен: создаются объекты классов, после чего вызывается некоторая функция для создания окна Windows. С целью рассмотрения как можно большего числа предоставляемых библиотекой возможностей мы включили в текст программы дополнительные строки. Но обо всем по порядку. Начнем с дочерних окон, имеющих установленный бит стиля WS_CHILD (процесс создания объекта класса ничем не отличается от уже рассмотренных, и мы больше не будем останавливаться на этом вопросе). Итак, вызов функции CWnd::Create:

```
apChildWnd[i]->Create(NULL, NULL, WS_VISIBLE,
  CRect(0, 0, 0, 0), this, (UINT)i);
```

Подчеркнем некоторые моменты. Во-первых, не определен класс Windows, к которому относится окно, — эту задачу мы переложили на плечи библиотеки. Во-вторых, установив бит стиля WS_VISIBLE, мы тем самым задали, чтобы окно сразу же после создания было выведено на экран. В-третьих, окно мы превратили в точку и поместили его в левый верхний угол экрана. В-четвертых, окно имеет "родителя" — пятый параметр определен как this. И, наконец, в качестве последнего параметра мы передаем номер создаваемого окна.

Как и при создании главного окна, мы переопределяем функцию:

```
BOOL CChildWnd::PreCreateWindow(CREATESTRUCT& cs)
{
  cs.style |= adwStyles[nCount];
  sprintf(szNameWnd, "Окно %d", nCount);
  cs.lpszName = szNameWnd;
  return CWnd::PreCreateWindow(cs);
}
```

и обработчик сообщения WM_CREATE:

```
int CChildWnd::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
```

```
  if (CWnd::OnCreate(lpCreateStruct) == -1)
    return -1;
  MoveWindow(nCount++ * 118, 24, 116, 96);
  return 0;
}
```

Первая заполняет поля style и lpszName структуры CREATESTRUCT соответственно значениями стиля и имени конкретного окна, а вторая — устанавливает размер и положение вновь созданного окна (напомним, что мы задали окну нулевые размеры). Со структурой CREATESTRUCT мы уже познакомились, поэтому рассмотрим изящную функцию CWnd::MoveWindow, которая реализована в двух вариантах:

void CWnd::**MoveWindow**(
int x,
int y,
int nWidth,
int nHeight,
BOOL bRepaint = TRUE);

и

void CWnd::**MoveWindow**(
LPCRECT lpRect,
BOOL bRepaint = TRUE) —

с помощью любой из этих функций можно одновременно изменить размер и положение окна, задаваемые в параметрах x, y, nWidth, nHeight или lpRect. Последний параметр этой функции определяет, будет ли окно перерисовываться после перемещения. Если этот параметр равен TRUE (по умолчанию), то объект "окно" получает сообщение Windows WM_PAINT. Кроме того, эта функция посылает еще и сообщение WM_GETMINMAXINFO, которое вы также можете обработать.

Примечание

Здесь следует иметь в виду, что позиция дочернего окна определяется координатами его владельца, в то время как для всплывающего окна положение задается относительно экрана.

Как мы уже отметили, задача регистрации оконного класса была поручена библиотеке. Нам лишь остается "подправить" некоторые поля в соответствии с собственными нуждами. Один из возможных путей при этом состоит в использовании функций API SetClassLong и SetWindowLong. Возможно, вы захотите их использовать в своих программах, поэтому рассмотрим их подробнее:

DWORD **SetClassLong**(
HWND hWnd,
int nIndex,
LONG dwNewLong) —

эта функция устанавливает в структуре WNDCLASS, описывающей оконный класс для hWnd, новое значение dwNewLong в поле, определяемом параметром nIndex, который может принимать одно из следующих значений:

GCL_CBCLSEXTRA	Устанавливает размер (в байтах) дополнительной памяти, предназначенной для использования всеми окнами, создаваемыми на базе этого класса; размер памяти, уже ассоциированной с классом, при этом не изменяется
GCL_CBWNDEXTRA	Устанавливает размер (в байтах) дополнительной памяти, предназначенной для использования отдельно каждым окном, создаваемыми на базе этого класса; размер памяти, уже ассоциированной с окном, при этом не изменяется
GCL_HBRBACKGROUND	Устанавливает дескриптор кисти для фона, ассоциированного с этим классом
GCL_HCURSOR	Устанавливает дескриптор курсора, ассоциированного с этим классом
GCL_HICON	Устанавливает дескриптор значка, ассоциированного с этим классом; это значение не определено в Windows NT
GCL_HMODULE	Устанавливает дескриптор модуля, который зарегистрировал этот класс
GCL_MENUNAME	Устанавливает адрес строки меню, которая идентифицирует ресурс меню, ассоциированного с этим классом
GCL_STYLE	Устанавливает биты стиля оконного класса
GCL_WNDPROC	Устанавливает адрес оконной процедуры, ассоциированной с классом

При успешном выполнении функция возвращает значение указанного слова из структуры класса окна, либо нулевое значение при ошибке. Дополнительную информацию об ошибке можно получить, вызвав функцию API GetLastError.

Мы воспользовались этой функцией для того, чтобы установить новые значения для курсора и цвета фона.

Обратную задачу — получение информации из структуры класса окна — выполняет функция:

```
DWORD GetClassLong(
    HWND hWnd,
    int nIndex)
```

Для работы с дополнительной памятью в структуре окна API предоставляет также две функции:

```
DWORD SetWindowLong(
    HWND hWnd,
    int nIndex,
    LONG dwNewLong)
```

и

DWORD **GetWindowLong**(
HWND hWnd,
int nIndex) —

По своему принципу действия они аналогичны только что рассмотренным, но работают с дополнительной памятью в структуре окна. Параметр nIndex при этом может принимать следующие значения:

GWL_EXSTYLE	Устанавливает новый расширенный стиль окна
GWL_STYLE	Устанавливает новый стиль окна
GWL_WNDPROC	Устанавливает новый адрес оконной процедуры
GWL_HINSTANCE	Устанавливает новый дескриптор приложения
GWL_ID	Устанавливает новый идентификатор окна
GWL_USERDATA	Устанавливает 32-разрядное значение, ассоциированное с конкретным окном и предназначенное для использования приложением, которым это окно создано

Если дескриптор hWnd определяет блок диалога, то доступными являются также следующие значения:

DWL_DLGPROC	Устанавливает новый адрес процедуры блока диалога
DWL_MSGRESULT	Устанавливает значение, возвращаемое при обработке сообщения в процедуре блока диалога
DWL_USER	Устанавливает новую дополнительную информацию для приложения, такую как идентификатор или указатель

Для функции GetWindowLong, кроме перечисленных, определено еще одно доступное значение:

GWL_HWNDPARENT	Возвращает дескриптор родительского окна или NULL, если его нет

Мы использовали эти функции для записи в дополнительную память номера созданного окна. Другими словами, у первого окна в этом поле будет нуль, у второго — 1, у третьего — 2 и т. д. Позже, во время отображения окон на экране, программа воспользуется этими значениями для идентификации окон.

Перед тем как закончить обсуждение вопроса об изменении характеристик оконных классов и собственно окон, осталось отметить способ получения дескриптора окна Windows. Для этой цели в классе CWnd предусмотрена специальная функция:

HWND CWnd::**GetSafeWnd**( ) —

позволяет получить дескриптор окна Windows, хранящийся в компоненте класса CWnd::m_hWnd, или NULL, если к объекту не присоединено никакое окно.

Но вернемся к нашей программе. Осталось разобраться в создании всплывающих окон. Обратите внимание на два несколько отличающихся вызова функции CFrameWnd::Create:

```
apPopupWnd[i]->Create(NULL, NULL, 0, rectDefault, this);
```

и

```
apPopupWnd[4]->Create(NULL, NULL, 0, rectDefault);
```

и связанное с этим отличие в поведении всплывающих окон 3 и 4. Для четвертого окна мы не указали родителя, и поэтому оно ведет себя достаточно независимо от нашего главного окна.

Второй момент, на который хотелось бы обратить ваше внимание, связан с еще одним способом изменения стилей созданных окон. Обратимся к тексту:

```
int CPopupWnd::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
  ModifyStyle((DWORD)-1, adwStyles[nCount]);
...
  return 0;
}
```

Как видите, для установки стиля окна мы воспользовались функцией класса CWnd:

BOOL CWnd::**ModifyStyle**(
DWORD dwRemove,
DWORD dwAdd,
UINT nFlags = 0) —

позволяет удалить (dwRemove) или установить (dwAdd) определенные атрибуты стиля окна. Если установлен параметр nFlags, то вызывается функция API ::SetWindowPos с этим параметром, являющимся комбинацией одного или нескольких значений: SWP_NOMOVE, SWP_NOACTIVATE, SWP_NOSIZE и SWP_NOZORDER.

Для изменения флагов расширенного стиля можно использовать функцию:

BOOL CWnd::**ModifyStyleEx**(
DWORD dwRemove,
DWORD dwAdd,
UINT nFlags = 0)

Попутно упомянем две функции, которые позволяют узнать текущие обычный и расширенный стили окна:

DWORD CWnd::**GetStyle**( )

и

DWORD CWnd::**GetExStyle**( )

При рассмотрении вопроса о создании главного окна приложения мы обошли молчанием конструктор и деструктор объекта этого окна. Вернемся теперь к их содержимому:

```
CMainFrame::CMainFrame()
{
  for(int i = 0; i < 5; i++)
  {
    apChildWnd[i] = NULL;
  }
}
CMainFrame::~CMainFrame()
{
  for(int i = 0; i < 5; i++)
  {
    if(apChildWnd[i] != NULL)
    {
      delete apChildWnd[i];
    }
  }
}
```

Как видите, мы явно освобождаем только ту память, которую занимали объекты, образованные из класса, производного от CWnd. При использовании в качестве базовых других классов (в нашем случае CFrameWnd) явно освобождать память не надо, т. к. объекты выполняют это сами.

Ограничение размеров окна

Мы создали главное окно приложения Styles таким образом, что пользователь не может изменить его размеры. Более того, окно не изменит свои размеры даже тогда, когда вы попытаетесь развернуть его. Такое поведение окна достигается введением соответствующей обработки сообщения Windows WM_GETMINMAXINFO, которая выполняется обработчиком:

```
void CMainFrame::OnGetMinMaxInfo(MINMAXINFO FAR* lpMMI)
{
  int cx = ::GetSystemMetrics(SM_CXSCREEN)/8;
  int cy = ::GetSystemMetrics(SM_CYSCREEN)/4;
  lpMMI->ptMinTrackSize.x = 6*cx;
  lpMMI->ptMinTrackSize.y = 2*cy;
  lpMMI->ptMaxTrackSize.x = 6*cx;
  lpMMI->ptMaxTrackSize.y = 2*cy;
```

```
  lpMMI->ptMaxPosition.x = cx;
  lpMMI->ptMaxPosition.y = cy;
  CFrameWnd::OnGetMinMaxInfo(lpMMI);
}
```

Обработчику в качестве параметра передается указатель на структуру типа MINMAXINFO, которая содержит информацию о максимальных возможных размерах и позиции окна, а также о минимальных и максимальных значениях изменяемых размеров:

```
typedef struct tagMINMAXINFO {
  POINT ptReserved;
  POINT ptMaxSize;
  POINT ptMaxPosition;
  POINT ptMinTrackSize;
  POINT ptMaxTrackSize;
} MINMAXINFO;
```

Поля этой структуры имеют следующее назначение: ptReserved — зарезервировано для внутреннего использования; ptMaxSize — устанавливает размеры окна, которые оно может иметь в развернутом виде; ptMaxPosition — устанавливает координаты, которые должен иметь левый верхний угол развернутого окна; ptMinTrackSize — устанавливает минимальные размеры, до которых пользователь может сжать границы окна; ptMaxTrackSize — устанавливает максимальные размеры, до которых пользователь может растянуть границы окна.

Как видите, все очень просто. Последнее, что нам осталось рассмотреть, — каким образом можно получить информацию о системных параметрах.

Функция GetSystemMetrics

Эта функция позволяет получить размеры (высоту и ширину) изображаемых элементов Windows, а также другую информацию. Все размеры даются в пикселях:

int **GetSystemMetrics**(int nIndex) —

значение nIndex определяет параметр, который вы хотите получить (возвращаемые функцией значения могут изменяться в зависимости от типа используемого дисплея и установок пользователя). Приведем небольшой список наиболее часто используемых параметров:

SM_CMOUSEBUTTONS	Число кнопок мыши; если мышь не установлена, то возвращается 0
SM_CXDORDER, SM_CYDORDER	Ширина и высота рамки окна
SM_CXFULLSCREEN, SM_CYFULLSCREEN	Ширина и высота рабочей области развернутого окна

SM_CXICON, SM_CYICON	Ширина и высота большого значка, заданные по умолчанию; обычно они равны 32, но могут изменяться в зависимости от установок пользователя
SM_CXMIN, SM_CYMIN	Минимальные допустимые ширина и высота окна
SM_CXMINTRACK, SM_CYMINTRACK	Минимальные величины, до которых пользователь может уменьшить ширину и высоту окна; для их изменения необходимо обработать сообщение Windows WM_GETMINMAXINFO
SM_CXSCREEN, SM_CYSCREEN	Ширина и высота экрана
SM_CYCAPTION	Высота области заголовка окон
SM_CYMENU	Высота полосы меню
SM_SWAPBUTTON	Не нуль, если левая и правая кнопки мыши обменялись своими функциями
SM_CXEDGE, SM_CYEDGE	Горизонтальный и вертикальный размеры трехмерной рамки
SM_CXFIXEDFRAME, SM_CYFIXEDFRAME	Толщина (по горизонтали и вертикали) рамки окна, которое имеет заголовок, но не может изменить свои размеры
SM_CXMAXIMIZED, SM_CYMAXIMIZED	Предельные допустимые размеры развернутого окна
SM_CXMAXTRACK, SM_CYMAXTRACK	Максимальные величины, до которых пользователь может увеличить ширину и высоту окна; для их изменения необходимо обработать сообщение Windows WM_GETMINMAXINFO
SM_CXMINIMIZED, SM_CYMINIMIZED	Предельные допустимые размеры свернутого окна
SM_CXSIZEFRAME, SM_CYSIZEFRAME	Толщина (по горизонтали и вертикали) рамки окна, которое может изменять свои размеры
SM_CXSMICON, SM_CYSMICON	Рекомендуемые размеры маленького значка, который обычно используется в заголовке окна
SM_CYSMCAPTION	Высота области уменьшенного заголовка
SM_NETWORK	В возвращаемом значении установлен самый младший бит, если сеть присутствует; остальные биты зарезервированы для будущего использования
SM_SECURE	Функция возвращает ненулевое значение, если установлена защита, и 0 в противном случае

Ограничение доступа к окну

Большинство из стилей окна, использованных в приложении Styles, достаточно ясны. Поэтому мы остановимся только на одном из них — WS_DISABLED. Окна, имеющие такой стиль, абсолютно неподвижны, и ими нельзя манипулировать ни с помощью мыши, ни с помощью клавиатуры. Обратите также внимание, что у такого окна и курсор не соответствует заданному в оконном классе. Для доступа к этому свойству окон в классе CWnd реализована специальная функция:

BOOL CWnd::**EnableWindow**(BOOL bEnable = TRUE) —

разрешает (bEnable = TRUE) или запрещает (bEnable = FALSE) доступ к окну для мыши и клавиатуры. Если при вызове этой функции изменилось состояние окна, то до возврата из нее Windows посылает окну сообщение WM_ENABLE.

Для того чтобы узнать текущее состояние окна в смысле его доступности, можно использовать функцию:

BOOL CWnd::**IsWindowEnable**( ) —

возвращает значение TRUE, если окно доступно, и FALSE в противном случае.

Основная цель рассмотренного приложения и этой главы заключается в знакомстве с основными классами библиотеки MFC, отвечающими за создание и работу окон Windows практически всех типов. Именно по этой причине мы опять не использовали AppWizard для создания каркаса приложения. Исключение составляют окна так называемых MDI-приложений — приложений многодокументного (или многооконного) интерфейса, к которым мы сейчас и переходим.

Глава 10

Создание многодокументных приложений

Мы надеемся, что нет необходимости подробно представлять *многодокументный интерфейс* (MDI, Multiple Document Interface), который позволяет в одном приложении работать с несколькими документами или с разными представлениями одного и того же документа. Этот интерфейс описан в руководстве по разработке интерфейса пользователя System Application Architecture Common User Access Advanced Interface Design Guide (SAA/CUA), созданном фирмой IBM. В Windows интерфейс MDI используется, начиная с версии 3.0, типичным примером приложения на базе MDI является хорошо всем знакомый текстовый процессор Microsoft Word для Windows.

Как и все приложения Windows, MDI-приложение имеет одно главное окно, которое управляет своими окнами "MDI child". Поведение окон "MDI child" напоминает поведение как дочерних, так и перекрывающихся окон. Так же как и первые, их невозможно переместить за пределы главного окна приложения. И в то же время окна "MDI child" можно сделать активными, при этом их заголовок выделяется цветом. Кроме того, каждое окно "MDI child" имеет, как правило, системное меню и кнопки изменения размера. Если свернуть такое окно, то его значок расположится в нижней части главного окна приложения. При увеличении размеров окна до максимальных, пиктограмма системного меню окна "MDI child" помещается в левую часть основного меню, а в правой части расположатся кнопки изменения размера и закрытия окна.

Примечание

Для названия окна документа мы будем придерживаться терминологии фирмы Microsoft, которая для дочерних окон MDI-приложений использует названия "MDI child frame" или "MDI child window".

Одним словом, можно, конечно, создать аналогичное приложение с использованием обычных окон, но для этого придется затратить немало усилий.

Более того, достаточно серьезную работу надо проделать, чтобы создать полноценное MDI-приложение, используя функции библиотеки SDK (Software Development Kit). Для того чтобы в этом убедиться, достаточно рассмотреть исходные тексты приложения MULTIPAD, поставляемые в составе примеров приложений с системой разработки Microsoft Developer Studio Visual C++. Совсем другая картина предстанет перед вами, когда вы возьмете исходные тексты такого же приложения, но созданного с использованием библиотеки MFC. Мы попытаемся обойти эти две крайние позиции и рассмотреть интерфейс MDI одновременно с деталями его внутренних механизмов, знание которых позволит вам создавать приложения любой сложности наиболее эффективно и с наименьшими затратами на поиск ошибок, которые зачастую возникают из-за непонимания происходящего.

Начнем с рассмотрения двух основных классов библиотеки MFC, разработанных специально для создания многодокументных приложений, — CMDIFrameWnd и CMDIChildWnd.

Класс CMDIFrameWnd

CMDIFrameWnd — базовый класс для главного окна любого многодокументного приложения. Точнее, в данном случае следует говорить не об окне, а о фрейме, поскольку помимо него создается специальное окно, называемое окном MDICLIENT, которое управляет рабочей областью фрейма и является, собственно, родительским для окон "MDI child".

Рис. 10.1. Типы окон интерфейса MDI

Уточним некоторые моменты, которые связаны с процессом создания MDI-окна. Как обычно, сначала создается объект нашего оконного класса. Далее, если вы сравните тексты функции InitInstance для создания главных окон SDI- и MDI-приложений, то не увидите никакой разницы. И в том и в другом случае вызываются одни и те же функции. Однако если при создании SDI-окна вызывается функция CFrameWnd::OnCreateClient, то при создании

MDI-окна вызывается функция CMDIFrameWnd::OnCreateClient. Первая отвечает за создание (если необходимо) присоединяемого к окну представления, а вторая — за создание специального окна MDICLIENT, которое, как уже говорилось, полностью управляет рабочей областью главного окна MDI-приложения и отвечает за работу с окнами "MDI child". На рис. 10.1 показаны все перечисленные окна MDI-приложения.

Осталось рассмотреть функциональные возможности, которые класс CMDIFrameWnd предоставляет в наше распоряжение.

void CMDIFrameWnd::**MDIActivate**(CWnd *pWndActivate) —

pWndActivate — указатель на окно "MDI child", которое должно быть активизировано.

Эта функция посылает сообщение WM_MDIACTIVATE в два окна: сначала в то, которое активизируется, затем в то, которое теряет активность. То же сообщение посылается, если пользователь изменил фокус дочернего окна при помощи мыши или клавиатуры. Окна "MDI child" активизируются независимо от основного MDI-окна. Когда активизируется основное окно, то дочернее, которое было активно последним, посылает сообщение WM_NCACTIVATE, чтобы нарисовать его рамку и заголовок, но при этом не получает сообщения WM_MDIACTIVATE.

CMDIChildWnd* CMDIFrameWnd::**MDIGetActive**(BOOL *pbMaximized = NULL) —

возвращает указатель на текущее активное окно "MDI child". Дополнительно, в параметре pbMaximized, возвращается информация о том, развернуто это окно или нет.

void CMDIFrameWnd::**MDIIconArrange**( ) —

упорядочивает все свернутые окна "MDI child", не затрагивая те, которые не свернуты.

void CMDIFrameWnd::**MDIMaximize**(CWnd *pWnd) —

разворачивает окно "MDI child". При этом Windows изменяет его размеры таким образом, чтобы оно заполнило всю рабочую область окна фрейма. Windows размещает системное меню дочернего окна в полосе основного меню, а его заголовок добавляется к заголовку окна фрейма.

Примечание

Если активное окно развернуто, и в этот момент активизируется другое окно "MDI child", то Windows его также разворачивает.

void CMDIFrameWnd::**MDINext**( ) —

активизирует следующее окно "MDI child".

void CMDIFrameWnd::**MDIRestore**(CWnd *pWnd) —

восстанавливает исходные размеры окна "MDI child", заданного параметром pWnd, если оно было развернуто или свернуто.

CMenu* CMDIFrameWnd::**MDISetMenu**(
CMenu *pFrameMenu,
CMenu *pWindowMenu) —

если pFrameMenu и/или pWindowMenu не равны NULL, то заменяет соответствующие текущие меню на переданные в качестве параметров. Возвращает временный указатель на замененное меню.

После вызова этой функции следует вызвать функцию CWnd::DrawMenuBar для обновления полосы меню. Если функция изменяет раскрывающееся меню, то элементы меню окна "MDI child" удаляются из предыдущего меню окна и добавляются в раскрывающееся меню. Если вы используете фрейм для управления окнами "MDI child", то не следует вызывать эту функцию.

void CMDIFrameWnd::**MDITile**( )

и

void CMDIFrameWnd::**MDITile**(int nType) —

располагают все окна "MDI child" без перекрытия. Первая версия функции располагает окна по вертикали, а второй в качестве параметра nType можно передавать одно из следующих значений:

MDITILE_HORIZONTAL	Расположить окна по горизонтали
MDITILE_VERTICAL	Расположить окна по вертикали
MDITILE_SKIPDISABLED	Предотвращает отображение блокированных окон "MDI child"

void CMDIFrameWnd::**MDICascade**( )

и

void CMDIFrameWnd::**MDICascade**(int nType) —

располагают все окна "MDI child" каскадом. Первая версия функции отображает и блокированные окна, а вторая, которая может быть вызвана только с параметром MDITILE_SKIPDISABLED, — все, кроме блокированных.

virtual BOOL CMDIFrameWnd::**CreateClient**(
LPCREATESTRUCT lpCreateStruct,
Cmenu *pWindowMenu) —

создает для текущего объекта CMDIFrameWnd окно Windows MDICLIENT, которое управляет объектами CMDIChildWnd. Эта функция должна быть вызвана, если вы переопределили функцию CWnd::OnCreate.

Примечание

Если вы переопределяете эту функцию, то не забывайте вызывать функцию базового класса, т. к. фирма Microsoft не рекомендует производить какие-либо изменения, связанные с окном MDICLIENT.

virtual HMENU CMDIFrameWnd::**GetWindowMenuPopup**(HMENU hMenuBar) —

позволяет получить дескриптор раскрывающегося меню для элемента меню **Window** (окно) основного меню, заданного параметром hMenuBar. Реализация, заданная по умолчанию, ищет раскрывающееся меню, которое содержит стандартные команды элемента меню **Window** основного меню типа ID_WINDOW_NEW и ID_WINDOW_TILE_HORZ. Если это раскрывающееся меню не содержит идентификаторов стандартных команд, то эту функцию следует переопределить.

Как видите, всю основную работу по поддержанию работы MDI-приложений выполняет ОС Windows совместно с библиотекой классов MFC.

Прежде чем двигаться дальше, давайте так же внимательно рассмотрим класс, который отвечает за функционирование окон "MDI child".

Класс CMDIChildWnd

Как вы уже поняли, основным назначением этого класса является обеспечение работоспособности дочерних окон MDI-приложений — окон "MDI child", которые, говоря упрощенно, позаимствовали свои свойства как у дочерних, так и у перекрывающихся окон.

Большую часть возможностей этот класс наследует от своего базового — CFrameWnd, но имеет и свойства, присущие только ему. К ним можно отнести автоматическое отображение меню окна "MDI child" вместо основного меню MDI-приложения, а также добавление заголовка дочернего окна к заголовку его родителя.

Для того чтобы произошла автоматическая замена основного меню на меню окна "MDI child", в классе определена единственная переменная:

HMENU CMDIChildWnd::**m_hMenuShared** —

содержит дескриптор меню, ассоциированного с окном "MDI child".

Сохранив в ней дескриптор меню дочернего окна, вы можете больше не заботиться о вопросах переключения между различными меню.

Кроме того, в классе определены несколько функций:

BOOL CMDIChildWnd::**Create**(
LPCTSTR lpszClassName,
LPCTSTR lpszWindowName,
DWORD dwStyle = WS_CHILD|WS_VISIBLE |
WS_OVERLAPPEDWINDOW,
const RECT &rect = rectDefault,
CMDIFrameWnd *pParentWnd = NULL,
CCreateContext *pContext = NULL) —

эта функция абсолютно идентична функции CFrameWnd::Create, однако задачи, которые они решают, достаточно сильно различаются, и нельзя подменять вызов одной из них

вызовом другой. Как уже упоминалось, активное текущее окно "MDI child" может определять заголовок своего родительского окна, если для него установить бит стиля FWS_ADDTOTITLE. Эта функция вызывается MDI-окном в ответ на команду пользователя создать дочернее окно. При этом параметр pContext позволяет правильно связать дочернее окно с приложением. Если такой необходимости нет, то для него можно использовать значение по умолчанию (NULL).

void CMDIChildWnd::**MDIDestroy**( ) —

удаляет заголовок окна "MDI child" из MDI-окна и разрушает само окно "MDI child".

void CMDIChildWnd::**MDIActivate**( ) —

активизирует текущее окно "MDI child". Когда MDI-окно становится активным, то активизируется и окно "MDI child", которое находилось в этом состоянии последним.

void CMDIChildWnd::**MDIMaximize**( ) —

разворачивает текущее окно "MDI child". Если у него не снят бит стиля FWS_ADDTOTITLE, то его заголовок добавляется в MDI-окно.

void CMDIChildWnd::**MDIRestore**( ) —

восстанавливает исходные размеры текущего окна "MDI child", если оно было развернуто или свернуто.

CMDIFrameWnd* CMDIChildWnd::**GetMDIFrame**( ) —

позволяет получить указатель на родительское MDI-окно объекта класса CMDIFrameWnd. Для того чтобы получить указатель на родительское окно типа MDICLIENT, которое управляет объектом класса CMDIChildWnd, необходимо вызвать функцию CWnd::GetParent.

Теперь, когда мы познакомились с обоими классами, позволяющими создавать приложения на базе интерфейса MDI, пришло время рассмотреть конкретный пример, который позволит закрепить полученные сведения и ознакомиться с дополнительными возможностями библиотеки MFC.

MDI — пример приложения

В этом разделе представлен текст приложения, на примере которого мы рассмотрим основные принципы построения MDI-приложений. Оно может послужить основой для разработки многодокументных приложений любой степени сложности. Кроме того, будут рассмотрены некоторые предварительные вопросы обработки команд меню и работы со шрифтами.

Как и в примере, приведенном в предыдущей главе, все англоязычные комментарии вставлены в текст приложения средством AppWizard или ClassWizard, и оставлены без изменения. Результат работы приложения представлен на рис. 10.2.

```
/////////////////////////////////////////////////////////////
// Файл MDI.h
// Copyright (c) 1999 Мешков А., Тихомиров Ю.
```

```
/////////////////////////////////////////////////////////////////////////////
// MDI.h : main header file for the MDI application
//
#ifndef __AFXWIN_H__
  #error include 'stdafx.h' before including this file for PCH
#endif
#include "resource.h"       // main symbols
/////////////////////////////////////////////////////////////////////////////
// CMDIApp:
// See MDI.cpp for the implementation of this class
//
class CMDIApp : public CWinApp
{
public:
  CMDIApp();
// Overrides
  // ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CMDIApp)
  public:
  virtual BOOL InitInstance();
  //}}AFX_VIRTUAL
// Implementation
  //{{AFX_MSG(CMDIApp)
  afx_msg void OnAppAbout();
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()
};
/////////////////////////////////////////////////////////////////////////////
// Файл MDI.cpp
// Copyright (c) 1999 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////////////////////////
// MDI.cpp : Defines the class behaviors for the application.
//
#include "stdafx.h"
#include "MDI.h"
#include "AppAbout.h"
#include "MainFrm.h"
/////////////////////////////////////////////////////////////////////////////
// CMDIApp
```

```
BEGIN_MESSAGE_MAP(CMDIApp, CWinApp)
  //{{AFX_MSG_MAP(CMDIApp)
  ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()
/////////////////////////////////////////////////////////////
// CMDIApp construction
CMDIApp::CMDIApp()
{
}
/////////////////////////////////////////////////////////////
// The one and only CMDIApp object
CMDIApp theApp;
/////////////////////////////////////////////////////////////
// CMDIApp initialization
BOOL CMDIApp::InitInstance()
{
// Standard initialization
  CString csCaption;
// Загружаем строковый ресурс с именем приложения
  csCaption.LoadString(IDR_MAINFRAME);
// Проверяем, не было ли приложение уже загружено
  CWnd *pWnd = CWnd::FindWindow(NULL, csCaption);
  if(pWnd != NULL){
  // Если да, то делаем его видимым и переводим на передний план.
  // Однако если вы хотите запускать произвольное число копий
  // приложения, то эти действия можно не выполнять
  if(pWnd->IsIconic())
    pWnd->ShowWindow(SW_RESTORE);
  pWnd->SetForegroundWindow();
  return FALSE;
  }
  Enable3dControls();   // Call this when using MFC in a shared DLL
  // create main MDI Frame window
  CMainFrame* pMainFrame = new CMainFrame;
  if (!pMainFrame->LoadFrame(IDR_MAINFRAME))
  return FALSE;
  m_pMainWnd = pMainFrame;
  pMainFrame->ShowWindow(m_nCmdShow);
```

```
  pMainFrame->UpdateWindow();
  return TRUE;
}
// App command to run the dialog
void CMDIApp::OnAppAbout()
{
  CAppAbout *aboutDlg = new CAppAbout();
}
/////////////////////////////////////////////////////////////
// Файл AppAbout.h
// Copyright (c) 1999 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////
// CAppAbout dialog
class CAppAbout : public CDialog
{
// Construction
public:
  CAppAbout(CWnd* pParent = NULL);   // standard constructor
// Dialog Data
  //{{AFX_DATA(CAppAbout)
  enum { IDD = IDD_ABOUTBOX };
  // NOTE: the ClassWizard will add data members here
  //}}AFX_DATA
// Overrides
  // ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CAppAbout)
  //}}AFX_VIRTUAL
// Implementation
protected:
// Generated message map functions
  //{{AFX_MSG(CAppAbout)
  // NOTE: the ClassWizard will add member functions here
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()
};
/////////////////////////////////////////////////////////////
// Файл AppAbout.cpp
// Copyright (c) 1999 Мешков А., Тихомиров Ю.
```

```
/////////////////////////////////////////////////////////////
// AppAbout.cpp : implementation file
//
#include "stdafx.h"
#include "MDI.h"
#include "AppAbout.h"
/////////////////////////////////////////////////////////////
// CAppAbout dialog
ppAbout::CAppAbout(CWnd* pParent /*=NULL*/)
{
  Create(CAppAbout::IDD, pParent);
  //{{AFX_DATA_INIT(CAppAbout)
  // NOTE: the ClassWizard will add member initialization here
  //}}AFX_DATA_INIT
}
BEGIN_MESSAGE_MAP(CAppAbout, CDialog)
  //{{AFX_MSG_MAP(CAppAbout)
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()
/////////////////////////////////////////////////////////////
// Файл MainFrm.h
// Copyright (c) 1999 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////////
// MainFrm.cpp : implementation of the CMainFrame class
//
#include "stdafx.h"
#include "MDI.h"
#include "MDIChilds.h"
#include "MainFrm.h"
/////////////////////////////////////////////////////////////
// CMainFrame
BEGIN_MESSAGE_MAP(CMainFrame, CMDIFrameWnd)
  //{{AFX_MSG_MAP(CMainFrame)
  ON_COMMAND(ID_FILE_TEXT, OnText)
  ON_WM_CREATE()
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()
LPCTSTR lpszTextClass;
```

```
/////////////////////////////////////////////////////////////
// CMainFrame message handlers
// Обработчик команды ID_FILE_TEXT
void CMainFrame::OnText()
{
  // Создаем объект класса для дочернего окна "MDI child"
  CTextWnd *pTextWnd = new CTextWnd;

  // Создает окно Windows "MDI child"
  if (!pTextWnd->Create(lpszTextClass,
                        _TEXT("Дочернее окно \"MDI child\""),
                        0, rectDefault, this))
  return;
}
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
  if (CMDIFrameWnd::OnCreate(lpCreateStruct) == -1)
  return -1;
// Регистрируем новый класс окна Windows для "MDI child"
 lpszTextClass = AfxRegisterWndClass(
     CS_HREDRAW | CS_VREDRAW,
     AfxGetApp()->LoadCursor(IDC_MDITEXT),
     (HBRUSH) (COLOR_WINDOW+1),
     AfxGetApp()->LoadIcon(IDI_MDITEXT));

 // Вызываем обработчик команды ID_FILE_TEXT
  OnText();
  return 0;
}
/////////////////////////////////////////////////////////////
// Файл MDIChilds.h
// Copyright (c) 1999 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////////
// MDIChilds.h : header file
//
/////////////////////////////////////////////////////////////
// CTextWnd frame
class CTextWnd : public CMDIChildWnd
{
```

```
public:
  CTextWnd();
// Implementation
protected:
  static CMenu    menu;     // меню для всех дочерних окон
  UINT     nIDColor;
  COLORREF clrText;
// Overrides
  // ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CTextWnd)
  protected:
  virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
  //}}AFX_VIRTUAL
// Generated message map functions
  //{{AFX_MSG(CTextWnd)
  afx_msg void OnColor();
  afx_msg void OnPaint();
  afx_msg void OnUpdateColor(CCmdUI* pCmdUI);
  afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()
};
/////////////////////////////////////////////////////////////
// Файл MDIChilds.cpp
// Copyright (c) 1999 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////////
// MDIChilds.cpp : implementation file
//
#include "stdafx.h"
#include "MDI.h"
#include "MDIChilds.h"
CMenu CTextWnd::menu;     // меню для всех дочерних окон
COLORREF colorArray[] =
{
  RGB (0, 0, 0),
  RGB (255, 255, 255),
  RGB (255, 0, 0),
  RGB (0, 0, 255),
```

```
  RGB (0, 255, 0)
};
BEGIN_MESSAGE_MAP(CTextWnd, CMDIChildWnd)
  //{{AFX_MSG_MAP(CTextWnd)
  ON_COMMAND(IDM_BLACK, OnColor)
  ON_COMMAND(IDM_BLUE, OnColor)
  ON_COMMAND(IDM_GREEN, OnColor)
  ON_COMMAND(IDM_RED, OnColor)
  ON_COMMAND(IDM_WHITE, OnColor)
  ON_UPDATE_COMMAND_UI(IDM_BLACK, OnUpdateColor)
  ON_UPDATE_COMMAND_UI(IDM_BLUE, OnUpdateColor)
  ON_UPDATE_COMMAND_UI(IDM_GREEN, OnUpdateColor)
  ON_UPDATE_COMMAND_UI(IDM_RED, OnUpdateColor)
  ON_UPDATE_COMMAND_UI(IDM_WHITE, OnUpdateColor)
  ON_WM_PAINT()
  ON_WM_LBUTTONDOWN()
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()
/////////////////////////////////////////////////////////////
// CTextWnd
CTextWnd::CTextWnd()
{
  nIDColor = IDM_BLACK;
  clrText = RGB(0, 0, 0);
}
/////////////////////////////////////////////////////////////
// CTextWnd message handlers
BOOL CTextWnd::PreCreateWindow(CREATESTRUCT& cs)
{
  cs.style |= WS_CHILD | WS_VISIBLE |
              WS_OVERLAPPEDWINDOW;
  // Загружаем меню дочернего окна
  if (menu.m_hMenu == NULL)
  menu.LoadMenu(IDR_MDITEXT);
  m_hMenuShared = menu.m_hMenu;

  return CMDIChildWnd::PreCreateWindow(cs);
}
```

```
// Обработчик сообщения WM_PAINT
void CTextWnd::OnPaint()
{
  CPaintDC dc(this);
  CRect rect;
  GetClientRect(rect);
  // Устанавливаем текущий цвет текста
  dc.SetTextColor(clrText);
  dc.SetBkColor(::GetSysColor(COLOR_WINDOW));
  // Отображаем текст в центре окна
  dc.DrawText(_TEXT("Обработка сообщения WM_PAINT"), -1, rect,
              DT_SINGLELINE | DT_CENTER | DT_VCENTER);
}
// Обработчик команд IDM_BLACK - IDM_WHITE (один на все)
void CTextWnd::OnColor()
{
  // Получаем идентификатор выбранной команды
  nIDColor = LOWORD(GetCurrentMessage()->wParam);

  // По идентификатору команды выбираем из массива
  // соответствующее значение цвета
  clrText = colorArray[nIDColor - IDM_BLACK];

  // Перерисовываем окно в ответ на сообщение WM_PAINT
  Invalidate();
}

// Обработчик команд обновления IDM_BLACK - IDM_WHITE (один на все)
void CTextWnd::OnUpdateColor(CCmdUI* pCmdUI)
{
  // Устанавливаем или убираем пометку у соответствующего
  // элемента меню
  pCmdUI->SetCheck(pCmdUI->m_nID == nIDColor);
}

// Обработчик сообщения WM_LBUTTONDOWN
void CTextWnd::OnLButtonDown(UINT nFlags, CPoint point)
{
```

```
  TCHAR strText[] = "Обработка сообщения WM_LBOTTONDOWN";

  CClientDC dc(this);
  CRect rect;

  // Устанавливаем текущий цвет текста
  dc.SetTextColor(clrText);
  dc.SetBkColor(::GetSysColor(COLOR_WINDOW));

  // Отображаем текст в окне, начиная с точки (10,10)
  dc.TextOut(10, 10, strText, strlen(strText));

  // Не забываем передать управление родительскому
  // окну для дальнейшей обработки
  CMDIChildWnd::OnLButtonDown(nFlags, point);
}

/////////////////////////////////////////////////////////////////////////////
// Файл resource.h
// Copyright (c) 1999 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////////////////////////
//{{NO_DEPENDENCIES}}
// Microsoft Developer Studio generated include file.
// Used by MDI.rc
#define IDD_ABOUTBOX                    100
#define IDM_BLACK                       101
#define IDM_WHITE                       102
#define IDM_RED                         103
#define IDM_BLUE                        104
#define IDM_GREEN                       105
#define IDR_MAINFRAME                   128
#define IDR_MDITYPE                     129
#define IDR_MDITEXT                     129
#define IDI_MDITEXT                     130
#define ID_FILE_TEXT                    32771
// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
```

```
#define _APS_3D_CONTROLS                1
#define _APS_NEXT_RESOURCE_VALUE        131
#define _APS_NEXT_COMMAND_VALUE         32787
#define _APS_NEXT_CONTROL_VALUE         1000
#define _APS_NEXT_SYMED_VALUE           106
#endif
#endif

/////////////////////////////////////////////////////////////////////////////
// Файл MDI.rc
// Copyright (c) 1999 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////////////////////////
//Microsoft Developer Studio generated resource script.
//
#include "resource.h"
#define APSTUDIO_READONLY_SYMBOLS
/////////////////////////////////////////////////////////////////////////////
// Generated from the TEXTINCLUDE 2 resource.
  #include "afxres.h"
#undef APSTUDIO_READONLY_SYMBOLS
/////////////////////////////////////////////////////////////////////////////
// Russian resources
#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_RUS)
#ifdef _WIN32
LANGUAGE LANG_RUSSIAN, SUBLANG_DEFAULT
#pragma code_page(1251)
#endif //_WIN32
#ifdef APSTUDIO_INVOKED
// TEXTINCLUDE
//
1 TEXTINCLUDE DISCARDABLE
BEGIN
  "resource.h\0"
END
2 TEXTINCLUDE DISCARDABLE
BEGIN
  "#include ""afxres.h""\r\n"
  "\0"
END
```

```
3 TEXTINCLUDE DISCARDABLE
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 9, 1\r\n"
  "#pragma code_page(1252)\r\n"
  "#endif\r\n"
  "#include ""res\\MDI.rc2""  // non-Microsoft Visual C++ edited
                              // resources\r\n"
    "#include ""afxres.rc""   // Standard components\r\n"
    "#endif\0"
END
#endif    // APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////
// Icon
// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDR_MAINFRAME           ICON    DISCARDABLE     "res\\MDI.ico"
IDI_MDITEXT             ICON    DISCARDABLE     "res\\mditext.ico"
/////////////////////////////////////////////////////////////
// Menu
IDR_MAINFRAME MENU PRELOAD DISCARDABLE
BEGIN
  POPUP "Файл"
  BEGIN
    MENUITEM "Новое окно\t",                ID_FILE_TEXT
    MENUITEM SEPARATOR
    MENUITEM "Выход",                       ID_APP_EXIT
  END
    POPUP "?"
    BEGIN
    MENUITEM "О программе ...",             ID_APP_ABOUT
  END
END
```

```
IDR_MDITEXT MENU PRELOAD DISCARDABLE
BEGIN
  POPUP "Файл"
    BEGIN
     MENUITEM "Новое окно",                  ID_FILE_TEXT
     MENUITEM SEPARATOR
     MENUITEM "Выход",                       ID_APP_EXIT
  END
  POPUP "Цвет"
  BEGIN
        MENUITEM "Черный",                      IDM_BLACK
        MENUITEM "Белый",                       IDM_WHITE
        MENUITEM "Красный",                     IDM_RED
        MENUITEM "Синий",                       IDM_BLUE
        MENUITEM "Зеленый",                     IDM_GREEN
    END
    POPUP "Окно"
    BEGIN
        MENUITEM "Каскадом",                    ID_WINDOW_CASCADE
        MENUITEM "По горизонтали",              ID_WINDOW_TILE_HORZ
        MENUITEM "Упорядочить значки",          ID_WINDOW_ARRANGE
    END
    POPUP "?"
    BEGIN
        MENUITEM "&О программе ...",            ID_APP_ABOUT
  END
END
/////////////////////////////////////////////////////////
// Dialog
IDD_ABOUTBOX DIALOG DISCARDABLE  0, 0, 217, 55
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "About MDI"
FONT 8, "MS Sans Serif"
BEGIN
  ICON            IDR_MAINFRAME,IDC_STATIC,7,7,18,20
  CTEXT           "MDI-приложение Версия 1.0",IDC_STATIC,51,17,119,8,
                  SS_NOPREFIX
  CTEXT           "Copyright © 1999",IDC_STATIC,127,34,71,8
  DEFPUSHBUTTON   "OK",IDOK,178,7,32,14,WS_GROUP
```

```
  LTEXT           "Мешков А., Тихомиров Ю.",IDC_STATIC,35,34,89,8
END
#ifndef _MAC
/////////////////////////////////////////////////////////////
// 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 "041904e3"
    BEGIN
      VALUE "Comments", "Пример создания MDI-приложения\0"
      VALUE "CompanyName", "Pragma Design Ltd.\0"
      VALUE "FileDescription", "MDI-приложение на базе MFC\0"
      VALUE "FileVersion", "1, 0, 0, 1\0"
      VALUE "InternalName", "MDI\0"
      VALUE "LegalCopyright", "Copyright © 1999\0"
      VALUE "OriginalFilename", "MDI.EXE\0"
      VALUE "ProductName", "MDI-приложение\0"
      VALUE "ProductVersion", "1, 0, 0, 1\0"
    END
  END
  BLOCK "VarFileInfo"
  BEGIN
    VALUE "Translation", 0x419, 1251
  END
END
```

```
#endif    // !_MAC
/////////////////////////////////////////////////////////////////////////////
// DESIGNINFO
#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
  IDD_ABOUTBOX, DIALOG
  BEGIN
    LEFTMARGIN, 7
    RIGHTMARGIN, 210
    TOPMARGIN, 7
    BOTTOMMARGIN, 48
  END
END
#endif    //
APSTUDIO_INVOKED/////////////////////////////////////////////////////
// Cursor
IDC_MDITEXT             CURSOR  DISCARDABLE     "res\\text.cur"
/////////////////////////////////////////////////////////////////////////////
// String Table
STRINGTABLE PRELOAD DISCARDABLE
BEGIN
  IDR_MAINFRAME         "Главное окно MDI-приложения"
END
STRINGTABLE DISCARDABLE
BEGIN
  ID_APP_ABOUT         "Вывод информации о программе и ее версии\nO"
  ID_APP_EXIT          "Завершение работы приложения"
END
STRINGTABLE DISCARDABLE
BEGIN
  ID_NEXT_PANE          "Переход к следующему окну"
  ID_PREV_PANE          "Переход к предыдущему окну "
END
STRINGTABLE DISCARDABLE
BEGIN
  ID_WINDOW_ARRANGE     "Упорядочить значки"
  ID_WINDOW_CASCADE     "Расположить окна с перекрытием"
  ID_WINDOW_TILE_HORZ   "Расположить окна без перекрытия"
```

```
END
STRINGTABLE DISCARDABLE
BEGIN
  ID_FILE_TEXT          "Открыть новое окно"
END
#endif    // Russian resources
#ifndef APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////////////////////
// Generated from the TEXTINCLUDE 3 resource.
#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 9, 1
#pragma code_page(1252)
#endif
#include "res\MDI.rc2"  // non-Microsoft Visual C++ edited resources
#include "afxres.rc"         // Standard components
#endif
#endif    // not APSTUDIO_INVOKED
```

Рис. 10.2. Главное и дочерние окна MDI-приложения

Первое, что сразу бросается в глаза при рассмотрении программы, это практически полная идентичность интерфейса и реализации нашего класса

"приложение" (CMDIApp) и CFirstApp, описанного выше. А если не считать появления специального обработчика команды, который мы рассмотрим дальше в этой главе, и вызова функции CWnd::FindWindow, то совпадение будет полным. На самом деле это совсем не случайно. Достаточно большая часть программ для Windows создается с использованием метода "клея и ножниц", т. е. взяв за основу некоторый шаблон (а такие заготовки имеются у всех, даже начинающих, программистов), вы выполняете несколько маленьких изменений и потом забываете об этом.

Примечание

В следующих примерах мы не будем приводить полные листинги программ, а ограничимся обсуждением только тех изменений, которые появляются в программе по сравнению с имеющейся заготовкой.

После этого небольшого отступления рассмотрим фрагмент кода:

```
CString csCaption;
csCaption.LoadString(IDR_MAINFRAME);
CWnd *pWnd = CWnd::FindWindow(NULL, csCaption);
if(pWnd != NULL)
{
  if(pWnd->IsIconic())
    pWnd->ShowWindow(SW_RESTORE);
  pWnd->SetForegroundWindow();
  return FALSE;
}
```

Не будем пока акцентировать внимание на новом для нас классе CString, а ограничимся только замечанием, что в объекте этого класса хранится символьная строка с именем окна, загруженная из файла ресурсов.

Мы вставили этот код для того, чтобы не загружать наше приложение еще раз, если оно уже было запущено. Действительно, MDI-приложения позволяют работать с несколькими документами одновременно (в отличие от SDI-приложений), и нет никакой необходимости расходовать дополнительную память. Как уже говорилось, для проверки того, не было ли приложение уже загружено, мы не можем использовать параметр hInstance, передаваемый в командной строке, т. к. для приложений Win32 он всегда равен NULL. Поэтому мы воспользовались функцией CWnd::FindWindow, которая позволяет найти окно верхнего уровня по имени оконного класса или по заголовку окна (ее нельзя использовать для поиска дочерних окон):

static CWnd* CWnd::**FindWindow**(
  LPCTSTR lpszClassName,
  LPCTSTR lpszWindowName) —

через параметр lpszClassName передается указатель на символьную строку, которая определяет имя класса искомого окна. Если этот параметр равен NULL, как в нашем случае, то поиск будет осуществляться во всех классах. Поскольку нам надо найти вполне определенное окно с заданным заголовком, то имя заголовка мы передаем через второй параметр lpszWindowName, который в общем случае также может быть равен NULL.

Если окно найдено, то функция FindWindow вернет временный указатель на этот объект класса CWnd, в противном случае — NULL.

Примечание

Если назначение имени класса мы отдаем на откуп библиотеке классов MFC, то параметр lpszClassName использовать просто невозможно, поскольку имя класса генерируется каждый раз новое и имеет примерно такой вид: "Afx:400000:8:1526:0:13F7" (при следующем запуске, даже второй копии приложения (если предусмотрен такой режим), класс будет иметь имя "Afx:400000:8:1526:0:251F" — почувствуйте разницу!).

После того как главное окно найдено, нужно проверить, не было ли оно свернуто. Для этого используется специальная функция:

BOOL CWnd::**IsIconic**( ) —

возвращает TRUE, если окно свернуто.

В этом случае восстанавливаем его нормальный размер с помощью уже известной вам функции CWnd::ShowWindow.

Теперь осталось только поместить окно на передний план и активизировать. Это легко сделать при помощи функции, которая предназначена именно для подобных целей:

BOOL CWnd::**SetForegroundWindow**( ) —

переводит поток, который создал окно, в приоритетный режим и активизирует его главное окно.

Ее дополняет функция:

static CWnd* CWnd::**GetForegroundWindow**( ) —

возвращает указатель на временный объект приоритетного окна (окна, с которым пользователь в текущий момент работает).

Для того чтобы картина была полной, приведем еще одну функцию, которая позволяет изменить Z-порядок окон:

void CWnd::**BringWindowToTop**( ) —

позволяет вывести на передний план окна верхнего уровня, всплывающие или окна "MDI child". По своему действию похожа на функцию CWnd::SetWindowpPos, однако не позволяет изменять стили окна.

Возвращаясь к нашему примеру, по этому поводу осталось сказать, что если была найдена работающая копия нашего приложения, то остается только

завершить работу текущей копии. Если же работающая копия не найдена, то переопределенная функция InitInstance приступает к инициализации приложения. Этот процесс уже был подробно описан, и мы не будем к нему возвращаться.

Займемся лучше используемыми классами окон. Прежде всего, обратим внимание на базовый класс нашего главного окна:

```
class CMainFrame : public CMDIFrameWnd
{
  ...
};
```

Как видите, он образован из специально спроектированного для окон MDI класса. При этом действия по созданию главного окна MDI-приложения точно такие же, как и при создании любого другого окна — создание объекта "окно", регистрация оконного класса и создание окна Windows. Чтобы не повторяться, коснемся только одного момента. Нет строгих правил, определяющих место, где следует регистрировать оконный класс. Как правило, большие классы регистрируют в отдельном модуле, а маленькие — в той же самой функции, что и основное окно. Однако это дело вкуса, и вы вправе разрабатывать или перенимать любой стиль, который вас устраивает. Поскольку мы собираемся работать со многими окнами "MDI child" одновременно, и, в данном случае, все они принадлежат одному оконному классу, то в качестве места регистрации их оконного класса мы выбрали обработчик сообщения WM_CREATE главного окна приложения:

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct){
 if (CMDIFrameWnd::OnCreate(lpCreateStruct) == -1)
  return -1;
  lpszTextClass = AfxRegisterWndClass(
    CS_HREDRAW | CS_VREDRAW,
    AfxGetApp()->LoadCursor(IDC_MDITEXT),
    (HBRUSH) (COLOR_WINDOW+1),
    AfxGetApp()->LoadIcon(IDI_MDITEXT));
  OnText();
  return 0;
}
```

Помимо регистрации оконного класса для окон "MDI child", в этот обработчик мы включили явный вызов обработчика команды OnText для создания и вывода на экран первого дочернего окна MDI-приложения. Об этом чуть позже, а пока обратим ваше внимание на то, что нам не надо явно создавать специальное окно MDICLIENT, т. к. оно автоматически создается функциями библиотеки. Вы, конечно, можете вмешаться в этот процесс, пере-

определив функцию класса CMDIFrameWnd — CreateClient. Однако еще раз подчеркнем: чтобы не нарушать правильную работу составных частей вашего приложения, не забывайте в таких случаях вызывать переопределенную функцию базового класса.

В этом месте необходимо сделать небольшое отступление и рассмотреть вопросы создания и обработки команд меню.

Примечание

Более подробно меню и другие ресурсы мы рассмотрим в отдельной главе.

Обработка команд меню

Код, создающий основное меню для нашей программы, достаточно прост:

```
IDR_MAINFRAME MENU PRELOAD DISCARDABLE
BEGIN
POPUP "Файл"
  BEGIN
    MENUITEM "Новое окно\t",                ID_FILE_TEXT
    MENUITEM SEPARATOR
    MENUITEM "Выход",                       ID_APP_EXIT
  END
    POPUP "?"
    BEGIN
    MENUITEM "О программе ...",             ID_APP_ABOUT
  END
END
```

Прежде всего, определяем идентификатор меню — IDR_MAINFRAME, который передаем в функцию LoadFrame для регистрации оконного класса и создания окна Windows. Само тело меню заключено между операторными скобками BEGIN и END. Как видно из текста, элементы меню бывают двух видов: POPUP — это целое окно (например, раскрывающийся список, который появляется, когда вы открываете меню **File** (файл) во многих приложениях); MENUITEM — это единичный элемент меню, после объявления которого в кавычках записана символьная строка, изображаемая в строке меню. И, наконец, объявляется константа, которая ассоциирована с данным пунктом меню. Обратите внимание, что *каждый* пункт меню ассоциируется с некоторой константой. Именно они и представляют для нас сейчас основной интерес. Дело в том, что когда пользователь выбирает конкретный пункт меню, соответствующая константа и пункт меню упаковываются в параметры сообщения WM_COMMAND и посылаются в соответствующую оконную проце-

дуру. Связав с этой константой определенный обработчик сообщений, мы можем выполнить любые необходимые нам действия. Например, для того чтобы создать новое окно "MDI child" и вывести его на экран, мы создали обработчик сообщения OnText и связали его посредством карты сообщений с командой меню ID_FILE_TEXT, аналогично тому, как были связаны между собой сообщение WM_CREATE и обработчик OnCreate:

```
protected:
  //{{AFX_MSG(CMainFrame)
  afx_msg void OnText();
  afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()
...
BEGIN_MESSAGE_MAP(CMainFrame, CMDIFrameWnd)
  //{{AFX_MSG_MAP(CMainFrame)
  ON_COMMAND(ID_FILE_TEXT, OnText)
  ON_WM_CREATE()
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()
...
void CMainFrame::OnText()
{
  CTextWnd *pTextWnd = new CTextWnd;
  if (!pTextWnd->Create(lpszTextClass,
                        _TEXT("Дочернее окно \"MDI child\""),
                        0, rectDefault, this))
  return;
}
```

Непосредственный процесс создания окна вам уже хорошо знаком. Поэтому обратим ваше внимание только на следующую строку:

```
_TEXT("Дочернее окно \"MDI child\"").
```

Интерес здесь представляет макрос _TEXT, содержащийся во включаемом файле TCHAR.H. Чтобы ответить на вопрос "для чего он нужен?", сделаем еще одно небольшое отступление.

Unicode

Unicode — стандарт, разработанный в 1988 году фирмами Apple и Xerox. Его полное описание содержится в *The Unicode Standard: Worldwide Character*

Encoding, Version 1.0. Нас же будет интересовать общее понимание того, что же такое Unicode.

Поскольку система Windows становится все популярнее, то разработчикам приложений для нее надо ориентироваться на международный рынок. Естественно, что при этом встает вопрос о поддержке самых разных языков. А ведь существуют и такие, что возможностей одного байта, позволяющего кодировать не более 256 символов, недостаточно для представления всех букв этого языка. Для поддержки подобных языков были созданы двухбайтовые наборы символов (double-byte character sets, DBCS). Однако работать с ними оказалось очень неудобно, т. к. для представления одних символов достаточно одного байта, а для других необходимо два.

В отличие от DBCS, всем символам в Unicode требуется ровно два байта, что позволяет кодировать около 65 000 символов — более чем достаточно для работы с любым языком. В настоящее время *кодовые позиции* (code points) определены для арабского, китайского, русского (кириллица), греческого, еврейского, латинского (английского), японской каны и некоторых других алфавитов. Кроме того, в набор символов включено большое количество знаков препинания, математических, технических, диакритических и других символов.

И хотя пока Unicode поддерживается только в системе Windows NT и не поддерживается в Windows 95/98, для того чтобы упростить задачу локализации программного продукта в будущем, мы советуем вам уже сейчас включать поддержку Unicode в ваши программные проекты. Тем более, что это совсем несложно, ведь фирма Microsoft разработала Win32 API для Unicode так, чтобы как можно меньше повлиять на ваш код. Для переключения на Unicode надо всего лишь определить два макроса и перекомпилировать исходный текст. Естественно, это возможно при условии, что вы вставили в вашу программу файл заголовков TCHAR.H и используете макросы из этого файла вместо стандартных функций для работы со строками, такими, например, как strlen. Полный перечень макросов вы найдете в файле TCHAR.H. Но это еще не все. Для задания массива символов ANSI/Unicode следует использовать тип данных TCHAR. При этом, если определен тип _UNICODE, то TCHAR объявляется так:

```
typedef wchar_t TCHAR;
```

а если не определен, то так:

```
typedef char TCHAR;
```

При использовании этого типа данных строка символов объявляется так:

```
TCHAR szString[128];
```

Кроме этого, для корректной работы со строками необходимо использовать макрос _TEXT (или _T), который определен так:

```
#define _TEXT(x) L##x     // для Unicode
```

и

```
#define _TEXT(x) x        // для ANSI
```

что мы и продемонстрировали в нашей программе.

В заключение просто перечислим шаги, которые необходимо проделать, чтобы подготовить приложение к использованию Unicode:

- Задавайте текстовые строки массивами символов, а не байтов (char)
- Используйте настраиваемые типы данных (вроде TCHAR или LPTSTR) для символов и текстовых строк
- Используйте макрос _TEXT для задания символьных или строковых литералов
- Для избежания каких-либо проблем в строковой арифметике используйте выражение (sizeof(szBuffer) / (sizeof(TCHAR)) вместо sizeof(szBuffer)

Вот, пожалуй, и все, что мы пока хотели сказать о Unicode, и можно вернуться к нашему примеру.

Обработка команд меню (продолжение)

Как мы уже отмечали, для того чтобы подключить меню окна "MDI child" к основному окну MDI-приложения, мы определили член menu класса CTextWnd:

```
class CTextWnd : public CMDIChildWnd
{
...
protected:
  static CMenu menu;  // Меню для всех окон "MDI child"
...
};
```

Теперь остается только присвоить идентификатор меню окна "MDI child" специальному члену класса:

```
BOOL CTextWnd::PreCreateWindow(CREATESTRUCT& cs)
{
...
  if (menu.m_hMenu == NULL)
    menu.LoadMenu(IDR_MDITEXT);
  m_hMenuShared = menu.m_hMenu;
...
}
```

Все остальные заботы по отслеживанию команд меню взяла на себя операционная система. Нам осталось только правильно их обработать. Если вы внимательно посмотрите на карту сообщений класса CTextWnd, например, для меню "Цвет":

```
BEGIN_MESSAGE_MAP(CTextWnd, CMDIChildWnd)
  //{{AFX_MSG_MAP(CTextWnd)
...
  ON_COMMAND(IDM_BLACK, OnColor)
  ON_COMMAND(IDM_BLUE, OnColor)
  ON_COMMAND(IDM_GREEN, OnColor)
  ON_COMMAND(IDM_RED, OnColor)
  ON_COMMAND(IDM_WHITE, OnColor)
  ON_UPDATE_COMMAND_UI(IDM_BLACK, OnUpdateColor)
  ON_UPDATE_COMMAND_UI(IDM_BLUE, OnUpdateColor)
  ON_UPDATE_COMMAND_UI(IDM_GREEN, OnUpdateColor)
  ON_UPDATE_COMMAND_UI(IDM_RED, OnUpdateColor)
  ON_UPDATE_COMMAND_UI(IDM_WHITE, OnUpdateColor)
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

то увидите, что для всех пяти команд меню мы использовали один и тот же обработчик сообщений — OnColor, возложив на него задачу идентификации этих команд:

```
void CTextWnd::OnColor()
{
  nIDColor = LOWORD(GetCurrentMessage()->wParam);
  clrText = colorArray[nIDColor — IDM_BLACK];
  Invalidate();
}
```

Примечание

Другой способ использования одного обработчика для нескольких команд, основанный на применении макроса ON_COMMAND_RANGE, представлен в примере главы 15.

Использованная в этом коде функция CWnd::GetCurrentMessage возвращает указатель на структуру MSG, в слове wParam которой содержится идентификатор выбранного пункта меню (идентификатор команды). Из файла RESOURCE.H:

```
...
#define IDM_BLACK                    101
```

```
#define IDM_WHITE                    102
#define IDM_RED                      103
#define IDM_BLUE                     104
#define IDM_GREEN                    105
...
```

видно, что используемые в данном фрагменте идентификаторы команд идут один за другим. Это не случайно и позволяет нам использовать их для определения индекса в массиве цветов (nIDColor — IDM_BLACK). Попробуйте изменить числовое значение у какого-либо идентификатора — программа сразу же перестанет адекватно реагировать на эту команду.

Последней в этом обработчике вызывается функция CWnd::Invalidate. Нам представляется важным познакомиться с ней поближе:

void CWnd::**Invalidate**(BOOL bErase = TRUE) —

определяет в качестве области модификации всю рабочую область окна как требующую перерисовки. В результате приложению будет послано сообщение WM_PAINT. Это сообщение Windows посылает всякий раз, когда область модификации окна непуста, и для этого окна нет других сообщений в очереди. Параметр, передаваемый в функцию, определяет необходимость стирания фона окна перед перерисовкой. Если он имеет значение TRUE, фон стирается, если FALSE — нет.

Примечание

Для того чтобы отметить не всю рабочую область, а какую-нибудь ее часть, в классе CWnd реализованы специальные функции InvalidateRect и InvalidateRgn, первая из которых "помечает" прямоугольную область, а вторая — произвольную.

И хотя мы вплотную подошли к вопросу обработки сообщения WM_PAINT, задержимся еще для того, чтобы рассмотреть принцип действия специальной команды обновления и связанный с ней класс CCmdUI.

Вернемся к нашей программе. Нашей задачей было поставить "галочку" у того элемента меню, который является выбранным на текущий момент времени, что мы и сделали, воспользовавшись функцией CCmdUI::SetCheck в обработчике OnUpdateColor. Поскольку этот обработчик каждый раз вызывается для всех элементов раскрывающегося меню, то перед перерисовкой "галочка" будет установлена только у того, который был перед этим выбран (pCmdUI->m_nID == nIDColor), а у остальных — убрана:

```
void CTextWnd::OnUpdateColor(CCmdUI* pCmdUI)
{
  pCmdUI->SetCheck(pCmdUI->m_nID == nIDColor);
}
```

Обработка сообщения WM_PAINT

Сообщение, которое мы рассмотрим, — одно из наиболее важных, если не самое важное, поскольку приложение, как правило, выполняет всю работу по рисованию именно во время обработки этого сообщения. Оно посылается в окно всякий раз, когда требуется изменить содержимое клиентской области, т. е. того пространства окна, о котором программист заботится самостоятельно, в то время как Windows обслуживает остальные его части. Например, если окно было "спрятано" под другим окном и выводится на передний план, то оно получает сообщение WM_PAINT, т. к. необходимо перерисовать область, скрытую под другим окном.

Рассмотрим обработчик этого сообщения, реализованный в нашей программе:

```
void CTextWnd::OnPaint()
{
  CPaintDC dc(this);
  dc.SetTextColor(clrText);
  dc.SetBkColor(::GetSysColor(COLOR_WINDOW));
  CRect rect;
  GetClientRect(rect);
  dc.DrawText(_TEXT("Обработка сообщения WM_PAINT"), -1, rect,
    DT_SINGLELINE | DT_CENTER | DT_VCENTER);
}
```

Как вы помните, любой вывод в программах, написанных для Windows, осуществляется в контекст устройства, который позволяет абстрагироваться от конкретного типа внешнего устройства. Здесь еще не место и не время подробно рассматривать поддержку контекстов устройств, имеющуюся в библиотеке MFC, — этому вопросу будет посвящен отдельный большой раздел. Однако ввиду важности сообщения WM_PAINT и приняв в качестве постулата необходимость наличия специального контекста для окна, рассмотрим обработчик этого сообщения достаточно подробно.

Первая же строка обработчика содержит код для создания объекта класса CPaintDC, который предоставляет контекст дисплея для обработки сообщения WM_PAINT.

Конструктор этого объекта в качестве параметра получает указатель на оконный объект, для которого запрашивается контекст, автоматически вызывает функцию CWnd::BeginPaint, обеспечивающую доступ к нему, и сохраняет структуру PAINTSTRUCT в специальной переменной m_ps. После окончания работы с контекстом устройства его необходимо вернуть в систему во избежание истощения ресурсов. Для этого в деструкторе автоматически вызывается функция CWnd::EndPaint. Между этими двумя процедурами располагается сердцевина обработчика сообщения WM_PAINT.

Наш обработчик демонстрирует лишь один из основных способов вывода текста в Windows-программах. Он основан на использовании функции DrawText:

virtual int CDC::**DrawText**(
LPCTSTR lpszStr,
int nCount,
LPRECT lpRect,
UINT nFormat) —

lpszStr — указатель на выводимую строку, nCount — длина строки (может быть равна -1 для ASCIIZ-строк, т. е. строк заведомо оканчивающихся нуль-символом), lpRect — область, внутри которой выводится строка, nFormat — задает метод форматирования текста.

При успешном завершении функция возвращает значение высоты текста. В нашем примере для форматирования мы использовали центрирование по вертикали и горизонтали, а также вывод всего текста одной строкой. Эта функция использует контекст устройства, чтобы установить цвета для фона и текста. Для того чтобы задать свои значения этих параметров, мы воспользовались специальными функциями SetTextColor и SetBkColor, которые будут рассмотрены при подробном обсуждении графического интерфейса.

В нашем обработчике осталась еще одна функция. Поскольку она не имеет отношения к контексту устройств, опишем ее подробно:

void CWnd::**GetClientRect**(LPRECT lpRect) —

заполняет структуру (или объект) RECT, на которую указывает параметр lpRect, размерами рабочей области окна. При этом параметры top и left будут равны нулю, а right и bottom определяют ширину и высоту рабочей области.

Сама структура объявлена в файле WINDEF.H:

```
typedef struct tagRECT{
  LONG left;
  LONG top;
  LONG right;
  LONG bottom;
} RECT;
```

и определяет координаты левого верхнего (left — x-координата, top — y-координата) и правого нижнего (right — x-координата, bottom — y-координата) углов некоторого прямоугольника.

Ввиду того что Windows большей частью работает с прямоугольными областями, в библиотеке MFC определен специальный класс CRect, который похож на только что рассмотренную структуру RECT, но, кроме этого, включает специальные функции для манипулирования объектами этого класса. Мы не будем рассматривать эти функции, т. к. их использование достаточно тривиально, а перечень и описание легко можно посмотреть в Help. То же самое относится к удобным и часто используемым классам CSize и CPoint.

Последнее, что хотелось бы сказать по поводу сообщения WM_PAINT. Проделайте такой эксперимент. Добавьте в программу обработчик нажатия левой кнопки мыши:

```
void CTextWnd::OnLButtonDown(UINT nFlags, CPoint point)
{
  CClientDC dc(this);
  TCHAR strText[] = "Обработка сообщения WM_LBUTTONDOWN";
  dc.SetTextColor(clrText);
  dc.SetBkColor(::GetSysColor(COLOR_WINDOW));
  dc.TextOut(10, 10, strText, strlen(strText));
  CMDIChildWnd::OnLButtonDown(nFlags, point);
}
```

Еще раз подчеркнем — в Windows для проведения любых операций, связанных с выводом, *необходимо* сначала получить контекст устройства. Как видите, здесь для получения контекста при выводе текста мы использовали другой класс — CClientDC. Кроме того, для вывода текста мы воспользовались еще одной имеющейся в библиотеке функцией — TextOut:

virtual BOOL CDC::**TextOut**(
  int x,
  int y,
  LPCTSTR lpszString,
  int nCount) —

> выводит в контекст устройства, начиная с точки (x,y), строку lpszString длиной nCount. Все остальные параметры, такие как шрифт, цвета фона и текста и т. д. берутся из контекста устройства.

Для того чтобы понять разницу в использовании CClientDC и CPaintDC, вам достаточно перетащить одно дочернее окно поверх другого, в котором предварительно был изображен текст. Если после этого вы вновь поместите первое окно на передний план, щелкнув на его заголовке, то увидите примерно то, что изображено на рис. 10.3.

Если вы отчетливо понимаете, что произошло, то вы действительно вникли в суть механизмов, задействованных в системе сообщений Windows. Сообщение WM_PAINT посылается в окно каждый раз, когда открывается его часть, временно закрытая другим окном. Таким образом, текст "Обработка сообщения WM_PAINT" будет виден всегда, когда видимо окно, просто потому, что Windows гарантирует посылку сообщения WM_PAINT в нужный момент. Однако любой текст, выводимый обработчиком OnLButtonDown, не будет перерисовываться вновь. Причина заключается в том, что в этот обработчик посылается только сообщение WM_LBUTTONDOWN, и он "ничего не знает" о сообщении WM_PAINT и не обращает на него внимания.

Рис. 10.3. Пример вывода текста в рабочую область окна с использованием и без использования обработчика сообщения WM_PAINT

В заключение скажем несколько слов о сообщениях, обрабатываемых окнами "MDI child" по умолчанию. К ним относятся: WM_MOVE, WM_SETFOCUS, WM_CHILDACTIVATE, WM_GETMINMAXINFO, WM_MENUCHAR, WM_SIZE, WM_NEXTMENU и WM_SYSCOMMAND. И если вы перехватываете какое-либо из них, то во избежание неприятностей не забывайте вызывать соответствующий обработчик базового класса.

ЭЛЕМЕНТЫ ИНТЕРФЕЙСА ПОЛЬЗОВАТЕЛЯ

Глава 11

Панели элементов управления

В главном окне любого современного профессионально написанного приложения имеется как минимум два дополнительных инструмента, значительно облегчающих работу, — это *панель инструментов* (Toolbar) и *строка состояния* (Statusbar). И хотя в библиотеке MFC для этих, а также некоторых других элементов давно реализованы соответствующие классы, только 32-разрядные операционные системы Windows 95/98 и Windows NT обеспечивают поддержку этих элементов интерфейса пользователя. На самом деле, создать панель элементов управления не сложнее, чем обыкновенное окно, тем более используя классы MFC. Пришло время познакомиться с этими, очень удобными и полезными компонентами любого современного приложения. Начнем с рассмотрения общих свойств и возможностей, характерных для всех подобных элементов.

Класс CControlBar

CControlBar — базовый класс для всех классов панелей элементов управления, включающих CStatusBar, CToolBar, CDialogBar, CReBar и COleResizerBar. Панель элементов управления, сама являясь окном, может содержать дочерние элементы, которые могут быть либо окнами Windows, способными отправлять и получать сообщения, либо не являться таковыми, т. е. управляться кодом приложения или фрейма. В качестве примера элементов первого типа можно назвать комбинированный список (COMBO BOX) и текстовое поле (EDIT), а второго типа — кнопки с рисунком (BITMAP BUTTONS) и области строки состояния (STATUS BAR PANES). Базовые функциональные возможности для всех типов панелей элементов управления похожи — все их объекты являются дочерними окнами некоторого родительского фрейма. Причем обычно они являются окнами одного уровня с окном MDICLIENT или представлением.

Панель элементов управления может быть *фиксированной* (docking) у любой стороны ее родительского фрейма или *плавающей* (floating), что определяется установкой соответствующего стиля. Помимо этого объекты CControlBar самостоятельно позиционируются, используя информацию о рабочей области своего родительского окна, после чего они изменяют ее размеры таким образом, чтобы представление или окно MDICLIENT заполнили оставшуюся часть рабочей области.

Класс содержит только одну общедоступную переменную:

BOOL CControlBar::**m_bAutoDelete** —

служит для индикации того, когда будет удаляться объект этого или производного класса — при разрушении объекта Windows (TRUE) или при разрушении объекта фрейма, владеющего этой панелью. Если объект этого класса создается динамически, то рекомендуется установить значение переменной равной TRUE с тем, чтобы родительский фрейм сам позаботился о его удалении.

Из функций выделим следующие:

int CControlBar::**GetCount**( ) —

возвращает число неоконных элементов объекта CControlBar, тип которых зависит от производного объекта — областей для объектов CStatusBar и кнопок или разделителей для объектов CToolBar. Для объекта CDialogBar функция возвращает 0.

virtual CSize CControlBar::**CalcFixedLayout**(
BOOL bStretch,
BOOL bHorz) —

определяет горизонтальный размер панели элементов управления. Параметр bHorz задает ориентацию панели (TRUE — горизонтальная, FALSE — вертикальная). Если bStretch равно TRUE, то панель элементов управления предварительно растягивается до размеров родительского фрейма и не может менять свое местоположение, а если FALSE, то изменения размеров не происходит, и она может быть как зафиксированной в фрейме, так и плавающей.

DWORD CControlBar::**GetBarStyle**( ) —

возвращает текущий стиль, установленный для панели элементов управления.

void CControlBar::**SetBarStyle**(DWORD dwStyle) —

устанавливает требуемый стиль панели элементов управления. При этом доступными являются нижеперечисленные стили, имеющие префикс CBRS_:

CBRS_ALIGN_TOP	Фиксация у верхней стороны рабочей области фрейма
CBRS_ALIGN_BOTTOM	Фиксация у нижней стороны рабочей области фрейма
CBRS_ALIGN_LEFT	Фиксация у левой стороны рабочей области фрейма
CBRS_ALIGN_RIGHT	Фиксация у правой стороны рабочей области фрейма
CBRS_ALIGN_ANY	Фиксация у какой-либо стороны рабочей области фрейма

CBRS_BORDER_TOP	Рамка очерчивает верхний край панели элементов управления, когда она видима
CBRS_ BORDER _BOTTOM	Рамка очерчивает нижний край панели элементов управления, когда она видима
CBRS_ BORDER _LEFT	Рамка очерчивает правый край панели элементов управления, когда она видима
CBRS_ BORDER _RIGHT	Рамка очерчивает левый край панели элементов управления, когда она видима
CBRS_FLOAT_MULTI	В единственном окне минифрейма могут располагаться несколько плавающих панелей элементов управления
CBRS_TOOLTIPS	Для панели элементов управления выводятся всплывающие подсказки (tool tips)
CBRS_FLYBY	текст сообщения обновляется одновременно со всплывающими подсказками

Кроме перечисленных, можно использовать специальные стили общих элементов управления:

CCS_ADJUSTABLE	Пользователь может изменять конфигурацию элемента управления
CCS_BOTTOM	Элемент управления должен быть расположен в нижней части рабочей области окна
CCS_TOP	Элемент управления должен быть расположен в верхней части рабочей области окна
CCS_NODIVIDER	В верхней части элемента управления не надо рисовать разделительную линию шириной 2 пикселя
CCS_NOHILITE	В верхней части элемента управления не надо рисовать разделительную линию шириной 1 пиксель
CCS_NOMOVEY	В ответ на сообщение WM_SIZE элемент управления может изменять свои размеры и перемещаться только горизонтально, а вертикальное положение меняться не будет. Не используйте этот стиль совместно со стилем CCS_NORESIZE
CCS_NOPARENTALIGN	Элемент управления не будет автоматически перемещаться в верхнюю или нижнюю часть родительского окна
CCS_NORESIZE	Приложение должно задать размеры элемента управления явным образом, т. к. размеры, заданные по умолчанию, не будут использоваться

Как видите, достаточно много стилей панелей элементов управления связано со способом ее размещения в родительском окне. Для управления этим процессом в классе реализованы специальные функции:

CFrameWnd* CControlBar::**GetDockingFrame**( ) —

служит для получения указателя на текущий фрейм, в котором зафиксирована ваша панель элементов управления.

BOOL CControlBar::**IsFloating**( ) —

возвращаемое значение позволяет определить состояние панели элементов управления: плавающая (TRUE) или фиксированная (FALSE).

void CControlBar::**EnableDocking**(DWORD dwStyle) —

разрешает привязать панель элементов управления к одной из сторон родительского фрейма, которая должна быть доступна для фиксирования. Если сторона, задаваемая параметром dwStyle, и сторона, разрешенная для фиксирования в фрейме, не совпадают, то панель элементов управления не может быть зафиксирована в родительском фрейме. Параметр dwStyle может принимать следующие значения: CBRS_ALIGN_TOP, CBRS_ALIGN_RIGHT, CBRS_ALIGN_LEFT, CBRS_ALIGN_BOTTOM, CBRS_ALIGN_ANY и CBRS_ALIGN_MULTI.

Поскольку возможность фиксации в родительском фрейме зависит "от желания" не только самой панели элементов управления, но и ее родителя, нам представляется необходимым здесь же рассмотреть функции класса CFrameWnd, которые отвечают за работу с панелями элементов управления:

void CFrameWnd::**SaveBarState**(LPCTSTR lpszProfileName)

и

void CFrameWnd::**LoadBarState**(LPCTSTR lpszProfileName) —

эти функции позволяют сохранять информацию о каждой панели элементов управления текущего фрейма в INI-файле или реестре и восстанавливать ее. При этом сохраняется и восстанавливается информация следующего рода: видимость, ориентация (горизонтальная или вертикальная), фиксация и положение.

void CFrameWnd::**ShowControlBar**(
CControlBar *pBar,
BOOL bShow,
BOOL bDelay) —

параметр bShow определяет видимость панели на экране: TRUE — показать, FALSE — скрыть панель элементов управления pBar. Значение параметра bDelay, равное TRUE, определяет, что перед выводом на экран нужна задержка.

void CFrameWnd::**EnableDocking**(DWORD dwDockStyle) —

разрешает фиксацию панели элементов управления у сторон(ы) фрейма, задаваемых(ой) в параметре dwDockStyle: CBRS_ALIGN_TOP, CBRS_ALIGN_BOTTOM, CBRS_ALIGN_RIGHT, CBRS_ALIGN_LEFT, CBRS_ALIGN_ANY. По умолчанию панели элементов управления фиксируются по сторонам фрейма в следующем порядке: верхняя, нижняя, левая и правая.

void CFrameWnd::**DockControlBar**(
CControlBar *pBar,
UINT nDockBarID = 0,
LPCRECT lpRect = NULL) —

привязывает панель элементов управления pBar к одной из сторон текущего фрейма, которая задается параметром nDockBarID и может быть нулем или комбинацией следующих значений:

AFX_IDW_DOCKBAR_TOP	Фиксация у верхней стороны фрейма
AFX_IDW_DOCKBAR_BOTTOM	Фиксация у нижней стороны фрейма

AFX_IDW_DOCKBAR_LEFT Фиксация у левой стороны фрейма

AFX_IDW_DOCKBAR_RIGHT Фиксация у правой стороны фрейма

Если этот параметр равен нулю, то сторона может быть любой из числа предварительно определенных при помощи двух функций: CControlBar::EnableDocking и CFrameWnd::EnableDocking. Параметр lpRect определяет, где (в координатах экрана) вне рабочей области фрейма будет зафиксировано окно, содержащее плавающую панель элементов управления.

CFrameWnd* CFrameWnd::**FloatControlBar**(
CControlBar *pBar,
CPoint point,
DWORD dwStyle = CBRS_ALIGN_TOP) —

переводит панель элементов управления pBar в плавающее состояние. Обычно вызывается при запуске приложения для восстановления предыдущих установок. Кроме того, эта функция вызывается фреймом, когда пользователь выполняет операцию отпускания (drop) после перемещения панели, не достигнув места, доступного для привязывания. Параметр point задает координаты точки, в которой будет находиться левый верхний угол панели элементов управления, а dwStyle определяет стиль ее выравнивания и ориентации внутри нового фрейма: CBRS_ALIGN_BOTTOM и CBRS_ALIGN_TOP — вертикальная ориентация, а CBRS_ALIGN_RIGHT и CBRS_ALIGN_ LEFT — горизонтальная. Если указаны оба типа ориентации, то панель будет ориентирована горизонтально.

virtual void CFrameWnd::**RecalcLayout**(BOOL bNotify = TRUE) —

перепозиционирует панели элементов управления текущего объекта. Параметр bNotify определяет, получит ли активизированный по месту элемент извещение (notify) при изменении размещения. Эта функция вызывается фреймом, когда переключаются стандартные панели элементов управления, или когда изменяется размер фрейма. Для того чтобы иметь возможность управлять отображением и поведением панелей управления, после того как изменилась компоновка фрейма, необходимо переопределить эту функцию в своем классе. Реализация этой функции, определенная по умолчанию, вызывает функцию CWnd::RepositionBars, чтобы перепозиционировать все панели элементов управления во фрейме.

Как уже говорилось, этот класс реализует основные функциональные возможности для всех панелей элементов управления. Более специфические и интересные возможности реализованы в его производных классах.

Примечание

Но прежде чем переходить к их рассмотрению, отметим еще один важный момент: в том случае, если кнопка панели элементов управления не имеет обработчиков для UPDATE_COMMAND_UI или COMMAND, то она автоматически блокируется фреймом.

Класс CToolBar

Этот класс определяет поведение *панели инструментов* — панели элементов управления, отображаемой полосой кнопок с рисунками (все они должны быть одного размера — стандартно 23×22 пикселя) и разделителей. Кнопки активизируют команды, т. к. нажатие кнопки панели инструментов подобно

выбору элемента меню. Так же как и элементы меню, кнопки панели инструментов могут действовать аналогично обычным кнопкам, переключателям и флажкам. Как и любая другая панель элементов управления, панель инструментов может быть привязана к какой-либо стороне родительского фрейма или быть плавающей. В последнем случае пользователь может изменять ее размер. Класс CToolBar поддерживает работу со всплывающей подсказкой, крошечным всплывающим окном, содержащим текстовую строку.

Для изображения всех кнопок панели инструментов создается один битовый массив, который должен содержать один и только один битовый рисунок (образ) для каждой кнопки. Обычно порядок расположения образов тот же, в котором они отображаются на экране. Все образы должны иметь один и тот же размер (стандартно 16×15 пикселей) и располагаться рядом друг с другом без промежутков. Изменение изображения кнопок панели инструментов в зависимости от их состояния и стиля (нажатая, заблокированная, неопределенная, отжатая и т. п.) формируется автоматически благодаря механизму обработки команды ON_UPDATE_COMMAND_UI: во время *цикла простоя* (idle time) панель инструментов вызывает обработчик команды ON_UPDATE_COMMAND_UI с идентификаторами команд для всех своих кнопок (исключая разделители). Для панелей инструментов класс CCmdUI, обеспечивающий правильное функционирование обработчика, поддерживает функции: CCmdUI::Enable, CCmdUI::SetCheck и CCmdUI::SetRadio.

(Подробно о классе CCmdUI см. главу 8)

Теперь, когда у вас сложилось общее представление о панели инструментов, мы, как обычно, рассмотрим основные функциональные возможности, предоставляемые классом CToolBar.

Поскольку панель инструментов — окно, для ее создания необходимо проделать стандартные шаги: создать объект класса и затем создать окно Windows на базе TOOLBARCLASSNAME — предопределенного оконного класса. Конструктор класса

CToolBar::**CToolBar**( ) —

создает объект класса и устанавливает размеры по умолчанию: размер кнопок — 23×22, размер образа — 16×15, высота верхней и нижней рамки — 3 пикселя.

Для создания окна Windows и присоединения его к объекту используется функция

```
BOOL CToolBar::Create(
    CWnd *pParentWnd,
    DWORD dwStyle = WS_CHILD | WS_VISIBLE | CBRS_TOP,
    UINT nID = AFX_IDW_TOOLBAR) —
```

pParentWnd — указатель на родительское окно панели инструментов, nID — идентификатор дочернего окна и dwStyle — стиль панели инструментов. Помимо общих стилей окон Windows и панелей элементов управления, она поддерживает дополнительные:

CBRS_TOP	Предписывает расположить панель вверху рабочей области родительского окна
CBRS_BOTTOM	Предписывает расположить панель внизу рабочей области родительского окна
CBRS_NOALIGN	Разрешает не переопределять размеры панели, если изменились родительские
CBRS_TOOLTIPS	Задает вывод подсказок для кнопок
CBRS_SIZE_DYNAMIC	Разрешает пользователю изменять размер панели инструментов при помощи мыши, когда панель находится в плавающем состоянии
CBRS_SIZE_FIXED	Запрещает изменять размер панели инструментов
CBRS_FLOATING	Задает плавающую панель инструментов
CBRS_FLYBY	Предписывает отображать информацию о кнопке в строке состояния
CBRS_HIDE_INPLACE	Предписывает сделать панель инструментов невидимой для пользователя

В библиотеке MFC версии 4.23 добавлена еще одна функция для создания панели инструментов —

BOOL CToolBar::**CreateEx**(
CWnd* pParentWnd,
DWORD dwCtrlStyle = TBSTYLE_FLAT,
DWORD dwStyle = WS_CHILD | WS_VISIBLE | CBRS_ALIGN_TOP,
CRect rcBorders = CRect(0, 0, 0, 0),
UINT nID = AFX_IDW_TOOLBAR) —

dwCtrlStyle — дополнительные стили для создания встроенного объекта класса CToolBarCtrl; rcBorders — объект класса CRect, позволяющий указать размер бордюра панели инструментов. Остальные параметры остались без изменения.

Примечание

Подробно стили объекта класса CToolBarCtrl будут описаны при рассмотрении соответствующего класса.

Эту функцию следует использовать вместо рассмотренной ранее, поскольку она позволяет одновременно создать и объект класса CToolBar библиотеки MFC, и встроенный в него соответствующий элемент управления TOOLBAR Windows.

После создания объекта осталось только загрузить образы кнопок из ресурсов. Для этого в классе предусмотрены две возможности: вызов функции LoadToolBar или загрузка образов кнопок при помощи функции LoadBitmap и привязка их к соответствующим командам функцией SetButtons. Рассмотрим эти функции более подробно:

BOOL CToolBar::**LoadToolBar**(LPCTSTR lpszResourceName)

и

BOOL CToolBar::**LoadToolBar**(UINT nIDResource) —

загружает изображение панели инструментов и ресурса, заданного либо именем (параметр lpszResourceName), либо идентификатором (параметр nIDResource). При этом сопоставление кнопок с образами и командами осуществляется автоматически. Visual C++, начиная с версий 4.x, поддерживает ресурс типа TOOLBAR, где не только задается вид кнопок, но также осуществляется привязка команд к ним.

Примечание

Там же можно задавать разделители и общие размеры кнопок, которые могут отличаться от стандартных.

BOOL CToolBar::**LoadBitmap**(LPCTSTR lpszResourceName)

и

BOOL CToolBar::**LoadBitmap**(UINT nIDResource) —

загружает битовый массив, определяемый параметром lpszResourceName или nIDResource, который должен содержать один образ для каждой кнопки панели инструментов. Если образы имеют нестандартный размер, то необходимо вызвать функцию SetSizes для установки правильных размеров кнопок и их образов.

BOOL CToolBar::**SetButtons**(
const UINT *lpIDArray,
int nIDCount) —

устанавливает идентификатор команды каждой кнопки панели инструментов в значение, определяемое соответствующим элементом массива lpIDArray, число элементов которого задается параметром nIDCount. Если элемент массива имеет предопределенное значение ID_SEPARATOR, то в соответствующей позиции панели инструментов создается разделитель. Если значение lpIDArray равно NULL, то функция просто отводит место для nIDCount элементов, а для установки атрибутов каждого элемента следует вызвать функцию SetButtonInfo. Эта функция устанавливает также для каждой кнопки стиль TBBS_BUTTON, а для каждого разделителя — стиль TBBS_SEPARATOR, и назначает каждой кнопке индекс образа. *Индекс образа* определяет его позицию внутри битового массива.

После того как панель инструментов создана, с ней можно начинать "общаться", и нам осталось рассмотреть предназначенные для этого функции:

void CToolBar::**SetSizes**(
SIZE sizeButton,
SIZE sizeImage) —

устанавливает размер (в пикселях) кнопок панели инструментов, задаваемый параметром sizeButton. При этом параметр sizeImage должен содержать размер образов, зафиксированный в соответствующем битовом массиве. Размеры кнопок должны задаваться исходя из размеров образов, плюс 7 пикселей для ширины и 6 — для высоты. Размеры самой панели инструментов функция устанавливает автоматически. Microsoft настаивает (это отражено в документе An Application Design Guide) на том, чтобы приложения содержали кнопки стандартных размеров, но любезно предоставляет возможность их изменять.

Примечание

Требования к пользовательскому интерфейсу не последние в ряду требований фирмы Microsoft к программным продуктам, претендующим на получение логотипа "сделано для Microsoft".

void CToolBar::**SetHeight**(int cyHeight) —

устанавливает новое значение высоты панели инструментов, определяемое параметром cyHeight. Если высота окажется слишком мала, то кнопки будут обрезаны снизу.

BOOL CToolBar::**SetBitmap**(HBITMAP hbmImageWell) —

задает для панели инструментов новый битовый массив, определяемый параметром hbmImageWell.

int CToolBar::**CommandToIndex**(UINT nIDFind) —

возвращает индекс первой кнопки (начиная с нуля), чей идентификатор команды совпадает с заданным в параметре nIDFind. Если такой идентификатор не найден, то возвращается значение –1.

UINT CToolBar::**GetItemID**(int nIndex) —

возвращает идентификатор команды кнопки или разделителя из позиции, задаваемой параметром nIndex. Если найден разделитель, функция возвращает значение ID_SEPARATOR.

virtual void CToolBar::**GetItemRect**(
int nIndex,
LPRECT lpRect) —

заполняет структуру lpRect координатами кнопки или разделителя, определяемой параметром nIndex. Координаты отсчитываются относительно левого верхнего угла панели инструментов. Эту функцию можно использовать для определения координат разделителя, который вы хотите заменить на комбинированный список или другой элемент управления.

UINT CToolBar::**GetButtonStyle**(int nIndex) —

возвращает стиль кнопки или разделителя, заданной номером позиции на панели инструментов nIndex.

void CToolBar::**SetButtonStyle**(
int nIndex,
UINT nStyle) —

позволяет установить стиль кнопки, заданной номером позиции nIndex. Стиль кнопки определяет ее реакцию на действия пользователя. Доступными являются следующие стили:

TBBS_BUTTON	Стандартная кнопка
TBBS_SEPARATOR	Разделитель между группами кнопок
TBBS_CHECKBOX	Кнопка с фиксацией: эта кнопка "залипает" при нажатии, для возврата в исходное состояние необходимо нажать ее еще раз

TBBS_GROUP	отмечает начало группы стандартных кнопок
TBBS_CHECKGROUP	Отмечает начало группы кнопок с фиксацией, остается нажатой до тех пор, пока не нажата другая кнопка из этой же группы

UINT CToolBar::**GetButtonInfo**(
int nIndex,
UINT nID,
UINT nStyle,
int iImage) —

позволяет получить идентификатор команды, стиль и индекс образа кнопки (начиная с 0), позиция которой задается параметром nIndex. Эти значения присваиваются переменным, на которые ссылаются соответственно nID, nStyle и iImage. Если запрашивается информация о разделителе, то в iImage заносится значение его ширины в пикселях.

void CToolBar::**SetButtonInfo**(
int nIndex,
UINT nID,
UINT nStyle,
int iImage) —

позволяет установить идентификатор команды nID, стиль nStyle и индекс образа iImage кнопки, позиция которой задается параметром nIndex. Для разделителя эта функция устанавливает его ширину в пикселях, значение которой передается в параметре iImage.

Помимо образа, каждой кнопке панели инструментов может быть поставлена в соответствие текстовая строка, которая появляется на ней.

Примечание

Это возможно только после дополнительных действий и выполнения многочисленных условий. Подробнее см. пример далее в этой главе.

Для определения текста этой строки можно воспользоваться следующими функциями:

void CToolBar::**GetButtonText**(
int nIndex,
CString &rString)

и

CString CToolBar::**GetButtonText**(int nIndex)

а также

BOOL CToolBar::**SetButtonText**(
int nIndex,
LPCTSRT lpszText)

Последняя функция класса CToolBar позволяет получить ссылку на объект CToolBarCtrl:

CToolBar& CToolBar::**GetToolBarCtrl**( ) —

позволяет воспользоваться функциональными возможностями общего элемента управления TOOLBAR, появившегося в Win32. Использование этого элемента управления позволяет, в частности, осуществлять более тонкую настройку панели инструментов.

Класс CToolBarCtrl

Для использования преимуществ общего элемента управления Windows TOOLBAR можно пойти двумя путями: либо непосредственно создать окно CToolBarCtrl, либо создать панель инструментов на основе класса CToolBar, а затем просто получить доступ к элементам объекта CToolBarCtrl, вызвав функцию этого класса CToolBar::CToolBarCtrl. Мы остановимся на втором способе и не будем рассматривать создание элемента управления TOOLBAR.

Любая кнопка панели инструментов имеет ассоциированную с ней структуру TBBUTTON, которая содержит полную информацию об этой кнопке. Описание структуры находится в файле COMMCTRL.H:

```
typedef struct _TBBUTTON {
  int iBitmap;
  int idCommand;
  BYTE fsState;
  BYTE fsStyle;
  DWORD dwData;
  int iString;
} TBBUTTON, NEAR* PTBBUTTON, FAR* LPTBBUTTON;
```

В поле iBitmap содержится номер кнопки (нумерация начинается с нуля). Для разделителя здесь должно быть записано нулевое значение.

В поле idCommand находится идентификатор команды, который будет передаваться окну-владельцу вместе с сообщением WM_COMMAND, когда пользователь выберет эту кнопку. Для разделителя здесь должно быть записано нулевое значение.

Поле fsState содержит комбинацию одного или нескольких флагов, определяющих состояние кнопки:

TBSTATE_CHECKED	Кнопка имеет стиль TBSTYLE_CHECKED — кнопка с фиксацией, и изображается в нажатом состоянии
TBSTATE_ENABLED	Кнопка доступна для действий пользователя; если это состояние не установлено, то кнопка заблокирована и изображается серым цветом
TBSTATE_HIDDEN	Кнопка невидима
TBSTATE_INDETERMINATE	Кнопка отображается серым цветом
TBSTATE_PRESSED	Кнопка изображается в нажатом состоянии

TBSTATE_WRAP	Кнопка, находящаяся в этом состоянии, сообщает о том, что все следующие кнопки будут отображаться на новой строке; использование этого флага позволяет создавать многострочные панели инструментов; этот флаг можно указывать только вместе с флагом TBSTATE_ENABLED

В поле fsStyle записывается стиль кнопки. Можно использовать комбинацию следующих стилей:

TBSTYLE_BUTTON	Стандартная кнопка
TBSTYLE_SEP	Разделитель, представляющий собой маленький промежуток между группами кнопок; пользователь не может взаимодействовать с кнопкой, имеющей такой стиль
TBSTYLE_CHECK	Кнопка с фиксацией: эта кнопка "залипает" при нажатии, для возврата в исходное состояние необходимо нажать ее еще раз
TBSTYLE_GROUP	Отмечает начало группы стандартных кнопок
TBSTYLE_CHECKGROUP	отмечает начало группы кнопок с фиксацией; кнопка, имеющая такой стиль, остается нажатой до тех пор, пока не нажата другая кнопка из этой же группы
TBSTYLE_DROPDOWN	Кнопка с раскрывающимся списком
TBSTYLE_AUTOSIZE	Ширина кнопок рассчитывается автоматически исходя из длины текста, а не из размеров рисунка
TBSTYLE_NOPREFIX	С кнопкой не ассоциирован никакой акселератор
TBSTYLE_TOOLTIPS	С кнопкой ассоциируется элемент управления "всплывающая подсказка"
TBSTYLE_WRAPABLE	Панель инструментов может иметь несколько линий кнопок
TBSTYLE_ALTDRAG	Пользователь может изменять расположение кнопок при нажатой клавише <Alt>. При этом для кнопок должен быть также установлен стиль CCS_ADJUSTABLE.
TBSTYLE_LIST	Текст кнопки располагается справа от ее битового образа
TBSTYLE_CUSTOMERASE	Родительскому окну будут посылаться уведомления об операциях drag-and-drop

Помимо знакомых, в версии Visual C++ 6 добавлены еще два, с помощью которых можно реализовать функциональные возможности, введенные в Microsoft Internet Explorer 4.0:

TBSTYLE_FLAT	Создаваемая панель инструментов будет иметь "плоский" вид
TBSTYLE_TRANSPARENT	Создаваемая панель инструментов будет "прозрачной"; текст кнопок показывается под рисунком

В поле dwData можно хранить дополнительные данные, связанные с конкретной кнопкой. Если это поле не используется, то лучше записать туда нулевое значение.

Если вы хотите написать на поверхности кнопки какой-либо текст, то в поле iString следует записать ее номер во внутреннем списке элемента управления. В противном случае здесь должно быть записано нулевое значение.

Полную информацию о любой кнопке панели инструментов можно получить при помощи функции

BOOL CToolBarCtrl::**GetButton**(
int nIndex,
LPTBBUTTON lpButton) —

кнопка, информация о которой записывается в структуру lpButton, задается номером nIndex.

Достаточно большая группа функций позволяет установить новое или узнать текущее состояние определенной кнопки этого элемента управления. Их действие настолько очевидно, что мы ограничимся только перечислением: IsButtonEnabled, IsButtonChecked, IsButtonPressed, IsButtonHidden, IsButtonIndeterminate, GetState, SetState, EnableButton, CheckButton, PressButton, HideButton, Indeterminate.

Если необходимо узнать общее число кнопок панели инструментов, то можно воспользоваться функцией

int CToolBarCtrl::**GetButtonCount**( )

Для смены окна, которое будет получать извещения от панели инструментов, необходимо сменить владельца этой панели. Это можно сделать при помощи функции

void CToolBarCtrl::**SetOwner**(CWnd *pWnd)

Как мы уже говорили, кнопки панели инструментов могут располагаться в несколько рядов. Для того чтобы узнать или изменить их число, в классе реализованы две функции

void CToolBarCtrl::**SetRows**(
int nRows,
BOOL bLarger,
LPRECT lpRect)

и

int CToolBarCtrl::**GetRows**( )

nRows — требуемое число строк; bLarger — определяет, будет ли увеличено (TRUE) или уменьшено (FALSE) число строк. Если панель инструментов не может изменить свой размер в соответствии с заданным числом строк, то она будет изменена на максимально возможное количество строк, естественно, меньшее, чем заданное в параметре nRow; lpRect — указатель на объект CRect или структуру RECT, в которую будут записаны новые размеры прямоугольника, ограничивающего панель инструментов. Число строк должно выбираться так, чтобы в каждой строке (может быть, за исключением последней) было одинаковое количество элементов. Обе эти функции могут быть вызваны только для панелей инструментов, которые созданы со стилем TBSTYLE_WRAPABLE.

Чтобы проверить, поддерживает ли дисплей "большие" битовые массивы, предназначена функция

UINT CToolBArCtrl::**GetBitmapFlags**( ) —

возвращает TBBF_LARGE, если "большие" битовые массивы можно использовать, и 0 в противном случае.

Следующая группа функций позволяет добавить новые битовые массивы или текстовые строки для кнопок:

int CToolBarCtrl::**AddStrings**(LPCTSTR lpszStrings)

и

int CToolBarCtrl::**AddString**(UINT nStringID)

int CToolBarCtrl::**AddBitmap**(
int nNumButtons,
UINT nBitmapID)

и

int CToolBarCtrl::**AddBitmap**(
int nNumButtons,
CBitmap *pBitmap)

При работе с панелью инструментов можно изменять ее внешний вид.

void CTollBatCtrl::**Customize**( ) —

выводит на экран специальное диалоговое окно для настройки панелей инструментов (Customize Toolbar). При этом сама панель инструментов должна иметь установленный флаг стиля CCS_ADJUSTABLE и обрабатывать специальные извещения настройки. Как это делается, описано ниже.

Можно сохранять параметры настройки в реестре и восстанавливать их из него:

void CToolBarCtrl::**SaveState**(
HKEY hKeyRoot,
LPCTSTR lpszSubKey,
LPCTSTR lpszValueName)

и

void CToolBarCtrl::**RestoreState**(
HKEY hKeyRoot,
LPCTSTR lpszSubKey,
LPCTSTR lpszValueName) —

hKeyRoot — идентификатор текущего открытого ключа в реестре или один из следующих предопределенных дескрипторов: HKEY_CLASSES_ROOT, HKEY_CURRENT_USER, HKEY_LOCAL_MACHINE и HKEY_USERS; lpszSubKey — указатель на символьную строку, которая содержит имя подключа, с которым ассоциируется значение; lpszValueName — указатель на символьную строку, содержащую имя сохраняемого или восстанавливаемого значения.

Если изменились размеры родительского окна или панели инструментов, то можно вызвать функцию:

void CToolBarCtrl::**AutoSize**( )

для автоматической установки новых размеров элемента управления.

Оставшиеся функции позволяют добавлять и удалять кнопки панели инструментов:

BOOL CToolBarCtrl::**AddButtons**(
int nNumButtons,
LPTBBUTTON lpButtons) —

добавляет nNumButtons кнопок к панели инструментов. Предварительно необходимо заполнить массив структур lpButtons полной информацией о кнопках. Этот массив должен иметь такое же число элементов, какое указано в параметре nNumButtons.

BOOL CToolBarCtrl::**InsertButton**(
int nIndex,
LPTBBUTTON lpButton) —

вставляет новую кнопку в панель инструментов слева от указанной параметром nIndex. Перед вызовом функции структура lpButton должна быть заполнена. Рисунок (текст) кнопки должен быть предварительно добавлен к списку панели инструментов при помощи функций AddBitmap, AddString или AddStrings.

BOOL CTollBarCtrl::**DeleteButton**(int nIndex) —

удаляет кнопку, заданную параметром nIndex, из панели инструментов.

Помимо рассмотренных функций в рассматриваемой версии библиотеки MFC появилось также несколько новых, таких как GetImageList, GetDisabledImageList, GetHotImageList, которые позволяют получить указатель на объект класса CImageList, хранящий изображения кнопок по умолчанию, заблокированных и подсвеченных. Некоторые другие функции, например, GetAnchorHighlight и SetAnchorHighlight предназначены, в основном, для работы в режиме Internet Explorer, где мы их и рассмотрим.

Последнее, что необходимо рассмотреть для полноценной работы с этим элементом управления, — настройка панели инструментов, по сути — обработка извещений, которые посылает система на каждом шаге настройки. Причем отсутствие обработчика исключает возможность, на первый взгляд, самых простых действий по настройке, например, смены положения кнопок в панели.

Если пользователь начинает перемещать какую-либо кнопку панели инструментов, удерживая нажатой клавишу <Shift>, то элемент управления автоматически обрабатывает операцию перемещения, посылая своему родительскому окну извещение TBN_QUERYDELETE, чтобы определить, можно ли удалить кнопку. Операция перемещения заканчивается, если родительское окно возвращает FALSE, в противном случае панель инструментов захватывает ввод от мыши и ждет, когда пользователь отпустит ее кнопку. Как только кнопка отпущена, панель инструментов определяет местонахождение курсора мыши. Если он находится вне панели, то кнопка удаляется. Если он

находится на другой кнопке, то в родительское окно посылается извещение TBN_QUERYINSERT, чтобы выяснить, можно ли вставить кнопку слева от данной. Кнопка вставляется, если родительское окно возвращает TRUE, и не вставляется в противном случае. При завершении операции перемещения панель инструментов посылает извещение TBN_TOOLBARCHANGE. Если манипуляции с кнопками панели инструментов осуществляются без нажатой клавиши <Shift>, то родительскому окну дополнительно посылаются два извещения — TBN_BEGINDRAG в начале выполнения операции перемещения и TBN_ENDDRAG в конце.

Когда пользователь настраивает панель инструментов, используя блок диалога для настройки (Customize Toolbar), она посылает определенную последовательность извещений. Прежде всего, это извещение TBN_BEGINADJUST, которое посылается после того, как пользователь сделает двойной щелчок левой кнопкой мыши на панели инструментов, но до создания блока диалога. Затем панель инструментов начинает посылать ряд извещений, чтобы определить, можно ли вставлять кнопки. Извещения посылаются до тех пор, пока родительское окно не вернет TRUE (если для всех кнопок возвращается FALSE, то панель инструментов уничтожает блок диалога). И, наконец, она проверяет, можно ли удалять кнопки, посылая для этого по одному сообщению TBN_QUERYDELETE для каждой. Если кнопку можно удалить, то родительское окно возвращает TRUE, в противном случае — FALSE. После этого панель инструментов добавляет все свои кнопки в блок диалога, блокируя те, которые не могут быть удалены.

Всякий раз, когда панель инструментов нуждается в информации о какой-либо кнопке блока диалога, она посылает извещение TBN_GETBUTTONINFO, задавая индекс кнопки и адрес структуры TBNOTIFY, куда родительское окно должно записать соответствующую информацию.

Блок диалога настройки панели инструментов включает также две кнопки: **Help** (справка) и **Reset** (сброс). Когда пользователь нажимает кнопку **Help**, панель инструментов посылает извещение TBN_CUSTHELP, в ответ на которое родительское окно должно отобразить справочную информацию. При нажатии кнопки **Reset** блок диалога посылает извещение TBN_RESET, которое говорит о том, что панель инструментов собирается переинициализировать блок диалога.

Для того чтобы подключить обработчик соответствующего извещения, необходимо в карту сообщений окна, владеющего панелью инструментов, добавить запись следующего вида:

```
ON_NOTIFY(wNotifyCode, idControl, memberFxn)
```

где wNotifyCode — код идентификатора извещения, например, TBN_ENDDRAG; idControl — идентификатор элемента управления, посылающего сообщение; memberFxn — функция-обработчик сообщения, которая должна иметь следующий прототип:

```
afx_msg void memberFxn(
    NMHDR *pNotifyStruct,
    LRESULT *result)
```

Если обработчик должен возвратить какое-либо значение, то его следует поместить в параметр result. Параметр pNotifyStruct указывает на одну из структур NMHDR или TBNOTIFY:

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

В этой структуре поле hwndFrom содержит идентификатор окна, приславшего сообщение, поле idFrom — идентификатор элемента управления, приславшего сообщение, а поле code — код извещения, который может быть специфическим для определенного типа элементов управления или одним из перечисленных ниже, общих для всех элементов управления:

NM_CLICK	Пользователь нажал левую кнопку мыши, когда указатель мыши находился на элементе управления
NM_DBLCLICK	Пользователь сделал двойной щелчок левой кнопкой мыши, когда указатель мыши находился на элементе управления
NM_RCLICK	Пользователь нажал правую кнопку мыши, когда указатель мыши находился на элементе управления
NM_RDBLCLICK	Пользователь сделал двойной щелчок правой кнопкой мыши, когда указатель мыши находился на элементе управления
NM_KILLFOCUS	Элемент управления потерял фокус ввода
NM_SETFOCUS	Элемент управления получил фокус ввода
NM_RETURN	Когда элемент управления имел фокус ввода, пользователь нажал клавишу <Enter>
NM_OUTOFMEMORY	Элемент управления не может завершить операцию из-за нехватки доступной памяти

```
typedef struct tagTBNOTIFY{
  NMHDR hdr;
  int iItem;
  TBBUTTON tbButton;
  int cchText;
```

```
  LPSTR pszText;
}TBNOTIFY, FAR* LPTBNOTIFY;
```

Поле hdr содержит информацию, общую для всех сообщений WM_NOTIFY; iItem — индекс кнопки, ассоциированной с сообщением; tbButton — структура, содержащая информацию о ней; cchText — число символов в тексте кнопки; lpszText — указатель на этот текст.

Помимо перечисленных, общих для всех элементов управления кодов извещений, для панели инструментов определены дополнительные, большинство из которых мы упоминали при обсуждении процесса настройки:

TBN_BEGINADJUST	Посылается, когда пользователь приступил к настройке панели инструментов; указатель pNotifyStruct обработчика указывает на структуру NMHDR; обработчик может ничего не возвращать
TBN_ENDADJUST	Посылается, когда пользователь закончил настройку панели инструментов; указатель pNotifyStruct обработчика указывает на структуру NMHDR; обработчик может ничего не возвращать
TBN_BEGINDRAG	посылается, когда пользователь начал перемещать кнопку по панели инструментов; указатель pNotifyStruct обработчика указывает на структуру TBNOTIFY, поле iItem которой содержит индекс перемещаемой кнопки; обработчик может ничего не возвращать
TBN_ENDDRAG	Посылается, когда пользователь закончил перемещать кнопку по панели инструментов; указатель pNotifyStruct обработчика указывает на структуру TBNOTIFY, поле iItem которой содержит индекс перемещаемой кнопки; обработчик может ничего не возвращать
TBN_CUSTHELP	Посылается, когда пользователь нажимает кнопку **Help** (справка) в блоке диалога настройки панели инструментов; указатель pNotifyStruct обработчика указывает на структуру NMHDR; обработчик должен восстановить исходный вид панели инструментов и может ничего не возвращать
TBN_RESET	Посылается, когда пользователь нажимает кнопку **Reset** (сброс) в блоке диалога настройки панели инструментов; указатель pNotifyStruct обработчика указывает на структуру NMHDR; обработчик может ничего не возвращать
TBN_GETBUTTONINFO	Посылается, когда пользователь настраивает панель инструментов, которая использует это извещение, чтобы получить информацию, необходимую блоку диалога настройки панели инструментов; указатель pNotifyStruct обработчика указывает на структуру TBNOTIFY, поле iItem которой содержит индекс кнопки; обработчик должен заполнить поля структуры и в переменной result возвратить TRUE; если поля структуры по каким-либо причинам не заполнены, то возвращается FALSE

TBN_QUERYDELETE	Посылается, когда пользователь настраивает панель инструментов, чтобы определить, может ли кнопка быть удалена с панели инструментов; указатель pNotifyStruct обработчика указывает на структуру TBNOTIFY, поле iItem которой содержит индекс удаляемой кнопки; обработчик должен возвратить TRUE в переменной result, чтобы удалить кнопку, или FALSE, чтобы предотвратить ее удаление
TBN_QUERYINSERT	Посылается, когда пользователь настраивает панель инструментов, чтобы определить, может ли кнопка быть вставлена в панель инструментов слева от указанной; указатель pNotifyStruct обработчика указывает на структуру TBNOTIFY, поле iItem которой содержит индекс кнопки, которую требуется вставить; обработчик должен возвратить TRUE в переменной result, чтобы вставить кнопку, или FALSE, чтобы предотвратить ее вставку
TBN_TOOLBARCHANGE	Посылается, когда пользователь закончил настройку панели инструментов; указатель pNotifyStruct обработчика указывает на структуру NMHDR; обработчик может ничего не возвращать

В заключение отметим, что настройка панели элементов управления — достаточно трудоемкий процесс — необходимо написать много функций, пусть и не больших по размеру. Кроме того, настраивать объект, созданный на базе класса CToolBar, несколько сложнее, чем объект класса CToolBarCtrl. Связано это с тем, что последний класс сам по себе не поддерживает списков описаний кнопок, и программист вынужден это делать самостоятельно, что существенно упрощает обработку такого уведомляющего сообщения, как TBN_GETBUTTONINFO. С другой стороны, библиотека классов MFC не поддерживает для объектов класса CToolBarCtrl фиксацию и управление расположением, в отличие от объектов класса CToolBar.

Класс CStatusBar

Рассмотрев достаточно подробно все аспекты работы с панелью инструментов, переходим к обсуждению не менее полезного элемента управления — *строки состояния*. Для него библиотека MFC также содержит два класса — CStatusBar и CStatusBarCtrl.

Класс CStatusBar определяет поведение строки состояния — панели элементов управления, отображаемой полосой, которую можно разбить на несколько областей для отдельного вывода в них текста или графической информации. Наиболее часто области используются для вывода сообщений (например, расширенной информации об элементах меню или о кнопках панели инструментов) и для отображения индикаторов состояния (например, клавиш <Caps Lock>, <Num Lock> и <Ins>). В качестве стандартного

шрифта для строки состояния используется шрифт MS Sans Serif, 10 пт. Изменение изображения строки состояния осуществляется автоматически благодаря механизму ON_UPDATE_COMMAND_UI: во время *цикла простоя* (idle time) строка состояния вызывает обработчик команды ON_UPDATE_COMMAND_UI с идентификаторами строк областей индикации. Для строки состояния класс CCmdUI, обеспечивающий правильное функционирование обработчика, поддерживает следующие функции: CCmdUI::Enable и CCmdUI::SetText.

Строка состояния может работать в двух режимах — *упрощенном* и *стандартном*. В первом режиме невозможно разделить это окно на несколько областей и можно выводить в нем только текстовую информацию. В стандартном режиме приложение может разбить окно строки состояния на несколько областей для раздельного вывода в них текстовой или графической информации. По умолчанию принимается, что самая левая область — "эластичная": ее длина определяется длиной области строки состояния, не используемой всеми другими областями, которые, к тому же, выровнены по правому краю. Переход из одного режима в другой осуществляется при помощи сообщения SB_SIMPLE. Все настройки обычно проводятся перед первым отображением строки состояния на экране.

Перейдем теперь к рассмотрению основных членов этого класса.

CStatusBar::**CStatusBar**( ) —

конструктор объекта, который определяет шрифт строки состояния и устанавливает его характеристики в значения, используемые по умолчанию.

BOOL CStatusBar::**Create**(
CWnd *pParentWnd,
DWORD dwStyle = WS_CHILD | WS_VISIBLE | CBRS_BOTTOM,
UINT nID = AFX_IDW_STATUS_BAR) —

создает дочернее окно "строка состояния", ассоциирует его с объектом CStatusBar, инициализирует шрифт и устанавливает его размер в значение, заданное по умолчанию. Для строки состояния помимо стандартных стилей окна доступны еще следующие: CBRS_TOP, CBRS_BOTTOM, CBRS_NOALIGN и CBRS_SIZEGRIP. Обратите внимание на последний стиль, при котором правый нижний угол строки состояния принимает вид, показанный на рис. 11.1.

Рис. 11.1. Строка состояния, созданная со стилем CBRS_SIZEGRIP

По умолчанию строка состояния располагается в нижней части родительского окна. Однако при желании вы можете расположить ее и в верхней части, использовав стиль CBRS_TOP. Очевидно, что этот стиль нельзя комбинировать со стилем CBRS_SIZEGRIP, т. к., во-первых, такой элемент управления будет выглядеть очень странно, а во-вторых, он не будет работать так, как ожидает пользователь.

Аналогично классу CToolBar, в рассматриваемом классе также реализована новая функция

BOOL CStatusBar::**CreateEx**(
CWnd* pParentWnd,
DWORD dwCtrlStyle = 0 ,
DWORD dwStyle = WS_CHILD | WS_VISIBLE | CBRS_BOTTOM,
UINT nID = AFX_IDW_STATUS_BAR) —

dwCtrlStyle — дополнительные стили для создания встроенного объекта класса CStatusBarCtrl. Остальные параметры остались без изменения.

Примечание

Подробно стили объекта класса CStatusBarCtrl будут описаны при рассмотрении соответствующего класса

Эту функцию следует использовать вместо рассмотренной ранее, поскольку она позволят одновременно создать и объект класса CStatusBar библиотеки MFC, и встроенный в него соответствующий элемент управления STATUSBAR Windows.

BOOL CStatusBar::**SetIndicators**(
const UINT *lpIDArray,
int nIDCount) —

устанавливает идентификатор каждой области в значение, заданное соответствующим элементом массива lpIDArray, загружает для них строки ресурса и записывает соответствующие текстовые строки в эти области. Параметр nIDCount определяет число элементов в массиве идентификаторов.

void CStatusBar::**SetPaneInfo**(
int nIndex,
UINT nID,
UINT nStyle,
int cxWidth) —

устанавливает для области, заданной номером nIndex, новые значения идентификатора (nID), стиля (nStyle) и ширины (cxWidth). Поддерживаются следующие дополнительные стили:

SBPS_NOBORDERS	Вокруг области нет трехмерной рамки
SBPS_POPOUT	Текст рисуется с рамкой, которая выглядит приподнятой над поверхностью строки состояния
SBPS_DISABLED	Запрещен вывод текста в область
SBPS_STRETCH	"Эластичная" (stretch) область, занимающая ту часть строки состояния, которая не используется под другие области; этот стиль может иметь только одна область строки состояния
SBPS_NORMAL	Обычная область — без рамок и без возможности растяжения

void CStatusBar::**GetPaneInfo**(
int nIndex,
UINT &nID,
UINT &nStyle,
int &cxWidth) —

позволяет получить информацию об области строки состояния, заданной номером nIndex, включающую идентификатор (nID), стиль (nStyle) и ширину (cxWidth).

UINT CStatusBar::**SetPaneStyle**(
int nIndex,
UINT nStyle) —

устанавливает стиль (nStyle) области (nIndex) строки состояния.

UINT CStatusBar::**GetPaneStyle**(int nIndex) —

возвращает стиль области строки состояния, который определяет ее внешний вид. Номер области задается параметром nIndex.

BOOL CStatusBar::**SetPaneText**(
int nIndex,
LPCTSTR lpszNewText,
BOOL bUpdate = TRUE) —

записывает текст, на который указывает lpszNewText, в область строки состояния, заданную номером nIndex. Если значение параметра bUpdate равно TRUE, то после записи текста область помечается как поврежденная для ее немедленного обновления.

void CStatusBar::**GetPaneText**(
int nIndex,
CString &rString)

и

CString CStatusBar::**GetPaneText**(int nIndex) —

позволяют получить текст, который находится в области с номером nIndex.

Теперь перерисовку областей строки состояния можно организовать непосредственно с помощью объекта рассматриваемого класса (напомним, что раньше эта задача возлагалась на элемент управления STATUSBAR Windows — там это сделать тоже можно).

virtual void CStatusBar::**DrawItem**( ) —

вызывается родительским окном, когда область элемента управления должна быть перерисована. Структура DRAWITEMSTRUCT, указатель на которую передается в качестве параметра функции, содержит всю информацию, необходимую для рисования.

Последняя функция класса CStatusBar позволяет получить ссылку на объект CStatusBarCtrl:

CStatusBarCtrl& CStatusBar::**GetStatusBarCtrl**( ) —

возвращает ссылку на объект общего элемента управления STATUSBAR для прямого доступа к его функциональным возможностям, включающим, в частности, настройку строки состояния.

Класс CStatusBarCtrl

Для полного использования функциональных возможностей общего элемента управления STATUSBAR можно пойти двумя путями: либо непосредственно создать окно CStatusBarCtrl, либо создать строку состояния на основе класса CStatusBar, а затем просто получить доступ к элементам объекта CStatusBarCtrl, вызвав функцию этого класса CStatusBar::CStatusBarCtrl. Остановимся на втором способе и не будем рассматривать непосредственное создание общего элемента управления STATUSBAR.

Для того чтобы прочитать текст из какой-либо области элемента управления, в классе реализована функция GetText :

int CStatusBarCtrl::**GetText**(
LPCTSTR lpszText,
int nPane,
int *pType) —

nPane — индекс области, из которой читается текст; lpszText — указатель на буфер, в который этот текст будет записан; pType — указатель на целое значение, в котором будет возвращена информация о типе отображения текста.

Сама функция возвращает длину текста, которую можно также получить при помощи функции

int CStatusBarCtrl::**GetTextLength**(
int nPane,
int *pType)

В любую область строки состояния можно записать произвольную текстовую или, как ни странно, графическую информацию, воспользовавшись для этого функцией

BOOL CStatusBarCtrl::**SetText**(
LPCTSTR lpszText,
int nPane,
int nType) —

nPane — индекс области, в которую записывается текст из lpszText. Если nPane = 255, то строка состояния содержит единственную область, в которую можно выводить только текст. Параметр nType задает тип отображения текста и может принимать одно из значений:

0	Текст выводится с рамкой, которая выглядит утопленной
SBT_NOBORDERS	Текст выводится без рамки

SBT_OWNERDRAW	Текст (или графическое изображение) выводится родительским окном
SBT_POPOUT	Текст выводится с рамкой, которая выглядит приподнятой

Если nType принимает значение SBT_OWNERDRAW, то в параметре lpszText передаются 32 бита данных. При выполнении этой функции область, в которую выводится текст, становится областью модификации, чтобы отобразить новый текст при получении сообщения WM_PAINT.

Для того чтобы разбить элемент управления на области определенных размеров, можно воспользоваться функцией

BOOL CStatusBarCtrl::**SetParts**(
int nParts,
int *pWidths) —

устанавливает число областей и координату правого края каждой из них. Число областей (nParts) не может быть больше 255. Параметр pWidths содержит адрес целочисленного массива, каждый элемент которого определяет позицию правого края соответствующей области. Если элемент равен –1, то правый край этой области совпадает с правым краем элемента управления.

Обратную задачу позволяет решить функция

int CStatusBarCtrl::**GetParts**(
int nParts,
int *pParts) —

возвращает число областей, на которое разбит элемент управления, а также координаты правых краев областей, число которых запрашивается в параметре nParts. Если это число больше, чем имеется у элемента управления, то информация возвращается только для существующих областей. Координаты записываются в целочисленный массив pParts, число элементов которого равно nParts. Если для какой-либо области в качестве координаты правого края области в массив записывается –1, то это означает, что он совпадает с правым краем элемента управления.

Если требуется информация о размерах рамок элемента управления и его областей, то ее можно получить при помощи функции GetBorders, которая реализована в двух вариантах:

BOOL CStatusBarCtrl::**GetBorders**(
int &nHorz,
int &nVert,
int &nSpacing)

и

BOOL CStatusBarCtrl::**GetBorders**(int *pBorders) —

первая возвращает ширину горизонтальной (nHorz) и вертикальной (nVert) рамки, а также рамки между различными областями (nSpacing). Вторая записывает всю эту информацию в целочисленный трехэлементный массив. Эти расстояния определяют интервал между внешним краем элемента управления и прямоугольником, внутри которого находится текстовая информация.

Помимо изменения горизонтальных размеров различных областей строки состояния можно установить ее минимальную высоту, необходимую для отображения текстовой или графической информации:

void CStatusBarCtrl::**SetMinHeight**(int nMin) —

минимальная высота определяется суммой значения параметра nMin и удвоенного значения ширины рамки.

Для переключения между упрощенным и стандартным режимами работы строки состояния используется функция

BOOL CStatusBarCtrl::**SetSimple**(BOOL bSimple = TRUE) —

если значение bSimple равно TRUE, то элемент управления переводится в упрощенный режим, в противном случае — в стандартный. При переходе из одного режима в другой происходит немедленная перерисовка строки состояния. Содержимое областей при переключении не теряется, т. к. данные соответствующих режимов хранятся в различных местах.

Узнать текущий режим строки состояния можно с помощью функции

BOOL CStatusBarCtrl::**IsSimple**( ) —

возвращает ненулевое значение, если строка состояния находится в упрощенном режиме, и нулевое — если в стандартном.

BOOL CStatusBarCtrl::**GetRect**(
int nPane,
LPRECT lpRect) —

служит для получения размеров и координат прямоугольника, ограничивающего область nPane элемента управления. Информация о прямоугольнике записывается в структуру RECT, на которую указывает lpRect.

Как уже упоминалось, в строке состояния может находиться не только текстовая, но и графическая информация. Для реализации этой возможности как и раньше можно при вызове функции CStatusBarCtrl::SetText указать константу SBT_OWNERDRAW и переопределить следующую функцию:

virtual void CStatusBarCtrl::**DrawItem**(
LPDRAWITEMSTRUCT lpDrawItemStruct) —

вызывается родительским окном, когда область элемента управления должна быть перерисована. Структура DRAWITEMSTRUCT, указатель на которую передается функции в качестве параметра, содержит всю информацию необходимую для рисования. Реализация этой функции по умолчанию не делает ничего.

А можно воспользоваться новой функцией CStatusBarCtrl::SetIcon, о которой речь пойдет ниже.

Рассмотрим поля структуры DRAWITEMSTRUCT применительно к элементу управления "строка состояния":

```
typedef struct tagDRAWITEMSTRUCT{
  UINT CtlType;      // для строки состояния не используется
  UINT CtlID;
```

```
  UINT itemID;
  UINT itemAction;
  UINT itemState;
  HWND hwndItem;
  HDC hDC;
  RECT rcItem;
  DWORD itemData;
} DRAWITEMSTRUCT;
```

Поле hwndItem содержит дескриптор окна строки состояния, поле CtlId — ее идентификатор, а поле itemID — номер области, которая будет перерисована. Контекст устройства, необходимый для выполнения операций рисования, находится в поле hDC, а координаты области, в которой будет осуществляться рисование, передаются через поле rcItem.

Кроме того, можно воспользоваться содержимым поля itemData, которое равно значению параметра lpszText функции CStatusBarCtrl::SetText. Например, через этот параметр можно передать идентификатор изображения, которое должно быть нарисовано в данной области строки состояния, или идентификатор шрифта, которым будет написан текст, или какую-либо другую информацию.

Теперь в библиотеке классов MFC добавлена поддержка работы со всплывающими подсказками и значками строки состояния, а также изменения ее цвета. Рассмотрим новые функции, реализующие эти возможности.

CString CStatusBarCtrl::**GetTipText**(int nPane)

и

void CStatusBarCtrl::**SetTipText**(
int nPane,
LPCTSTR CpszTipText) —

позволяют получить и, соответственно, определить текстовую строку всплывающей подсказки, которая появляется над заданной параметром nPane — индексом (начиная с нуля) области строки состояния. При получении текста он записывается в объект класса CString, а при определении на него указывает CpszTipText.

BOOL CStatusBarCtrl::**SetIcon**(
int nPane,
HICON hIcon) —

значительно упрощает вывод графической информации в произвольную область строки состояния. Номер области задается параметром nPane, начиная с нуля, а значок — дескриптором hIcon. Для того чтобы удалить значок из строки состояния, просто передайте в эту функцию значение NULL в качестве значения hIcon.

Последняя из рассматриваемых функций:

COLORREF CStatusBarCtrl::**SetBkColor**(COLORREF cr) —

позволяет изменить цвет строки состояния на заданный параметром cr. При этом функция возвращает предыдущее значение цвета фона. В качестве значения параметра cr можно использовать предопределенную константу CLR_DEFAULT, которая определяет цвет, установленный по умолчанию.

В заключение осталось описать, как строка состояния обрабатывает команду ON_UPDATE_COMMAND_UI. Как обычно, обработчик этой команды вызывается во время цикла простоя последовательно для всех областей строки состояния. Доступными функциями класса CCmdUI, отвечающего за обработку, являются Enable и SetText. Первая из них блокирует или разблокирует соответствующую область. При этом надо иметь в виду, что у заблокированных областей выключается отображение текста. Вторая функция позволяет изменить текст, который выводится в текущей области. Следует помнить, что при выводе текста не происходит автоматического изменения размеров области.

На этом мы заканчиваем рассмотрение наиболее часто используемых панелей элементов управления — панели инструментов и строки состояния — и переходим к описанию не менее интересного класса CDialogBar.

Класс CDialogBar

Панель диалога — панель элементов управления с функциональными возможностями немодального блока диалога, которая содержит стандартные элементы управления Windows. Она действует подобно блоку диалога, поддерживая, в частности, перемещение между элементами при помощи клавиши <Tab>. Кроме того, для нее, как и для блока диалога, используется шаблон диалога. Это с одной стороны. А с другой, ее можно рассматривать как более мощную панель инструментов. Извещения от элементов управления панели диалога посылаются ее родительскому окну точно так же, как и от кнопок панели инструментов. Основное их различие в том, что панель диалога может содержать произвольные элементы управления, в то время как панель инструментов, стандартно, — только кнопки. Панель диалога наиболее часто используется в качестве дополнения к основному окну, и любые извещения, такие как BN_CLICKED или EN_CHANGE будут посланы ее родителю — главному окну приложения.

Примечание

Однако, к сожалению, ClassWizard ничем помочь не может — он ничего "не знает" об элементах управления, указанных в шаблоне.

Класс CDialogBar имеет всего две основные функции — конструктор, создающий объект этого класса:

CDialogBar::**CDialogBar**( )

и функцию Create, как обычно, отвечающую за создание окна Windows и присоединение его к текущему объекту:

```
BOOL CDialogBar::Create(
    CWnd *pParentWnd,
    LPCTSTR lpszTemplateName,
    UINT nStyle,
    UINT nID)
```

и

```
BOOL CDialogBar::Create(
    CWnd *pParentWnd,
    UINT nIDTemplate,
    UINT nStyle,
    UINT nID) —
```

pParentWnd — указатель на родительское окно (не может быть равен NULL); lpszTemplateName — указатель на строку, содержащую имя ресурса шаблона панели диалога (второй вариант функции вместо имени использует идентификатор ресурса nIDTemplate); nID — идентификатор панели диалога; nStyle — стиль выравнивания панели диалога:

CBRS_TOP	Панель диалога располагается у верхней стороны фрейма
CBRS_BOTTOM	Панель диалога располагается у нижней стороны фрейма
CBRS_LEFT	Панель диалога располагается у левой стороны фрейма
CBRS_RIGHT	Панель диалога располагается у правой стороны фрейма
CBRS_NOALIGN	Панель диалога не меняет свое расположение, когда у родительского окна изменяется размер

Эта функция (в любом варианте) загружает шаблон диалога из ресурса, заданного параметром lpszTemplateName или nIDTemplate, создает окно панели диалога, устанавливает его стиль и ассоциирует с объектом CDialogBar. Если определен стиль CBRS_TOP или CBRS_BOTTOM, то ширина панели определяется шириной фрейма, а высота берется из шаблона nIDTemplate. Если же определен стиль CBRS_LEFT или CBRS_RIGHT, то высота панели определяется высотой фрейма, а ширина берется из шаблона nIDTemplate.

Все остальные функциональные возможности этот класс получает от своего базового CControlBar.

Класс CReBar

Этот класс определяет поведение *расширенной панели инструментов* — панели, отображаемой полосой элементов управления. Основным назначением этого класса является предоставление более простого доступа к одноименному дополнительному элементу управления Windows. Объект этого класса может содержать различные дочерние окна, чаще всего элементы управления. Более того, он может содержать и "обычные" панели управления.

В этом классе реализовано всего несколько функций (примечательно, что в справочной системе даже не упоминается конструктор). Рассмотрим их.

BOOL CReBar::**Create**(
CWnd* pParentWnd,
DWORD dwCtrlStyle = RBS_BANDBORDERS,
DWORD dwStyle = WS_CHILD | WS_VISIBLE | WS_CLIPSIBLINGS |
WS_CLIPCHILDREN | CBRS_TOP,
UINT nID = AFX_IDW_REBAR) —

pParentWnd — указатель на родительское окно дополнительной панели инструментов, dwCtrlStyle — стиль дополнительной панели инструментов (по умолчанию RBS_BANDBORDERS), nID — идентификатор этого дочернего окна; dwStyle — стиль окна дополнительной панели инструментов. Помимо общих стилей окон Windows и панелей элементов управления, она поддерживает дополнительные:

RBS_AUTOSIZE	Элемент управления будет автоматически изменять расположение полос при изменении размера и положения элемента управления.
RBS_BANDBORDERS	Для разделения полос элемент управления отображает узкую линию.
RBS_DBLCLKTOGGLE	Полосы элемента управления переключаются от минимизированного до максимизированного состояния, когда пользователь делает двойной щелчок на полосе. При отсутствии этого флага переключение осуществляется при одиночном щелчке на полосе
RBS_FIXEDORDER	Элемент управления всегда отображает полосы в одном и том же порядке
RBS_REGISTERDROP	Чтобы при перемещении элемента управления он посылал уведомление, необходимо установить этот флаг
RBS_TOOLTIPS	Название говорит само за себя, но, к сожалению, этот стиль пока не поддерживается
RBS_VARHEIGHT	Полосы элемента управления отображаются с минимальным размером по высоте
RBS_VERTICALGRIPPER	В вертикальном элементе управления размер участка захвата отображается также вертикально, вместо используемого по умолчанию горизонтального расположения. Однако, если для элемента управления не установлен стиль CCS_VERT, данный стиль игнорируется

Следующая функция позволяет вставить в расширенную панель управления новую полосу:

BOOL CReBar::**AddBar**(
CWnd* pBar,
LPCTSTR lpszText = NULL,
CBitmap* pbmp = NULL,
DWORD dwStyle = RBBS_GRIPPERALWAYS | RBBS_FIXEDBMP),

и

BOOL CReBar::**AddBar**(
CWnd* pBar,
COLORREF clrFore,
COLORREF clrBack,
LPCTSTR pszText = NULL,
DWORD dwStyle = RBBS_GRIPPERALWAYS) —

pBar — указатель на дочернее окно, которое вставляется в расширенную панель управления (оно всегда должно иметь установленный стиль WS_CHILD), pszText — указатель на текстовую строку, которая отображается в полосе, pbmp — указатель на битовый массив (объект класса CBitmap), который будет отображаться в качестве фона (по умолчанию NULL, что соответствует предопределенному цвету фона), ctrFore и ctrBack — значения, определяющие цвета фона и основной, dwCtrlStyle — стиль дополнительной панели инструментов, который может принимать комбинацию следующих значений:

RBBS_BREAK	Полоса отображается на новой линии
RBBS_CHILDEDGE	Границы полосы расположены сверху и снизу дочернего окна
RBBS_FIXEDBMP	Битовый массив, определяющий фон полосы, не перемещается при изменении ее размеров
RBBS_FIXEDSIZE	Полоса не может изменять размеры. В этом случае участок захвата у полосы не отображается
RBBS_GRIPPERALWAYS	Участок захвата отображается всегда, даже если полоса единственная
RBBS_HIDDEN	Полоса невидима
RBBS_NOGRIPPER	Участок захвата не отображается никогда, даже если полоса не единственная
RBBS_NOVERT	Полоса не отображается, когда у элемента управления установлен стиль CCS_VERT
RBBS_VARIABLEHEIGHT	Полоса может изменить вертикальные размеры элемента управления, когда он изменяет свои размеры

Последняя функция класса (едва ли не самая важная):

CReBarCtrl& CReBar::**GetReBarCtrl**( ) —

позволяет получить доступ к одноименному элементу управления.

Именно в элементе управления REBAR и реализованы все основные достоинства расширенной панели управления.

Класс CReBarCtrl

Объект этого класса чрезвычайно похож по поведению на объект панели инструментов. Однако в отличие от своего аналога, он содержит полосы (одну или несколько), каждая из которых может включать участок захвата (grid), текст, битовый массив и *единственное* дочернее окно. Этот элемент управления поддерживает:

- Список образов
- Обработчики сообщений
- Функциональные возможности по настройке перерисовки
- Разнообразные дополнительные стили (см. выше)

Рассмотрим компоненты класса.

CReBarCtrl::**CReBarCtrl**( ) —

конструктор класса, отвечающий за создание объекта.

BOOL CReBarCtrl::**Create**(
DWORD dwStyle,
const RECT& rect,
CWnd* pParentWnd,
UINT nID) —

создает соответствующий объект Windows и присоединяет его к объекту класса. Параметры: dwCtrlStyle — стиль дополнительной панели инструментов, который может принимать комбинацию значений, приведенных выше; rect — ссылка на объект класса CRect или структуру RECT, которая содержит положение и размер элемента управления; pParentWnd — указатель на родительское окно элемента управления (это значение не должно быть NULL); nID — идентификатор элемента управления.

После того как элемент управления создан, можно получить или установить информацию о нем. Для этого в классе реализованы две функции:

BOOL CReBarCtrl::**GetBarInfo**(REBARINFO* prbi)

и

BOOL CReBarCtrl::**SetBarInfo**(REBARINFO* prbi) —

в качестве параметра принимают указатель на структуру, содержащую информацию о характеристиках элемента управления REBAR:

```
typedef struct tagREBARINFO{
    UINT        cbSize;
    UINT        fMask;
    HIMAGELIST  himl;
} REBARINFO, FAR *LPREBARINFO;
```

cbSize — размер структуры в байтах; fMask — флаг, описывающий характеристики элемента управления REBAR, в настоящий момент определено только одно значение — RBIM_IMAGELIST, открывающее доступ к полю himl — дескриптору списка доступных образов.

Управлять списком образов, ассоциированных с элементом управления, можно с помощью функций:

CImageList* CReBarCtrl::**GetImageList**( )

и

BOOL CReBarCtrl::**SetImageList**(CImageList* pImageList) —

первая возвращает, а вторая принимает указатель pImageList на объект класса CImageList, где собственно, и хранятся образы, ассоциированные с элементом управления.

Как вы помните, расширенная панель инструментов может состоять из нескольких полос. Их число, а также подробную информацию о них можно получить (и установить) с помощью следующих функций класса:

UINT CReBarCtrl::**GetBandCount**( ) —

позволяет узнать число полос элемента управления.

BOOL CReBarCtrl::**SetBandInfo**(
UINT uBand,
REBARBANDINFO* prbbi)

и

BOOL CReBarCtrl::**GetBandInfo**(
UINT uBand,
REBARBANDINFO* prbbi) —

записывает информацию о полосе элемента управления, задаваемую индексом (начиная с нуля) uBand, в структуру REBARBANDINFO, указатель на которую передается в качестве второго параметра.

Перед тем как использовать две последние функции, необходимо установить параметр cbSize структуры REBARBANDINFO в значение sizeof (REBARBANDINFO) и определить маску fMask. Но чтобы было более понятно, о чем идет речь, рассмотрим саму структуру, которая содержит информацию о полосе расширенной панели управления:

```
typedef struct tagREBARBANDINFO{
    UINT        cbSize;
    UINT        fMask;
    UINT        fStyle;
    COLORREF    clrFore;
    COLORREF    clrBack;
    LPTSTR      lpText;
    UINT        cch;
    int         iImage;
    HWND        hwndChild;
    UINT        cxMinChild;
    UINT        cyMinChild;
    UINT        cx;
    HBITMAP     hbmBack;
    UINT        wID;
```

```
#if (_WIN32_IE >= 0x0400)
    UINT        cyChild;
    UINT        cyMaxChild;
    UINT        cyIntegral;
    UINT        cxIdeal;
    LPARAM      lParam;
    UINT        cxHeader;
#endif
 } REBARBANDINFO, FAR *LPREBARBANDINFO —
```

cbSize — размер структуры в байтах. Запомните — перед тем как использовать эту структуру, поле должно быть *обязательно* заполнено.

fMask — определяет доступность полей структуры и может быть комбинацией следующих значений:

RBBIM_BACKGROUND	Установка этого флага открывает доступ к полю hbmMask.
RBBIM_CHILD	Установка этого флага открывает доступ к полю hwndChild
RBBIM_CHILDSIZE	Установка этого флага открывает доступ к полям cxMinChild, cyMinChild, cyChild, cyMaxChild и cyIntegral
RBBIM_COLORS	Установка этого флага открывает доступ к полям crlFore и crlBack
RBBIM_HEADERSIZE	Установка этого флага открывает доступ к полю cxHeader
RBBIM_IDEALSIZE	Установка этого флага открывает доступ к полю cxIdeal
RBBIM_ID	Установка этого флага открывает доступ к полю wID
RBBIM_IMAGE	Установка этого флага открывает доступ к полю iImage
RBBIM_LPARAM	Установка этого флага открывает доступ к полю lParam
RBBIM_SIZE	Установка этого флага открывает доступ к полю cx
RBBIM_STYLE	Установка этого флага открывает доступ к полю fStyla
RBBIM_TEXT	Установка этого флага открывает доступ к полю lpText

fStyle — определяет стиль полосы расширенной панели управления. Возможные значения были приведены при описании функции CReBar::AddBar;

clrFore и clrBack — основной и фоновый цвета полосы (эти значения игнорируются, если в качестве фона используется битовый массив hbmBack);

lpText — указатель на строку текста, которая будет отображаться на полосе. Имейте в виду, что перед тем, как запрашивать у полосы информацию, предварительно необходимо подготовить буфер;

cch — размер буфера в байтах для размещения текста;

iImage — индекс (начиная с нуля) образа в списке, который должен быть изображен;

hwndChild — дескриптор дочернего окна полосы (напомним, что такое окно может быть только одно);

cxMinChild и cyMinChild — минимальные значения ширины и высоты (в пикселях) дочернего окна. Полосу нельзя сделать меньше этих размеров;

cx — длина полосы в пикселях;

hbmBack — дескриптор битового массива, который используется в качестве фона полосы;

wID — идентификатор элемента управления;

cyChild и cyMaxChild — начальное и максимальное значения высоты (в пикселях) полосы (эти значения игнорируются если определен стиль RBBS_VARIABLEHEIGHT);

cyIntegral — значение шага (в пикселях), на которое можно изменять ширину полосы (это значение игнорируется, если определен стиль RBBS_VARIABLEHEIGHT)

cxIdeal — "идеальное" значение ширины полосы (в пикселях);

lParam — 32-разрядное значение, определяемое приложением;

cxHeader — размер (в пикселях) заголовка полосы — области между границами полосы и дочернего окна (в этой области отображаются текст и/или образ, если определены);

Приведем также краткое описание еще нескольких функций: GetRect — позволяет получить прямоугольник, ограничивающий полосу; GetRowCount — возвращает число строк полосы; GetBandBorders —дает информацию о бордюрах полосы; IDToIndex — конвертирует идентификатор полосы в индекс; GetBarHeight и GetRowHeight — возвращают высоту (в пикселях) элемента управления и полосы; GetBkColor, SetBkColor, GetTextColor, SetTextColor, GetPalette и SetPalette — позволяют узнать или установить цвета фона и текста элемента управления, а также используемую палитру. Функция SetOwner позволяет изменить текущего владельца расширенной панели управления.

Примечание

Здесь следует иметь в виду, что функция SetOwner позволяет переопределить окно, которое будет получать уведомления от расширенной панели. При этом родительское окно *не меняется*.

CToolTipCtrl* CReBarCtrl::**GetToolTips**( )

и

void CReBarCtrl::**SetToolTips**(CToolTipCtrl* pToolTip) —

позволяют получить (соответственно, установить) указатель pToolTip на объект класса CToolTipCtrl для работы со всплывающими подсказками.

Группа функций предназначена для изменения размеров полос расширенной панели инструментов: MaximizeBand — приводит размеры полосы в соответствие с "идеальными", или наибольшими размерами, MinimazeBand — приводит размеры полосы в соответствие с наименьшим размером, SizeToRect — "подгоняет" размеры полосы к размерам прямоугольника, задаваемого объектом CRect.

BOOL CReBarCtrl::**ShowBand**(
UINT uBand,
BOOL fShow = TRUE) —

скрывает (fShow = FALSE) или показывает (fShow = TRUE) полосу, задаваемую индексом (с нуля) uBand.

Полосы в расширенной панели инструментов располагаются последовательно, их положение определяется индексом. Для изменения расположения полос в классе реализована функция:

BOOL CReBarCtrl::**MoveBand**(
UINT uFrom,
UINT uTo) —

uFrom — индекс перемещаемой полосы, uTo — новое значение индекса, которое не должно быть больше общего числа полос без единицы.

Естественно, что для добавления и удаления полос в классе реализованы соответствующие функции:

BOOL CReBarCtrl::**InsertBand**(
UINT uIndex,
REBARBANDINFO* prbbi) —

добавляет новую полосу в расширенную панель инструментов. Ее положение определяется индексом uIndex, а свойства задаются в prbbi, указателе на предварительно заполненную структуру REBARBANDINFO, описанную выше.

BOOL CReBarCtrl::**DeleteBand**(UINT uBand) —

удаляет полосу, заданную индексом uBand, из расширенной панели инструментов.

Правда, напрямую эти функции используются не так часто. Гораздо нагляднее подход, при котором сначала добавляется полоса, с помощью функции CReBar::AddBar, а затем для каждой добавленной полосы устанавливаются ее свойства. Как это сделано, например, в следующем фрагменте:

```
...
// Добавляем предварительно созданную панель инструментов,
// анимационную картинку и адресную полосу
// в расширенную панель инструментов
m_wndReBar.AddBar(&m_wndToolBar);
m_wndReBar.AddBar(&m_wndAnimate, NULL, NULL,
                  RBBS_FIXEDSIZE | RBBS_FIXEDBMP);
str.LoadString(IDS_ADDRESS);
m_wndReBar.AddBar(&m_wndAddress, str, NULL,
                  RBBS_FIXEDBMP | RBBS_BREAK);

// Устанавливаем минимальный, максимальный и "идеальный"
// размеры каждой из полос расширенной панели инструментов
REBARBANDINFO rbbi;

rbbi.cbSize = sizeof(rbbi);
rbbi.fMask = RBBIM_CHILDSIZE | RBBIM_IDEALSIZE | RBBIM_SIZE;
```

```
rbbi.cxMinChild = rectToolBar.Width();
rbbi.cyMinChild = rectToolBar.Height();
rbbi.cx = rbbi.cxIdeal = rectToolBar.Width() * 9;
m_wndReBar.GetReBarCtrl().SetBandInfo(0, &rbbi);
rbbi.cxMinChild = 0;

CRect rectAddress;

rbbi.fMask = RBBIM_CHILDSIZE | RBBIM_IDEALSIZE;
m_wndAddress.GetEditCtrl()->GetWindowRect(&rectAddress);
rbbi.cyMinChild = rectAddress.Height() + 10;
rbbi.cxIdeal = 200;
m_wndReBar.GetReBarCtrl().SetBandInfo(2, &rbbi);
...
```

То, какой из подходов предпочтете вы, — зависит от вашего вкуса и пристрастий. А мы пойдем дальше. Как отмечалось выше, компоненты расширенной панели инструментов можно перемещать, используя механизм drag-and-drop. Рассмотрим функции, которые реализованы в классе для поддержки этого механизма.

void CReBarCtrl::**BeginDrag**(
UINT uBand,
DWORD dwPos = (DWORD) −1) —

переводит расширенную панель инструментов в режим drag-and-drop. Параметры: uBand — индекс полосы, для которой осуществляется переход; dwPos — начальные координаты мыши (горизонтальные — младшее слово, вертикальные — старшее); если dwPos = –1, то в качестве значений координат используются последние координаты, которые имел указатель мыши при вызове ::GetMessage или ::PutMessage.

void CReBarCtrl::**DragMove**(DWORD dwPos = (DWORD) −1) —

устанавливает новое положение полосы расширенной панели инструментов, которое задается координатами мыши, передаваемыми в параметре dwPos.

void CReBarCtrl::**EndDrag**( ) —

выводит расширенную панель инструментов из режима drag-and-drop.

Последняя из рассматриваемых функций:

int **HitTest**(RBHITTESTINFO* prbht) —

позволяет определить, принадлежит ли некоторая точка полосе расширенной панели инструментов. Она возвращает номер полосы, если таковая есть, и –1 в противном случае. Предварительно необходимо заполнить специальную структуру:

```
typedef struct _RB_HITTESTINFO {
    POINT pt;
    UINT  flags;
```

```
    int   iBand;
    } RBHITTESTINFO;
```

pt — структура POINT, содержащая клиентские координаты тестируемой точки; flag — значение флага, показывающего, какой компонент полосы расположен в рассматриваемой точке:

RBHT_CAPTION	Точка принадлежит заголовку полосы
RBHT_CLIENT	Точка принадлежит клиентской области полосы
RBHT_GRABBER	Точка принадлежит "участку захвата" (grid) полосы
RBHT_NOWHERE	Точка не принадлежит полосе

iBand — поле, в которое записывается индекс соответствующей полосы, или –1, если таковой нет.

Работа с расширенной панелью инструментов иллюстрируется приведенным ниже примером.

ControlBars — пример приложения

Как известно, лучше один раз увидеть, чем сто раз услышать. Перейдем к примеру использования, конечно не всех, но хотя бы части заявленных возможностей. Сначала несколько слов о составе примера. Весь код приложения, окно которого представлено на рис. 11.2, занимает более 800 строк, не считая добавленных нами комментариев, плюс еще три файла для хранения битовых массивов панелей инструментов и файл значка приложения. Мы ограничились лишь двумя файлами исходного текста на C++ — этого больше чем достаточно.

Все приложение состоит из описания двух классов: неизменного класса приложения (CControlBarsApp) и его главного окна (CMainFrame).

```
/////////////////////////////////////////////////////////////
// Файл ControlBars.cpp
// Copyright (c) 1999 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////////
...
/////////////////////////////////////////////////////////////
// Инициализация CControlBarsApp
BOOL CControlBarsApp::InitInstance()
{
  ...
// Создание нового SDI-окна
  CFrameWnd* pMainFrame = new CMainFrame;
```

```
  if (!pMainFrame->LoadFrame(IDR_MAINFRAME))
    return FALSE;

  pMainFrame->ShowWindow(m_nCmdShow);
  pMainFrame->UpdateWindow();

  m_pMainWnd = pMainFrame;
  ...
}
...
/////////////////////////////////////////////////////////////
// Файл MainFrm.h
// Copyright (c) 1999 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////////
#ifndef __AFXEXT_H__
#include <afxext.h>         // Для включения объявлений классов
#endif                      // CToolBar и CStatusBar
class CMainFrame : public CFrameWnd
{
// Конструктор
public:
  CMainFrame();

// Операции
public:
  void SetColumns(UINT nColumns); // Изменение числа столбцов панели
                                  // инструментов рисования

public:
  virtual ~CMainFrame();
  virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
#ifdef _DEBUG
  virtual void AssertValid() const;
  virtual void Dump(CDumpContext& dc) const;
#endif

  // Названия функций говорят сами за себя
  BOOL CreateToolBar();
```

```
  BOOL CreatePaneBar();
  BOOL CreateStatusBar();

protected:
  // Встроенные объекты панелей управления
  CStatusBar m_wndStatusBar;  // Строка состояния
  CToolBar   m_wndToolBar;    // Панель инструментов (команд)
  CToolBar   m_wndPaneBar;    // Панель инструментов рисования
  CDialogBar m_wndDlgBar;     // Панель диалога

  // Дополнительные члены класса
  UINT       m_nPaneCol;      // Число столбцов панели инструментов
                              // рисования
  CComboBox  m_comboBox;      // Комбинированный список для панели
                              // инструментов (команд)
  CFont      m_font;          // Шрифт для комбинированного списка

// Обработчики сообщений
protected:
  //{{AFX_MSG(CMainFrame)
  afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct); // Создание
                                                       // окна
  afx_msg void OnViewPane(); // Команда отображения панели
                             // инструментов рисования
  afx_msg void OnUpdateViewPane(CCmdUI* pCmdUI);
  afx_msg void OnSelChangePane(); // Обработчик изменения выбора
                                  // инструмента рисования

  // Обработчики команд смены количества
  // столбцов панели инструментов рисования
  afx_msg void OnPane2column();
  afx_msg void OnPane3column();

  afx_msg void OnUpdatePane2column(CCmdUI* pCmdUI);
  afx_msg void OnUpdatePane3column(CCmdUI* pCmdUI);

  // Для обновления области строки состояния
  afx_msg void OnUpdateIndicatorIdle(CCmdUI* pCmdUI);
```

```
  // Обработчик таймера
  afx_msg void OnTimer(UINT nIDEvent);
  //}}AFX_MSG

  // Обработчики смены выбора для комбинированных списков
  afx_msg void OnStyle(UINT nID); // Список стилей выравнивания
  afx_msg void OnUpdateStyle(CCmdUI* pCmdUI);
  afx_msg void OnPane(UINT nID); // Список инструментов рисования
  afx_msg void OnUpdatePane(CCmdUI* pCmdUI);

  DECLARE_MESSAGE_MAP()
};
/////////////////////////////////////////////////////////////////////////////
// Файл MainFrm.cpp
// Copyright (c) 1999 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////////////////////////
#include "stdafx.h"
#include "ControlBars.h"
#include "MainFrm.h"
#ifdef _DEBUG
#undef THIS_FILE
static char BASED_CODE THIS_FILE[] = __FILE__;
#endif
/////////////////////////////////////////////////////////////////////////////
// CMainFrame
// Связываем функции с командами и сообщениями
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
  //{{AFX_MSG_MAP(CMainFrame)
  ON_WM_CREATE()
  ON_COMMAND(IDM_VIEWPANE, OnViewPane)
  ON_UPDATE_COMMAND_UI(IDM_VIEWPANE, OnUpdateViewPane)
  ON_CBN_SELCHANGE(IDC_PANE, OnSelChangePane)
  ON_COMMAND(IDM_PANE_2COLUMN, OnPane2column)
  ON_COMMAND(IDM_PANE_3COLUMN, OnPane3column)
  ON_UPDATE_COMMAND_UI(IDM_PANE_3COLUMN, OnUpdatePane3column)
  ON_UPDATE_COMMAND_UI(IDM_PANE_2COLUMN, OnUpdatePane2column)
  ON_UPDATE_COMMAND_UI(ID_INDICATOR_IDLE, OnUpdateIndicatorIdle)
  ON_WM_TIMER()
```

```
  //}}AFX_MSG_MAP
  ON_COMMAND_RANGE(ID_STYLE_LEFT, ID_STYLE_JUSTIFIED, OnStyle)
  ON_UPDATE_COMMAND_UI_RANGE(ID_STYLE_LEFT, ID_STYLE_JUSTIFIED,
↳OnUpdateStyle)
  ON_COMMAND_RANGE(ID_PANEERASE, ID_PANEOVAL, OnPane)
  ON_UPDATE_COMMAND_UI_RANGE(ID_PANEERASE, ID_PANEOVAL, OnUpdatePane)
END_MESSAGE_MAP()
/////////////////////////////////////////////////////////////////////////////
// Массивы идентификаторов для инициализации панелей элементов управления

static UINT BASED_CODE toolbar[] =
{
  // тот же порядок, что и в файле 'toolbar.bmp'
  ID_FILE_NEW,
  ID_FILE_OPEN,
  ID_FILE_SAVE,
    ID_SEPARATOR,
  ID_EDIT_CUT,
  ID_EDIT_COPY,
  ID_EDIT_PASTE,
    ID_SEPARATOR,
  ID_SEPARATOR,           // Резервируем для combo box
    ID_SEPARATOR,
  ID_FILE_PRINT,
  ID_APP_ABOUT,
  ID_CONTEXT_HELP
};

static UINT BASED_CODE pane[] =
{
  // Тот же порядок, что и в файле pane.bmp
  ID_PANEERASE,
  ID_PANEPEN,
  ID_PANESELECT,
  ID_PANEBRUSH,
  ID_PANESPRAY,
  ID_PANEPAINT,
  ID_PANELINE,
```

```
  ID_PANEEYEDROP,
  ID_PANEMAG,
  ID_PANERECT,
  ID_PANEROUND,
  ID_PANEOVAL
};

static UINT BASED_CODE indicators[] =
{
  ID_SEPARATOR,           // Индикатор выбираемой команды
  ID_INDICATOR_NUM,
  ID_INDICATOR_IDLE,
  ID_INDICATOR_TIME
};
/////////////////////////////////////////////////////////////
// CMainFrame
CMainFrame::CMainFrame()
{
  m_nPaneCol = 3;
}
CMainFrame::~CMainFrame()
{
}
// Перегружаем функцию PreCreateWindow для задания своего класса
// окна, задавая иной фон и отказываясь от стилей
// CS_VREDRAW/CS_HREDRAW, чтобы исключить перерисовку панелей
// элементов управления при изменении размеров главного окна
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
  cs.lpszClass = AfxRegisterWndClass(CS_DBLCLKS,
    AfxGetApp()->LoadStandardCursor(IDC_ARROW), (HBRUSH)(COLOR_WINDOW+1),
    AfxGetApp()->LoadIcon(IDR_MAINFRAME));

  return TRUE;
}

int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
```

```
  if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
    return -1;

  if (!CreateToolBar())
    return -1;
  if (!CreatePaneBar())
    return -1;
  if (!CreateStatusBar())
    return -1;

  if (!m_wndDlgBar.Create(this, IDD_VIEWSELECT,
    CBRS_LEFT|CBRS_TOOLTIPS|CBRS_FLYBY, IDD_VIEWSELECT))
  {
    TRACE0("Ошибка при создании панели \n");
    return -1;
  }

  // Разрешаем фиксацию панели инструментов
  EnableDocking(CBRS_ALIGN_ANY);

  m_wndPaneBar.EnableDocking(CBRS_ALIGN_ANY);
  DockControlBar(&m_wndPaneBar, AFX_IDW_DOCKBAR_TOP);

  // Размещаем панель инструментов в задаваемом месте экрана
  CPoint pt(::GetSystemMetrics(SM_CXSCREEN) - 200,
    ::GetSystemMetrics(SM_CYSCREEN) / 3);

  FloatControlBar(&m_wndPaneBar, pt);

  // Создаем таймер для смены содержимого области строки состояния
  SetTimer(0, 1000, NULL);

  return 0;
}

// Разбивка панели на столбцы - находим
// конечные кнопки каждой строки
void CMainFrame::SetColumns(UINT nColumns)
```

```
{
  int nCount = m_wndPaneBar.GetToolBarCtrl().GetButtonCount();

  for (int i = 0; i < nCount; i++)
  {
    UINT nStyle = m_wndPaneBar.GetButtonStyle(i);
    BOOL bWrap = (((i + 1) % nColumns) == 0);
    if (bWrap)
      nStyle |= TBBS_WRAPPED;
    else
      nStyle &= ~TBBS_WRAPPED;
    m_wndPaneBar.SetButtonStyle(i, nStyle);
  }

  m_wndPaneBar.Invalidate();
  m_wndPaneBar.GetParentFrame()->RecalcLayout();
}

BOOL CMainFrame::CreateToolBar()
{
  const int nDropHeight = 100;

  // Вставлять комбинированный список легче, если панель
  // инструментов или любой другой элемент управления
  // формируется на основе массива команд,
  // когда место для элемента заранее зарезервировано
  if (!m_wndToolBar.Create(this, WS_CHILD|WS_VISIBLE|CBRS_TOP|
      CBRS_TOOLTIPS|CBRS_FLYBY) ||
    !m_wndToolBar.LoadBitmap(IDB_TOOLBAR) ||
    !m_wndToolBar.SetButtons(toolbar, sizeof(toolbar)/sizeof(UINT)))
  {
    TRACE0("Ошибка при создании панели\n");
    return FALSE;       // ошибка при создании
  }

  //
  // Создание комбинированного списка
  //
```

```
// Задаем ширину для списка и связываем его
// идентификатор с местом на панели
m_wndToolBar.SetButtonInfo(8, IDW_COMBO, TBBS_SEPARATOR, 100);

// Microsoft требует (и не следует сама!), чтобы расстояние между
// списками и кнопками было 12 пикселей в ширину
m_wndToolBar.SetButtonInfo(7, ID_SEPARATOR, TBBS_SEPARATOR, 12);
m_wndToolBar.SetButtonInfo(9, ID_SEPARATOR, TBBS_SEPARATOR, 12);

CRect rect;
m_wndToolBar.GetItemRect(8, &rect);
rect.top = 2;
rect.bottom = rect.top + nDropHeight;
if (!m_comboBox.Create(CBS_NOINTEGRALHEIGHT|
    CBS_DROPDOWNLIST|WS_VISIBLE|WS_TABSTOP,
    rect, &m_wndToolBar, IDW_COMBO))
{
  TRACE0("Ошибка при создании комбинированного списка\n");
  return FALSE;
}

m_comboBox.AddString(_T("Влево"));
m_comboBox.AddString(_T("По центру"));
m_comboBox.AddString(_T("Вправо"));
m_comboBox.AddString(_T("По ширине"));

// Про работу со шрифтами тоже говорить будем не скоро,
// пока поверьте на слово
LOGFONT logFont;
memset(&logFont, 0, sizeof(logFont));

if (!::GetSystemMetrics(SM_DBCSENABLED))
{
  logFont.lfHeight = -12;
  logFont.lfWeight = FW_BOLD;
  logFont.lfPitchAndFamily = VARIABLE_PITCH | FF_SWISS;
  lstrcpy(logFont.lfFaceName, _T("MS Sans Serif"));
  if (!m_font.CreateFontIndirect(&logFont))
```

```
      TRACE0("Невозможно создать шрифт для списка\n");
    else
      m_comboBox.SetFont(&m_font);
  }
  else
  {
    m_font.Attach(::GetStockObject(SYSTEM_FONT));
    m_comboBox.SetFont(&m_font);
  }

  return TRUE;
}

BOOL CMainFrame::CreatePaneBar()
{
  if (!m_wndPaneBar.Create(this, WS_CHILD | WS_VISIBLE | CBRS_SIZE_FIXED
      | CBRS_TOP | CBRS_TOOLTIPS, ID_PANEBAR) ||
    !m_wndPaneBar.LoadBitmap(IDB_PANE) ||
    !m_wndPaneBar.SetButtons(pane,
      sizeof(pane)/sizeof(UINT)))
  {
    TRACE0("Ошибка при создании панели\n");
    return FALSE;
  }

  //
  // Добавляем к изображениям на кнопках текстовые строки
  //

  // Узнаем и запоминаем размер изображений
  CRect imageRect;
  m_wndPaneBar.GetItemRect(0, &imageRect);

  // Добавляем по строке на каждую кнопку,
  // если строк будет меньше - все оставшиеся кнопки
  // будут подписаны ПЕРВОЙ ЗАДАННОЙ СТРОКОЙ !!!
  for (int index = ID_PANEERASE; index <= ID_PANEOVAL; index++)
  {
```

```
    CString szPane;
    szPane.LoadString(index);

    // Строки хранятся в следующем виде:
    //   - сообщение для строки состояния о возможном выборе,
    //   - символ '\n',
    //   - краткое наименование для всплывающих подсказок.
    // Именно последнее мы и используем
    m_wndPaneBar.SetButtonText(index - ID_PANEERASE,
      szPane.Mid(szPane.Find('\n')+1));
  }

  // После каждой вставки текста автоматически происходило
  // изменение размеров, но об этом знает Windows,
  // объект класса CToolBar об этом знать не знает -
  // сообщим ему об этом, памятуя, что размер изображений
  // остался без изменений; константы 7 и 6 - требования Microsoft
  CRect buttonRect;
  m_wndPaneBar.GetItemRect(0, &buttonRect);
    m_wndPaneBar.SetSizes(CSize(buttonRect.Width(),
                          buttonRect.Height()),
    CSize(imageRect.Width()-7, imageRect.Height()-6));

  // Заголовок для будущего окна, если панель будет
  // плавающей (будет-будет!)
  m_wndPaneBar.SetWindowText(_T("Инструменты"));

  // Сразу настраиваем число столбцов
  SetColumns(m_nPaneCol);

  return TRUE;
}

BOOL CMainFrame::CreateStatusBar()
{
  if (!m_wndStatusBar.Create(this) ||
    !m_wndStatusBar.SetIndicators(indicators,
      sizeof(indicators)/sizeof(UINT)))
```

```
  {
    TRACE0("Ошибка при создании панели \n");
    return FALSE;
  }

  return TRUE;
}
...
/////////////////////////////////////////////////////////////////////////////
// Обработчики сообщений класса CMainFrame

// Управление отображением панели инструментов рисования
void CMainFrame::OnViewPane()
{
  BOOL bVisible = ((m_wndPaneBar.GetStyle() & WS_VISIBLE) != 0);

  ShowControlBar(&m_wndPaneBar, !bVisible, FALSE);
  RecalcLayout();
}

void CMainFrame::OnUpdateViewPane(CCmdUI* pCmdUI)
{
  BOOL bVisible = ((m_wndPaneBar.GetStyle() & WS_VISIBLE) != 0);
  pCmdUI->SetCheck(bVisible);
}

// Обработчик изменения выбора инструмента рисования
void CMainFrame::OnSelChangePane()
{
  CString strText;
  CString strItem;
  CComboBox* pCBox = (CComboBox*)m_wndDlgBar.GetDlgItem(IDC_PANE);
  int nIndex = pCBox->GetCurSel();
  if (nIndex == CB_ERR)
    return;
  pCBox->GetLBText(nIndex, strItem);
  AfxFormatString1(strText, IDS_SELECTED_PROMPT, (LPCTSTR)strItem);
  SetMessageText(strText);
}
```

```
// Обработчики команд смены числа
// столбцов панели инструментов рисования
void CMainFrame::OnPane2column()
{
  SetColumns(m_nPaneCol = 2);
}

void CMainFrame::OnPane3column()
{
  SetColumns(m_nPaneCol = 3);
}

void CMainFrame::OnUpdatePane2column(CCmdUI* pCmdUI)
{
  pCmdUI->SetCheck(m_nPaneCol == 2);
}

void CMainFrame::OnUpdatePane3column(CCmdUI* pCmdUI)
{
  pCmdUI->SetCheck(m_nPaneCol == 3);
}

// Каждый раз меняем символ в области строки состояния.
// Кстати, эта функция при отсутствии каких-либо действий
// вызывается примерно один раз в секунду, а при манипуляциях с мышью -
// практически постоянно.
void CMainFrame::OnUpdateIndicatorIdle(CCmdUI* pCmdUI)
{
  static BOOL bIdle = FALSE;
  pCmdUI->SetText(bIdle ? " " : "\x95");
  bIdle = !bIdle;
}

// Обработчик таймера
void CMainFrame::OnTimer(UINT nIDEvent)
{
  m_wndStatusBar.SetPaneText(m_wndStatusBar.CommandToIndex
↳(ID_INDICATOR_TIME),
    CTime::GetCurrentTime().Format("%H:%M:%S"));
```

```
    CFrameWnd::OnTimer(nIDEvent);
}

// Обработчик смены выбора стилей выравнивания
void CMainFrame::OnStyle(UINT nID)
{
  m_comboBox.SetCurSel((int)(nID-ID_STYLE_LEFT));
}

void CMainFrame::OnUpdateStyle(CCmdUI* pCmdUI)
{
  UINT nIndex = m_comboBox.GetCurSel();
  pCmdUI->SetCheck((UINT)(ID_STYLE_LEFT+nIndex) == pCmdUI->m_nID);
}

// Обработчик смены выбора инструментов рисования
void CMainFrame::OnPane(UINT nID)
{
  CComboBox* pCBox = (CComboBox*)m_wndDlgBar.GetDlgItem(IDC_PANE);
  pCBox->SetCurSel((int)(nID - ID_PANEERASE));
  OnSelChangePane();
}

void CMainFrame::OnUpdatePane(CCmdUI* pCmdUI)
{
  CComboBox* pCBox = (CComboBox*)m_wndDlgBar.GetDlgItem(IDC_PANE);
  UINT nIndex = pCBox->GetCurSel();
  pCmdUI->SetCheck((UINT)(ID_PANEERASE + nIndex) == pCmdUI->m_nID);
}

/////////////////////////////////////////////////////////////////////////////
// Файл ControlBars.cpp
// Copyright (c) 1999 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////////////////////////
...
/////////////////////////////////////////////////////////////////////////////
//
// Menu
```

```
//
IDR_MAINFRAME MENU PRELOAD DISCARDABLE
BEGIN
    POPUP "&Файл"
    BEGIN
        MENUITEM "Выход",                       ID_APP_EXIT
    END
    POPUP "&Вид"
    BEGIN
        MENUITEM "Панель инструментов",         ID_VIEW_TOOLBAR
        MENUITEM "Инструменты рисования",       IDM_VIEWPANE, CHECKED
        MENUITEM "Строка состояния",            ID_VIEW_STATUS_BAR
    END
    POPUP "&Стиль"
    BEGIN
        MENUITEM "Влево",                       ID_STYLE_LEFT
        MENUITEM "По центру",                   ID_STYLE_CENTERED
        MENUITEM "Вправо",                      ID_STYLE_RIGHT
        MENUITEM "По ширине",                   ID_STYLE_JUSTIFIED
    END
    POPUP "&Инструмент"
    BEGIN
        MENUITEM "&2 столбца",                  IDM_PANE_2COLUMN
        MENUITEM "&3 столбца",                  IDM_PANE_3COLUMN
    END
    POPUP "&?"
    BEGIN
        MENUITEM "О программе ...",             ID_APP_ABOUT
    END
END

/////////////////////////////////////////////////////////////////////
//
// Dialog
//

IDD_VIEWSELECT DIALOG DISCARDABLE  0, 0, 69, 84
STYLE WS_CHILD
FONT 8, "MS Sans Serif"
```

```
BEGIN
    GROUPBOX        "",IDC_STATIC,1,14,66,34
    CONTROL         "Команды",
                    ID_VIEW_TOOLBAR,"Button",BS_AUTOCHECKBOX |
                    WS_TABSTOP,5,25,59,10
    CONTROL         "Инструменты",
                    IDM_VIEWPANE,"Button",BS_AUTOCHECKBOX |
                    WS_TABSTOP,5,35,60,10
    COMBOBOX        IDC_PANE,2,55,65,50,CBS_DROPDOWNLIST | WS_VSCROLL
END

IDD_ABOUTBOX DIALOG DISCARDABLE  0, 0, 217, 55
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "О программе CTRLBARS"
FONT 8, "MS Sans Serif"
BEGIN
    ICON            IDR_MAINFRAME,IDC_STATIC,11,11,21,20
    LTEXT           "Приложение CTRLBARS  Версия 1.0",
                    IDC_STATIC, 43,10,130,8
    LTEXT           "Copyright © 1999 Мешков А., Тихомиров Ю.",
                    IDC_STATIC, 35,39,147,8
    DEFPUSHBUTTON   "OK",IDOK,176,6,32,14,WS_GROUP
END
...
```

Рис. 11.2. Главное окно приложения ControlBars

Все функции с префиксом OnUpdate используются во время простоя, т. е. в момент, когда очередь сообщений потока пуста. Все такие функции расположены в объявлении класса за функциями обработки команд или событий, и позволяют воздействовать на соответствующие им элементы управления.

В главное окно приложения помещены следующие панели элементов управления:

- Панель инструментов (команд) — привязанная к верхней части окна без возможности перемещения. В нее вставлен комбинированный список для задания стиля выравнивания (обработчик его сообщений — функция CreateToolBar), выбор элемента которого отслеживается функцией OnStyle.
- Панель инструментов рисования — плавающая с возможностью фиксации. Все кнопки имеют кроме рисунка еще и текст (функция CreatePaneBar). Выбор одного из инструментов сопровождается вызовом функции OnPane.
- Панель диалога — привязанная к левой части окна без возможности перемещения. Содержит элементы управления: статическая рамка, два флажка для отображения/скрытия панели инструментов и панели инструментов рисования (функция OnViewPane), и, наконец, комбинированный список отображения/выбора инструмента рисования, выбор элемента которого отслеживается функцией OnSelChangePane.

Примечание

Идентификатор команды для управления режимом показа панели инструментов равен ID_VIEW_TOOLBAR, который интерпретируется как команда и управляет панелями с идентификаторами AFX_IDW_TOOLBAR, так что специального кода создавать не надо — все делает библиотека MFC.

- Строка состояния — состоящая из четырех областей: для подсказки, отображения режима NUMLOCK, для демонстрации обработки во время простоя (функция OnUpdateIndicatorIdle) и для отображения текущего времени примерно раз в секунду по сигналу от таймера (функция OnTimer).

Осталось рассмотреть меню, в котором имеются, кроме общих, следующие пункты:

- Меню "Вид" — управляет отображением/скрытием панелей управления (кроме панели диалога) и строкой состояния.

Примечание

Идентификатор команды для управления режимом показа строки состояния равен ID_VIEW_STATUSBAR, и управляет строками состояния с идентификаторами AFX_IDW_STATUSBAR, так что специального кода создавать не надо — все делает библиотека MFC.

- Меню "Стиль" — управляет выбором стиля выравнивания. Используется, как и в случае панели инструментов (команд), функция OnStyle.
- Меню "Инструмент" — обеспечивает задание количества столбцов для отображения панели инструментов рисования (функции CPane2colomn и CPane3colomn).

Глава 12

Меню

Что такое меню, как его можно создать и с ним работать? Начальные сведения об этом были приведены в главе 10, здесь же мы постараемся ответить на эти вопросы более подробно. Но, прежде чем переходить к изучению средств Windows и библиотеки MFC, необходимо рассмотреть стандартные типы меню, которые можно создавать в приложениях Windows, и их основные характеристики.

Основные типы меню

Меню обычно располагается в верхней части перекрывающегося или всплывающего окна под полосой заголовка, и представляют собой словесные или символические опции, дающие возможность манипулировать особенностями программы. Такое меню называется *главное меню приложения*, и содержит отдельные элементы, расположенные в *полосе меню* (menu bar), — верхний уровень иерархии (рис. 12.1).

Примечание

В общем случае дочерние окна могут не иметь полосы меню.

Рис. 12.1. Заголовок и меню приложения WordPad

Элементы меню могут иметь различные типы и содержать ряд модификаторов (рис. 12.2 и 12.3):

- Акселератор (клавиша вызова) — сообщает пользователю, что вместо вызова меню можно вызвать команду с помощью акселератора
- Стрелка — показывает, что при выборе данного элемента появится подменю
- Разделительная черта — отделяет одну группу элементов меню от другой
- Маркер — означает, что данный параметр меню выбран
- Многоточие — сообщает пользователю о том, что при выборе данного элемента на экран будет выведен блок диалога
- Полутоновая надпись — говорит, что в текущем состоянии данный элемент меню недоступен
- Знак подчеркивания — означает, что данный элемент меню можно вызвать с помощью акселератора, в который входит эта буква. Такие буквы называются *мнемоническими*

Рис. 12.2. Раскрытое меню

Рис. 12.3. Команда, имеющая подменю

Каждое окно, имеющее заголовок, может предоставлять доступ к *системному* меню, которое вызывается щелчком левой кнопки мыши на значке, расположенном в левой части заголовка окна (рис. 12.4) и содержит стандартные команды. В Windows 95/98 системное меню появляется, если щелкнуть правой кнопкой мыши в полосе заголовка или на значке свернутого окна.

Windows предоставляет меню специального типа, которое можно создать в любом месте экрана. Такое меню называется *контекстным* или *плавающим* (floating popup menu) и не привязано к полосе меню (рис. 12.5). Оно ассоциируется с некоторой областью окна или объектом, например, значком. Контекстное меню в некоторых случаях удобнее других, т. к. его содержимое зависит от того, для какого объекта оно было создано, оно обычно выводится на экран щелчком правой кнопки мыши.

Рис. 12.4. Системное меню

Рис. 12.5. Контекстное меню рабочей области окна приложения WordPad

Теперь, перечислив основные типы меню, используемые в Windows, рассмотрим возможности, которые предоставляет библиотека MFC для работы с ними.

Точно так же, как в Win32 API есть функции, специально предназначенные для работы с меню, в библиотеке MFC есть адекватная замена этих функций, реализованная в виде класса *CMenu*, полностью вобравшего их в себя.

Класс CMenu

Это класс, который инкапсулирует объект Windows, определяемый дескриптором HMENU, и предоставляет функции для работы с меню. С помощью этого класса приложение может создавать меню, даже не имея его шаблона, добавлять или удалять элементы или подменю, управлять их состоянием — активизировать или блокировать, отмечать команды и т. д.

Рассмотрим основные члены этого класса:

CMenu::**m_hMenu** —

определяет дескриптор HMENU меню Windows, присоединенного к объекту класса, для создания которого, как обычно, используется конструктор.

CMenu::**CMenu**( ) —

конструктор. Следует совершенно четко представлять, что при создании объекта само меню — объект Windows — не создается автоматически. Для того чтобы это реализовать, необходимо вызвать одну из функций, отвечающих за создание или загрузку меню.

Для создания "пустого" меню, т. е. меню, не содержащего ни одной строчки и ни одного подменю, достаточно воспользоваться функцией:

BOOL CMenu::**CreateMenu**( ) —

создает пустое меню Windows и присоединяет его к объекту класса. Добавлять в него элементы можно, используя функции AppendMenu и InsertMenu. При успешном создании меню функция возвращает TRUE, FALSE — в случае неудачи.

Для того чтобы добавить в существующее меню некоторое подменю, в классе реализована функция:

BOOL CMenu::**CreatePopupMenu**( ) —

создает пустое подменю. Добавить в него элементы можно с помощью тех же функций — AppendMenu и InsertMenu. При успешном завершении функция возвращает TRUE и FALSE в случае неудачи.

Если создание меню производится на основе шаблона, то можно воспользоваться функциями

BOOL CMenu::**LoadMenu**(LPCTSTR lpszResourceName)

и

BOOL CMenu::**LoadMenu**(UINT nIDResource) —

загружают ресурс меню из исполняемого файла приложения и присоединяют его к объекту. В качестве параметра используется либо указатель на текстовую строку lpszResourceName, содержащую имя ресурса загружаемого меню, либо его числовой идентификатор nIDResource.

Аналогичная функция существует для меню, создаваемого на основе шаблона в оперативной памяти:

BOOL CMenu::**LoadMenuIndirect**(const void *lpMenuTemplate) —

загружает ресурс меню из шаблона в памяти и присоединяет его к объекту CMenu. В качестве параметра принимает адрес этого шаблона, который состоит из единственного заголовка (структура MENUITEMTEMPLATEHEADER) и одного или нескольких наборов элементов меню (структура MENUITEMTEMPLATE). При успешном завершении функция возвращает TRUE, а в случае неудачи — FALSE.

Примечание

Согласно документации, в Windows 95/98 дополнительно можно использовать расширенный шаблон, который вместо перечисленных структур использует, соответственно, структуры MENUEX_TEMPLATE_HEADER и MENUEX_TEMPLATE_ITEM — фактически же эти структуры не представлены ни в одном заголовочном файле, и работать с ними, естественно, нельзя.

Рассмотрим состав указанных структур.

Структура MENUITEMTEMPLATEHEADER определяет заголовок шаблона меню:

```
typedef struct{
  WORD versionNumber;
  WORD offset;
} MENUITEMTEMPLATEHEADER;
```

versionNumber — значение этого поля должно быть равно нулю (хотя, как утверждает документация, оно должно определять номер версии); offset — определяет смещение (в байтах) начала списка элементов меню от конца данной структуры (обычно его значение равно нулю, и список начинается сразу же после заголовка).

Структура MENUITEMTEMPLATE определяет элемент меню в шаблоне:

```
typedef struct{
  WORD mtOption;
  WORD mtID;
  WCHAR mtString[1];
} MENUITEMTEMPLATE;
```

mtOption — задает один или несколько флагов, из списка доступных, для элемента меню и должно содержать MF_END для последнего элемента подменю; mtID — определяет идентификатор сообщения (команды) элемента меню (для подменю это поле не исполь-

зуется); mtString — определяет строку Unicode (с нулевым завершающим символом) с текстом элемента меню.

Примечание

Список флагов приведен в разделе "Создание меню на основе шаблона".

Если по каким-либо причинам понадобилось меню, которое не присоединяется ни к одному окну, то при завершении работы с ним следует воспользоваться функцией:

BOOL CMenu::**DestroyMenu**( ) —

перед завершением работы приложения удаляет меню и возвращает системе занимаемые им ресурсы.

Для изменения состава меню, созданного функциями CreateMenuIndirect и CreateMenu, в классе предусмотрена специальная группа функций.

Добавление элементов в меню производится функциями AppendMenu и InsertMenu:

```
BOOL CMenu::AppendMenu(
    UINT nFlags,
    UINT nIDNewItem = 0,
    LPCTSTR lpszNewItem = NULL)
```

и

```
BOOL CMenu::AppendMenu(
    UINT nFlags,
    UINT nIDNewItem = 0,
    const CBitmap *pBmp) —
```

добавляют новый элемент в конец меню. Состояние элемента задается параметром nFlags, который может содержать одно или несколько из следующих значений:

MF_CHECKED, MF_UNCHECKED	При изображении элемент меню отмечается (MF_CHECKED) или не отмечается (MF_UNCHECKED) галочкой
MF_DISABLED, MF_ENABLED	Элемент меню блокирован (MF_DISABLED), но изображается обычным (черным) цветом, или доступен (MF_ENABLED)
MF_GRAYED	Элемент меню блокирован и изображается серым цветом
MF_MENUBREAK	Элемент меню верхнего уровня выводится с новой строки, а элемент подменю — в новом столбце без разделительной вертикальной линии
MF_MENUBARBREAK	Аналогично MF_MENUBREAK, но дополнительно новый столбец отделяется вертикальной линией
MF_OWNERDRAW	Меню само отвечает за изображение своего элемента
MF_POPUP	Элемент меню имеет ассоциированное с ним подменю, дескриптор которого задается в параметре nIDNewItem

MF_SEPARATOR	Элемент представляет собой горизонтальную разделительную линию; этот флаг не может комбинироваться с MF_POPUP и MF_DISABLED; при установленном данном флаге все другие игнорируются
MF_STRING	Элемент представляет собой текстовую строку

При использовании этих флагов следует иметь в виду, что нельзя комбинировать следующие флаги:

- MF_DISABLED, MF_ENABLED и MF_GRAYED
- MF_STRING, MF_OWNERDRAW, MF_SEPARATOR и битовый массив (bitmap)
- MF_MENUBREAK и MF_MENUBARBREAK
- MF_CHECKED и MF_UNCHECKED

Параметр nIDNewItem определяет идентификатор команды нового элемента, а если nFlags установлен в MF_POPUP, — то дескриптор (HMENU) подменю. Через этот параметр передается дополнительная информация о содержимом нового элемента, если nFlags принимает одно из следующих значений:

nFlags	lpszNewItem
MF_OWNERDRAW	Содержит 32-разрядное значение, которое приложение может использовать в качестве дополнительных данных, ассоциированных с элементом меню при обработке сообщений WM_MEASUREITEM и WM_DRAWITEM
MF_SEPARATOR	Значение игнорируется
MF_STRING	Содержит текстовую строку

Параметр pBmp определяет указатель на объект класса CBitmap, который может использоваться в качестве содержимого элемента меню вместо текстовой строки. При этом параметр nFlags *не может* принимать значения MF_OWNERDRAW и MF_STRING.

```
BOOL CMenu::InsertMenu(
    UINT nPosition,
    UINT nFlags,
    UINT nIDNewItem = 0,
    LPCTSTR lpszNewItem = NULL)
```

и

```
BOOL CMenu::InsertMenu(
    UINT nPosition,
    UINT nFlags,
    UINT nIDNewItem = 0,
    const CBitmap *pBmp) —
```

вставляют новый элемент в позицию, определяемую параметром nPosition, и сдвигают вниз существующие элементы. Значения всех параметров, кроме nFlags, такие же, как и для функции AppendMenu. Параметр nFlags дополнительно используется для интерпретации значения nParam:

nFlags	nPosition
MF_BYCOMMAND	Определяет идентификатор элемента меню, перед которым вставляется новый элемент
MF_BYPOSITION	Определяет порядковый номер элемента меню, перед которым вставляется новый элемент; если установить nPosition = –1, то элемент будет добавлен в конец меню

Если требуется не вставлять новый, а заменить существующий элемент меню, то можно воспользоваться функциями:

BOOL CMenu::**ModifyMenu**(
UINT nPosition,
UINT nFlags,
UINT nIDNewItem,
LPCTSTR lpszNewItem)

и

BOOL CMenu::**ModifyMenu**(
UINT nPosition,
UINT nFlags,
UINT nIDNewItem,
const CBitmap *pBmp) —

параметры этих функций идентичны параметрам функций AppendMenu и InsertMenu за исключением того, что новый элемент не вставляется в меню, а заменяет тот, на который указывает параметр nPosition.

Для удаления элемента из меню в классе предусмотрены две функции:

BOOL CMenu::**DeleteMenu**(
UINT nPosition,
UINT nFlags)

и

BOOL CMenu::**RemoveMenu**(
UINT nPosition,
UINT nFlags) —

удаляют элемент меню, задаваемый параметром nPosition, значение которого интерпретируется аналогично функции InsertMenu. Различие между функциями заключается в том, что при удалении подменю функцией DeleteMenu все связанные с ним ресурсы освобождаются, а в случае RemoveMenu — нет, и можно снова воспользоваться удаленным элементом (по его идентификатору).

Для изменения состояния элемента меню используется функция:

UINT CMenu::**EnableMenuItem**(
UINT nIDEnableItem,
UINT nEnable) —

позволяет установить элемент меню nIDEnableItem в одно из трех состояний (задаваемых параметром nEnable): доступное (MF_ENABLED), запрещенное (MF_DISABLED) или забло-

кированное (MF_GRAYED). Эти значения должны комбинироваться с одним из флагов MF_BYCOMMAND или MF_BYPOSITION, которые определяют, каким образом задается элемент — идентификатором или порядковым номером. Функция возвращает предыдущее состояние элемента или –1, если элемент не существует.

У элемента можно проставить маркер с помощью функций:

UINT CMenu::**CheckMenuItem**(
UINT nIDCheckItem,
UINT nCheck) —

устанавливает или сбрасывает галочку у элемента меню. Параметр nCheck может принимать значение MF_CHECKED или MF_UNCHECKED, как обычно в комбинации с флагами MF_BYCOMMAND или MF_BYPOSITION, которые определяют интерпретацию параметра nIDChecked — идентификатор или порядковый номер. Функция возвращает предыдущее состояние элемента или -1, если элемент не существует.

BOOL CMenu::**CheckMenuRadioItem**(
UINT nIDFirst,
UINT nIDLast,
UINT nIDItem,
UINT nFlags) —

помечает определенный элемент меню в группе переключателей, одновременно снимая маркеры остальных элементов группы. Для отметки элемента вместо галочки используется кружок. Параметры nIDFirst и nIDLast определяют, соответственно, первый и последний элемент в группе переключателей. Сам отмечаемый элемент задается параметром nIDItem. Параметр nFlags определяет интерпретацию этих параметров и может принимать одно из значений MF_BYCOMMAND или MF_BYPOSITION.

Вместо стандартных галочки или кружка, используемых для отметки элемента, в меню можно поместить произвольный рисунок, задаваемый битовым массивом:

BOOL CMenu::**SetMenuItemBitmaps**(
UINT nPosition,
UINT nFlags,
const CBitmap *pBmpUnchecked,
const CBitmap *pBmpChecked) —

ассоциирует с элементом меню битовые массивы, которые будут выводиться для помеченного (pBmpChecked) и непомеченного (pBmpUnchecked) состояний. Параметры nPosition и nFlags определяют элемент меню по идентификатору или порядковому номеру. Если значение pBmpChecked, либо pBmpUnchecked равно NULL, то для соответствующего атрибута рядом с элементом ничего не отображается, а если оба параметра равны NULL, то используются стандартные галочка и кружок. Используя эту возможность, следует иметь в виду, что при разрушении меню битовые массивы не удаляются автоматически — эта задача возлагается на приложение.

После внесения в меню всех изменений необходимо вызвать функцию:

void CWnd::**DrawMenuBar**( ) —

отвечает за перерисовку полосы меню.

В классе CMenu реализовано несколько функций, которые позволяют получить различную информацию о самом меню и о его элементах.

UINT CMenu::**GetMenuItemCount**( ) —

позволяет получить число элементов меню верхнего уровня и подменю.

UINT CMenu::**GetMenuItemID**(int nPos) —

возвращает идентификатор элемента подменю. Если параметр nPos определяет позицию самого подменю, то функция возвращает –1, а если позицию разделителя (элемент имеет стиль MF_SEPARATOR), возвращается 0.

UINT CMenu::**GetMenuState**(
UINT nID,
UINT nFlags) —

возвращает состояние элемента или число элементов в раскрывающемся меню, что определяется передаваемыми ей параметрами nID и nFlags. Параметр nID определяет элемент меню, для которого запрашивается информация, и его интерпретация зависит от значения, указанного в параметре nFlags. Если там указан флаг MF_BYCOMMAND, то nID определяет идентификатор элемента, а если установлен флаг MF_BYPOSITION — порядковый номер элемента. Для раскрывающегося меню старший байт возвращаемого значения содержит число элементов в нем, младший байт — набор флагов, описывающих раскрывающееся меню. Для меню верхнего уровня возвращаемое значение является набором флагов, описывающих указанный элемент меню. Полный список флагов приведен при описании функции AppendMenu.

int CMenu::**GetMenuString**(
UINT nIDItem,
LPTSTR lpString,
int nMaxCount,
UINT nFlags)

и

int CMenu::GetMenuString(
UINT nIDItem,
CString &rString,
UINT nFlags) —

позволяют получить текстовую строку, соответствующую элементу меню. Возвращают число байт (без завершающего нулевого символа), скопированных в буфер. Параметры: nIDItem и nFlags действуют так же, как для всех предыдущих функций; rString — ссылка на объект класса CString, получающий строку; lpString — адрес буфера, куда должна копироваться строка; nMaxCount — размер буфера. Следует иметь в виду, что символы, не поместившиеся в буфер, будут отброшены.

CMenu* CMenu::**GetSubMenu**(int nPos) —

возвращает указатель на объект CMenu, определяющий подменю, или NULL. Если объект класса CMenu не существует, то создается временный. Параметр nPos задает позицию подменю (начиная с 0).

Кроме перечисленных, в последней на данный момент версии (4.3) библиотеки MFC есть еще три новые функции. Две из них позволяют получить (установить) информацию об элементе меню, установленном по умолчанию:

BOOL CMenu::**SetDefaultItem**(
UINT uItem,
BOOL fByPos = FALSE) —

и

UINT CMenu::**GetDefaultItem**(
UINT gmdiFlags,
BOOL fByPos = FALSE) —

позволяют, соответственно, установить и получить информацию об элементе меню, установленном по умолчанию. Параметр gmdiFlags задает способ, которым функция будет искать элемент меню, и может принимать комбинацию следующих значений:

GMDI_GOINTOPOPUPS	Если элемент меню, установленный по умолчанию, имеет подменю, то поиск производится рекурсивно в этом подменю. В случае, когда в подменю нет элемента, установленного по умолчанию, функция возвращает идентификатор (или позицию) этого имеющего подменю элемента
GMDI_USEDISABLED	Идентификатор (позиция) элемента меню, установленного по умолчанию Если этот флаг не установлен, то идентификаторы заблокированных и/или изображенных серым цветом элементов меню не возвращаются

fByPos — значение, показывающее способ задания элемента меню: если fByPos = FALSE, то элемент меню uItem определяется своим идентификатором, а если fByPos = TRUE, то порядковым номером. Это же относится и к значению, возвращаемому функцией GetDefaultItem.

Третья функция позволяет получить детальную информацию о конкретном элементе меню:

BOOL CMenu::**GetMenuItemInfo**(
UINT nIDItem,
LPMENUITEMINFO lpMenuItemInfo,
BOOL ByPos = FALSE) —

позволяет получить подробную информацию о конкретном элементе меню, задаваемом в параметре nIDItem. В зависимости от значения параметра ByPos, это может быть идентификатор (ByPos = FALSE — установлено по умолчанию), либо порядковый номер (ByPos = TRUE). Информация об элементе заносится в структуру MENUITEMINFO:

```
typedef struct {
    UINT        cbSize;
    UINT        fType;
    UINT        fState;
    UINT        wID;
    HBITMAP     hbmpChecked;
    HBITMAP     hbmpUnchecked;
```

```
    DWORD       dwItemData;
    CHAR        szString[IMEMENUITEM_STRING_SIZE];
    HBITMAP     hbmpItem;
} MEMENUITEMINFO;
```

cbSize — размер структуры в байтах; fType — флаг, характеризующий тип элемента меню — может принимать следующие значения:

IMFT_RADIOCHECK	Элемент меню входит в группу переключателей
IMFT_SEPARATOR	Элемент меню представляет собой горизонтальную разделительную линию
IMFT_SUBMENU	Элемент меню имеет ассоциированное с ним подменю

fState — комбинация одного или нескольких флагов, показывающих текущее состояние элемента меню. Может принимать следующие значения:

MFS_GRAYED	Элемент меню не может быть выбран и отображается серым цветом
MFS_DISABLED	Элемент меню заблокирован
MFS_CHECKED	Элемент меню отмечен галочкой
MFS_HILITE	Элемент меню подсвечен
MFS_ENABLED	Элемент меню разблокирован
MFS_UNCHECKED	Элемент меню не отмечен галочкой
MFS_UNHILITE	Элемент меню не подсвечен
MFS_DEFAULT	Элемент меню, установленный по умолчанию

wID — идентификатор команды (сообщения) элемента меню; hbmpChecked и hbmpUnchecked дескрипторы битовых массивов Windows, для изображения отмеченного и неотмеченного состояний элемента меню; dwItemData — данные, ассоциированные с элементом меню; szString — строка, определяющая текст элемента меню; hbmpItem — дескриптор битового массива Windows для изображения элемента меню.

Примечание

Если посмотреть исходные тексты, то можно обнаружить также функцию SetMenuItemInfo, позволяющую динамически изменять характеристики элементов меню.

Для поддержки работы с контекстными меню в классе реализована функция:

BOOL CMenu::**TrackPopupMenu**(
UINT nFlags,
int x,
int y,
CWnd *pWnd,
LPCRECT lpRect = 0) —

выводит на экран контекстное меню и создает свой собственный цикл обработки сообщений. Функция не завершается до тех пор, пока работа с меню не будет закончена либо вы-

бором элемента, либо отказом от выбора, после чего меню уничтожается. Параметр nFlags определяет расположение контекстного меню (относительно экрана) и кнопку мыши, с помощью которой должен выполняться выбор. Он может содержать одно из трех значений (для расположения на экране) в комбинации с одним из двух флагов для кнопки мыши.

Значения параметра nFlags для расположения на экране

TMP_CENTERALIGN	Центрирование относительно координаты, заданной параметром x
TMP_LEFTALIGN	Выравнивание по левой границе относительно координаты, заданной параметром x
TMP_RIGHTALIGN	выравнивание по правой границе относительно координаты, заданной параметром x

Значения параметра nFlags для кнопки мыши

TMP_LEFTBUTTON	Использование левой кнопки мыши
TMP_RIGHTBUTTON	Использование правой кнопки мыши

Параметр pWnd задает окно, которое получит сообщение WM_COMMAND после того, как пользователь сделает выбор в контекстном меню, а lpRect является указателем на структуру типа RECT (или объект класса CRect), определяющую координаты прямоугольной области, в которой пользователь может выполнять выбор из меню. Если щелчок мышью будет сделан вне этой области, контекстное меню исчезнет с экрана, что эквивалентно отказу от выбора. Если для него задать значение NULL, то размеры и расположение этой прямоугольной области будут совпадать с размерами контекстного меню.

В классе представлены две виртуальные функции, которые произвольным образом позволяют изменять внешний вид меню:

Примечание

Процесс *самоотображения* (owner-draw) и используемые структуры подробно описаны в главе 23.

virtual void CMenu::**MeasureItem**(
LPMEASUREITEMSTRUCT lpMeasureItemStruct) —

вызывается библиотекой MFC, когда создается меню, имеющее стиль самоотображения (owner-draw). В качестве параметра в функцию передается указатель на структуру MEASUREITEMSTRUCT, с помощью которой можно настроить размеры меню.

virtual void CMenu::**DrawItem**(LPDRAWITEMSTRUCT lpDrawItemStruct) —

вызывается библиотекой MFC, когда создается меню, имеющее стиль самоотображения. В качестве параметра в функцию передается указатель на структуру DRAWITEMSTRUCT, которая содержит информацию для рисования элемента меню.

При использовании стиля самоотображения обе функции должны быть обязательно переопределены, т. к. их реализация в классе CMenu не содержит кода.

Две оставшиеся функции класса предназначены для работы со справочной системой:

DWORD CMenu::**GetMenuContextHelpId**( )

и

BOOL CMenu::**SetMenuContextHelpId**(DWORD dwContextHelpId) —

позволяют получить или установить идентификатор контекстной справки, ассоциированной с меню.

Теперь, после теоретического знакомства с возможностями, заложенными в классе CMenu, можно переходить к практическим аспектам и тонкостям программирования при построении и работе с меню.

В системе Windows реализованы три способа создания меню:

- На основе шаблона, задаваемого в файле ресурса
- Динамическое создание при помощи специальных функций
- На основе шаблона в оперативной памяти

Рассмотрим каждый из этих способов более подробно.

Создание меню на основе шаблона

В главе 10 был приведен код из файла ресурсов, который создавал меню для программы MDI. Не будем подробно останавливаться на этом вопросе, а приведем только описание атрибутов, определяющих внешний вид и поведение строки меню (рис. 12.6).

Рис. 12.6. Атрибуты меню в окне свойств, открытом в режиме настройки параметров меню

Атрибут	Описание
Separator	Элемент представляет собой горизонтальную разделительную линию, служащую для разбиения элементов меню на группы
Checked	При выводе на экран элемент меню отмечается слева галочкой

(окончание)

Атрибут	Описание
Pop-up	Элемент определяет подменю
Grayed	Элемент отображается серым цветом и находится в заблокированном состоянии; несовместим с атрибутом **Inactive**
Help	Элемент выравнивается по правому краю полосы меню
Break	Этот атрибут может принимать одно из трех значений: **None** — обычный элемент меню; **Column** — для меню верхнего уровня элемент выводится с новой строки, а для подменю — в новом столбце; **Bar** — дополнительный столбец подменю отделяется вертикальной линией
ID	Идентификатор элемента меню
Caption	Текст, определяющий имя элемента меню; текстовая строка может содержать символы: **&** — буква, перед которой стоит этот знак, будет подчеркнута; **\t** — включает в строку символ табуляции; **\a** — выравнивает текст по правой границе меню
Prompt	Текст, который будет отображаться в строке состояния и (после **\n**) в окне всплывающей подсказки
Inactive	Элемент находится в заблокированном состоянии, но отображается обычным цветом

После того как создан шаблон (одним из способов, описанных в главе 10), необходимо подключить меню к окну приложения. Этот вопрос также рассматривался достаточно подробно, поэтому только перечислим те места, где это можно сделать:

- При регистрации класса окна — когда меню определяется для нескольких окон одного класса:

```
wc.lpszMenuName = "IDR_MAINFRAME";
```

- При создании окна — если для него необходимо меню, отличное от указанного в классе окна:

```
if (!pMainFrame->LoadFrame(IDR_MAINFRAME))
    return FALSE;
```

- При работе в рамках архитектуры "документ/представление" — достаточно, чтобы идентификатор меню совпадал с идентификаторами фрейма, документа и представления. Тогда вопросами подключения меню полностью занимается библиотека классов MFC при создании шаблона документа:

```
pDocTemplate = new CMultiDocTemplate(
                        IDR_NOTETYPE,
```

```
                    RUNTIME_CLASS(CNoteDoc),
                    RUNTIME_CLASS(CNoteFrame),
                    RUNTIME_CLASS(CNoteView));
```

При использовании любого из перечисленных способов подключения меню к окну нет необходимости уничтожать меню перед завершением работы приложения, т. к. библиотека MFC делает это сама.

Меню, созданное в системе Windows, где взаимодействие между объектами осуществляется при помощи сообщений, также работает по этому принципу. Наиболее важным из сообщений, которое передает меню, является WM_COMMAND, означающее, что выбран тот или иной элемент меню. Ниже приведены этапы работы с меню и возникающие при этом сообщения.

Этап работы	Сообщение	Обработчик в MFC
Инициализация меню	WM_INITMENU	OnInitMenu
Вывод всплывающего меню на экран	WM_INITMENUPOPUP	OnInitMenuPopup
Инициализация и вывод на экран всплывающего меню	WM_INITMENU и WM_INITMENUPOPUP	OnInitMenu и OnInitMenuPopup
Нахождение нужного элемента меню	WM_MENUSELECT	OnMenuSelect
Выбор элемента меню	WM_COMMAND	OnCommand
Выбор элемента системного меню	WM_SYSCOMMAND	OnSysCommand

Рассмотрим кратко перечисленные сообщения.

Примечание

Действие приведенных обработчиков справедливо и для двух других способов создания меню.

Когда пользователь активизирует элемент системного меню, сначала вызывается обработчик сообщения WM_SYSCOMMAND:

void CWnd::**OnSysCommand**(
    UINT nID,
    LPARAM lParam)

Первый параметр nID может содержать идентификатор строки системного меню, а также некоторые другие значения:

SC_CLOSE	Удаление окна
SC_HOTKEY	Активизация окна, связанного с акселератором, определенным приложением
SC_HSCROLL	Прокрутка по горизонтали

SC_KEYMENU	Выбор элемента меню при помощи акселератора
SC_MAXIMIZE или SC_ZOOM	Разворачивание окна
SC_MINIMIZE или SC_ICON	Сворачивание окна
SC_MOUSEMOVE	Выбор произведен при помощи мыши
SC_MOVE	Перемещение окна
SC_NEXTWINDOW	Переключение на следующее окно
SC_PREVWINDOW	Переключение на предыдущее окно
SC_RESTORE	Восстановление нормального положения и размеров окна
SC_SCREENSAVE	Запуск приложения-заставки (screen-saver application), определенного в разделе [boot] файла system.ini
SC_SIZE	Изменение размеров окна
SC_TASKLIST	Запуск или активизация приложения Task Manager
SC_VSCROLL	Прокрутка по вертикали

Кроме этих, можно использовать и свои идентификаторы, как показано в рассматриваемом ниже примере.

Второй параметр lParam используется только для значений первого параметра:

nID	Значение параметра
SC_HOTKEY	Содержит идентификатор активизируемого окна
SC_MOUSEMOVE	Младшее слово содержит x-координату, а старшее — y-координату

Перед отображением всплывающего меню (один раз, при активизации), Windows посылает сообщение WM_INITMENU с тем, чтобы приложение могло изменить меню перед тем, как его увидит пользователь. При этом библиотека MFC вызывает обработчик:

void CWnd::**OnInitMenu**(CMenu *pMenu) —

pMenu обработчика задает указатель на объект CMenu, определяющий активизируемое меню. Обработка может заключаться в активизации или деактивизации строк меню, изменении их состояния и т. п.

Когда пользователь выбирает элемент, имеющий подменю, Windows посылает владельцу сообщение WM_INITMENUPOPUP перед отображением всплывающего меню. Библиотека MFC вызывает обработчик, который дает

приложению возможность изменить всплывающее меню перед его отображением:

void CWnd::**OnInitMenuPopup**(
CMenu pPopupMenu,
UINT nIndex,
BOOL bSubMenu) —

pPopupMenu — указатель на объект "всплывающее меню"; nID — индекс всплывающего меню в меню верхнего уровня; bSubMenu — определяет, является ли всплывающее меню системным (TRUE) или нет.

Каждый раз, когда пользователь перемещается от одного элемента меню к другому, Windows посылает сообщение WM_MENUSELECT, которое идентифицирует текущий элемент, на что библиотека MFC реагирует вызовом обработчика:

void CWnd::**OnMenuSelect**(
UINT nItemID,
UINT nFlags,
HMENU hSubMenu)

nItemID — идентификатор выделенного элемента или индекс всплывающего меню; nFlags — содержит комбинацию следующих флагов, определяющих атрибуты элемента меню:

MF_BITMAP	Элемент является битовым массивом
MF_CHECKED	Элемент отмечается галочкой
MF_DISABLED	Элемент является заблокированным, но отображается в нормальном виде
MF_GRAYED	Элемент отображается серым цветом и не может быть выбран
MF_MOUSESELECT	Элемент выделен мышью
MF_OWNERDRAW	Элемент рисуется окном, создавшим меню
MF_POPUP	С данным элементом связано всплывающее меню
MF_SEPARATOR	Элемент является разделителем
MF_SYSMENU	Элемент принадлежит системному меню

Параметр hSubMenu используется в двух случаях, зависящих от состояния параметра nFlags:

nFlags	**Значение параметра**
MF_SYSMENU	Идентифицирует меню, ассоциированное с сообщением
MF_POPUP	Идентифицирует дескриптор меню верхнего уровня

Многие приложения используют это сообщение для отображения дополнительной информации о выбранном элементе меню в строке состояния.

Случай, когда пользователь выбирает элемент меню, а Windows посылает сообщение WM_COMMAND, уже не раз описывался в предыдущих примерах — необходимо либо воспользоваться одним из предопределенных обработчиков стандартных команд меню, либо создать свой, например, так, как было показано в главе 10.

В Windows 95 появилось еще одно сообщение, предназначенное для работы с контекстными меню — WM_CONTEXTMENU.

На наш взгляд, нет необходимости дольше задерживаться на этом вопросе, поскольку при использовании шаблонов подавляющую часть работы выполняет операционная система. И все необходимые действия для работы с меню представляют относительную сложность только при построении самого первого меню.

Динамическое создание меню

В приведенном ниже примере показаны основные принципы работы с меню "в динамическом режиме". В нем использовано большинство функций класса CMenu, предназначенного, в частности, для динамического создания и изменения состава меню.

За основу взято SDI-приложение, построенное с использованием мастера AppWizard, поэтому мы не будем приводить весь код, а ограничимся фрагментами, которые иллюстрируют соответствующие аспекты работы с меню в динамическом режиме. Сразу после запуска приложения в полосе меню определены названия стандартных раскрывающихся меню, доступные для выбора, одно меню заблокировано, его название изображается серым цветом (рис. 12.7). Для того чтобы открыть доступ к нему, необходимо в системном меню выбрать элемент **Разблокировать меню Цвет** (рис. 12.8), который при этом будет отмечен галочкой.

Рис. 12.7. Главное окно приложения DynMenu

Рис. 12.8. Системное меню приложения DynMenu

После этого можно выбирать любой цвет из первой строки списка цветов стандартного блока диалога **Color** (Цвет), который выводится на экран по команде **Добавить цвет**. Соответствующий цвет будет добавлен к новому, формируемому подменю команды **Список цветов** (рис. 12.9).

Рис. 12.9. Список выбранных и доступных цветов приложения DynMenu

Удалить цвета из этого списка можно двумя способами: выбрав элемент **Удалить все цвета** меню **Цвет** для очистки списка, или с помощью контекстного меню, появляющегося на экране по щелчку правой кнопкой мыши в окне приложения (рис. 12.10) — выбранный в нем цвет будет удален из списка.

Кроме этого, приложение демонстрирует пример создания собственных маркеров, используемых вместо стандартной галочки (рис. 12.11).

Теперь, после описания возможностей приложения, переходим непосредственно к фрагментам кода, разбив их на соответствующие темы. Фрагменты снабжены подробными комментариями и не требуют дополнительных пояснений.

Рис. 12.10. Контекстное меню приложения DynMenu

Рис. 12.11. Новый вид маркеров перед элементами меню в приложении DynMenu

Динамическое изменение меню

```
/*************************************************************
Файл MainFrm.h
Copyright (c) 1999 Мешков А., Тихомиров Ю.
*************************************************************/
class CMainFrame : public CFrameWnd
{
protected: // create from serialization only
  CMainFrame();
  DECLARE_DYNCREATE(CMainFrame)
// Attributes
```

```
public:
  ...
// Объект для создания динамического раскрывающегося меню
  CMenu   m_PopupMenu;
// Operations
public:
// функция, изменяющая состав динамического раскрывающегося меню
  void ChangeColorMenu();
...
// Generated message map functions
protected:
  //{{AFX_MSG(CMainFrame)
    ...
    // Обработчик команды "Удалить все цвета"
    afx_msg void OnDeleteAll();
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()
};
/************************************************************
Файл MainFrm.cpp
Copyright (c) 1999 Мешков А., Тихомиров Ю.
************************************************************/
IMPLEMENT_DYNCREATE(CMainFrame, CFrameWnd)
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
  //{{AFX_MSG_MAP(CMainFrame)
    ...
    ON_COMMAND(ID_DELETE_ALL, OnDeleteAll)
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()
void CMainFrame::ChangeColorMenu()
{
  // Получаем доступ к документу
  CDynMenuDoc* pDoc = (CDynMenuDoc*)GetActiveDocument();
  ASSERT_KINDOF(CDynMenuDoc, pDoc);
// Создаем два указателя на класс CMenu для доступа
// к меню верхнего уровня (pTopMenu) и к динамическому меню цветов
  CMenu* pColorMenu = NULL;
  CMenu* pTopMenu = AfxGetMainWnd()->GetMenu();
  int iPos;
```

```
  // Ищем элемент меню "Добавить цвет"
  // по идентификатору ID_ADD_COLOR
  for (iPos = pTopMenu->GetMenuItemCount()-1; iPos >= 0; iPos--)
  {
   // Получаем указатель на элемент меню верхнего уровня
    CMenu* pMenu = pTopMenu->GetSubMenu(iPos);
    if (pMenu && pMenu->GetMenuItemID(0) == ID_ADD_COLOR)
    {
     // Если элемент - "Цвет", то прекращаем поиск
      pColorMenu = pMenu;
      break;
    }
  }
// Поиск должен закончиться успешно, иначе - ошибка
  ASSERT(pColorMenu != NULL);
  // Обновляем состав подменю команды "Список цветов".
  // Сначала удаляем все элементы подменю...
  for (iPos = pColorMenu->GetMenuItemCount()-3; iPos > 0; iPos--)
    pColorMenu->DeleteMenu(iPos, MF_BYPOSITION);
  m_PopupMenu.m_hMenu = NULL;
  // Затем вставляем все выбранные цвета и полосу разделителя
  if(pDoc->m_icolor != -1)
  {
    // Создаем подменю
    m_PopupMenu.CreatePopupMenu();
    // и вставляем его между разделителем
    // и элементом "Удалить все цвета"
    pColorMenu->InsertMenu(ID_DELETE_ALL,
                           MF_STRING | MF_POPUP | MF_BYCOMMAND,
                           (UINT)m_PopupMenu.m_hMenu, "Список цветов");
  }
    // Вставляем элементы подменю
  for (int i = 0; i < NUM_COLOR; i++)
  {
    if (pDoc->m_aColor[i].m_btr == TRUE)
    {
      TCHAR strColor[64];
      sprintf(strColor, "Цвет %d", i);
```

```
      m_PopupMenu.AppendMenu(MF_STRING,
                              pDoc->m_aColor[i].m_nID, strColor);
    }
  }
  // Вставляем разделитель
  if(pDoc->m_icolor != -1)
  pColorMenu->InsertMenu(ID_DELETE_ALL,
                         MF_BYCOMMAND | MF_SEPARATOR);
}
void CMainFrame::OnDeleteAll()
{
  // Получаем доступ к документу
  CDynMenuDoc* pDoc = (CDynMenuDoc*)GetActiveDocument();
  ASSERT_KINDOF(CDynMenuDoc, pDoc);
  // Снимаем маркер элемента
  for (int i = 0; i < NUM_COLOR; i++)
  {
    pDoc->m_aColor[i].m_btr = FALSE;
  }

  // Определяем, что нет доступного текущего цвета
  pDoc->m_icolor = -1;

  // Обновляем содержимое меню
  ChangeColorMenu();
  // Обновляем изображение в окне документа
  pDoc->UpdateAllViews(NULL);
}
/*************************************************************
Файл DynMenuDoc.h
Copyright (c) 1999 Мешков А., Тихомиров Ю.
*************************************************************/
// Максимальное разрешенное число элементов подменю
#define NUM_COLOR 8
struct MenuColor
{
  int      m_pos; // Порядковый номер в меню
  UINT     m_nID; // Идентификатор меню
  COLORREF m_clr; // Цвет
```

```
  BOOL     m_btr; // Включать ли в список
};
class CDynMenuDoc : public CDocument
{
protected: // create from serialization only
  CDynMenuDoc();
  DECLARE_DYNCREATE(CDynMenuDoc)
// Attributes
public:
  BOOL bDelete;              // Флаг удаления или вставки элемента в меню
  int       m_icolor;        // Текущий цвет
  static MenuColor m_aColor[NUM_COLOR]; // Массив элементов меню
// Operations
public:
  void DoUpdateSelectColor(int iColor, CCmdUI*pCmdUI);
  void DoSelectColor(int iColor);
...
// Generated message map functions
protected:
  //{{AFX_MSG(CDynMenuDoc)
    // Обработчик команды ID_COLOR для добавления элемента в меню
    afx_msg void OnAddColor();
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()
};
/************************************************************
Файл DynMenuDoc.cpp
Copyright (c) 1999 Мешков А., Тихомиров Ю.
************************************************************/
// Инициализация массива цветов: исходно все недоступны
MenuColor CDynMenuDoc::m_aColor[NUM_COLOR] =
{
  {0, ID_COLOR0,  8421631, FALSE},
  {0, ID_COLOR1,  8454143, FALSE},
  {0, ID_COLOR2,  8454016, FALSE},
  {0, ID_COLOR3,  8453888, FALSE},
  {0, ID_COLOR4, 16777088, FALSE},
  {0, ID_COLOR5, 16744448, FALSE},
  {0, ID_COLOR6, 12615935, FALSE},
```

```
  {0, ID_COLOR7, 16744703, FALSE}
};
IMPLEMENT_DYNCREATE(CDynMenuDoc, CDocument)
BEGIN_MESSAGE_MAP(CDynMenuDoc, CDocument)
  //{{AFX_MSG_MAP(CDynMenuDoc)
    ON_COMMAND(ID_ADD_COLOR, OnAddColor)
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()
/////////////////////////////////////////////////////////////////////////////
// CDynMenuDoc construction/destruction
CDynMenuDoc::CDynMenuDoc()
{
  m_icolor = -1;
  bDelete  = FALSE;
}
void CDynMenuDoc::OnAddColor()
{
  // Настраиваем стандартный блок диалога Color (Цвет)
  CColorDialog dlg;
  dlg.m_cc.Flags |= CC_PREVENTFULLOPEN;
  if(dlg.DoModal() == IDOK)
  {
    // Получаем выбранный цвет
    COLORREF color = dlg.GetColor();

    // Если его еще нет в списке доступных цветов,
    // то добавляем его туда
    for(int i = 0; i < NUM_COLOR; i++)
    {
      if(color == m_aColor[i].m_clr)
      {
         m_aColor[i].m_btr = TRUE;
         m_icolor = i;
         break;
      }
    }

    // Получаем доступ к текущему представлению
    POSITION pos = GetFirstViewPosition();
```

```
    CView* pview = GetNextView(pos);
    ASSERT(pview != NULL);

    // Получаем доступ к активному фрейму
    CMainFrame* pwnd = (CMainFrame*)pview->GetParentFrame();
    ASSERT_KINDOF(CMainFrame, pwnd);

    // Обновляем состав доступных цветов
    pwnd->ChangeColorMenu();

    // Обновляем изображение
    UpdateAllViews(NULL);
  }
}
void CDynMenuDoc::DoSelectColor(int iColor)
{
  ASSERT(iColor >= 0 && iColor < NUM_COLOR);
  ASSERT(m_aColor[iColor].m_btr == TRUE);
  m_icolor = iColor;
  UpdateAllViews(NULL);
}
void CDynMenuDoc::DoUpdateSelectColor(int iColor, CCmdUI*pCmdUI)
{
  ASSERT(iColor >= 0 && iColor < NUM_COLOR);
  ASSERT(m_aColor[iColor].m_btr == TRUE);
  pCmdUI->SetCheck(m_icolor == iColor);
  pCmdUI->Enable(TRUE);
}

// Поскольку используемые команды не зарегистрированы,
// внедряемся в стандартный маршрут команд для перехвата
BOOL CDynMenuView::OnCmdMsg(UINT nID, int nCode, void* pExtra,
                            AFX_CMDHANDLERINFO* pHandlerInfo)
{
  CDynMenuDoc* pDoc = GetDocument();
  ASSERT_VALID(pDoc);
  // Если pHandlerInfo = NULL, обрабатываем сообщение
  if (pHandlerInfo == NULL)
  {
```

```
    // Организуем фильтр для перехвата нужной команды
    for (int i = NUM_COLOR-1; i >= 0; i--)
    {
      if (nID == pDoc->m_aColor[i].m_nID)
      {
        if (nCode == CN_COMMAND)
        {
          // Обрабатываем перехваченную команду
          // Отдельная обработка для удаления элемента и его вставки
          if(pDoc->bDelete)
          {
            pDoc->bDelete = FALSE;
            pDoc->m_aColor[i].m_btr = FALSE;
          }
          else
            pDoc->DoSelectColor(i);
          ...
        }
        else if (nCode == CN_UPDATE_COMMAND_UI)
        {
          // Получаем доступ к фрейму
          pDoc->DoUpdateSelectColor(i, (CCmdUI*)pExtra);
        }
        return TRUE;
      }
    }
  }
  // Не забываем вернуть управление в стандартный маршрут команд
  return CView::OnCmdMsg(nID, nCode, pExtra, pHandlerInfo);
}
```

Системное меню

```
/*********************************************************
Файл MainFrm.h
Copyright (c) 1999 Мешков А., Тихомиров Ю.
*********************************************************/
class CMainFrame : public CFrameWnd
{
```

```
protected: // create from serialization only
  CMainFrame();
  DECLARE_DYNCREATE(CMainFrame)
  ...
  // ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CMainFrame)
  public:
    ...
    // Переопределяем обработчик системных команд
    afx_msg void OnSysCommand( UINT nID, LPARAM lParam);
  //}}AFX_VIRTUAL
  DECLARE_MESSAGE_MAP()
};
/************************************************************
Файл MainFrm.cpp
Copyright (c) 1999 Мешков А., Тихомиров Ю.
************************************************************/
IMPLEMENT_DYNCREATE(CMainFrame, CFrameWnd)
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
  //{{AFX_MSG_MAP(CMainFrame)
    ON_WM_SYSCOMMAND()
    ...
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()
void CMainFrame::ActivateFrame(int nCmdShow)
{
...
  CMenu *pViewMenu = NULL;
  CMenu *pTopMenu = AfxGetMainWnd()->GetMenu();
  int iPos;
// Добавляем элемент "Разблокировать меню Цвет" в системное меню.
// IDM_COLOR должен находиться в границах системных команд
  ASSERT((IDM_COLOR & 0xFFF0) == IDM_COLOR);
  ASSERT(IDM_COLOR < 0xF000);
  // Получаем доступ к системному меню
  CMenu* pSysMenu = GetSystemMenu(FALSE);
  CString strColorMenu("Разблокировать меню Цвет");
  // Добавляем в него разделитель и новый элемент
  // "Разблокировать меню Цвет"
```

```
  pSysMenu->AppendMenu(MF_SEPARATOR);
  pSysMenu->AppendMenu(MF_BYCOMMAND | MF_STRING,
                       IDM_COLOR, strColorMenu);
  // Снимаем метку, показывая, что элемент "Цвет"
  // меню приложения заблокирован
  pSysMenu->CheckMenuItem(IDM_COLOR, MF_BYCOMMAND | MF_UNCHECKED);
  // Ищем элемент меню Add Color по идентификатору ID_ADD_COLOR
  for (iPos = pTopMenu->GetMenuItemCount()-1; iPos >= 0; iPos--)
  {
    CMenu* pMenu = pTopMenu->GetSubMenu(iPos);
    if (pMenu && pMenu->GetMenuItemID(0) == ID_ADD_COLOR)
    {
      break;
    }
  }
  // Разблокируем элемент "Цвет" в полосе меню
  pTopMenu->EnableMenuItem(iPos,
                           MF_BYPOSITION | MF_GRAYED);
  // Перерисовываем полосу меню
  DrawMenuBar();
  // Передаем управление фрейму
  CFrameWnd::ActivateFrame(nCmdShow);
}
// Поскольку команда не зарегистрирована,
// необходимо переопределить обработчик
void CMainFrame::OnSysCommand( UINT nID, LPARAM lParam)
{
  if ((nID & 0xFFF0) == IDM_COLOR)
  {
    CMenu* pColorMenu = NULL;
    CMenu* pTopMenu = AfxGetMainWnd()->GetMenu();
    int iPos;
    // Ищем элемент меню "Добавить цвет" по идентификатору
    // ID_ADD_COLOR
    for (iPos = pTopMenu->GetMenuItemCount()-1; iPos >= 0; iPos--)
    {
      CMenu* pMenu = pTopMenu->GetSubMenu(iPos);
      if (pMenu && pMenu->GetMenuItemID(0) == ID_ADD_COLOR)
      {
        break;
```

```
      }
    }
    // Получаем доступ к системному меню
    CMenu *pSysMenu = GetSystemMenu(FALSE);
    // Определяем текущее состояние элемента
    UINT nState = pSysMenu->GetMenuState(IDM_COLOR, MF_BYCOMMAND);
    // Если элемент отмечен, то убираем маркер и запрещаем
    // доступ к элементу "Цвет" полосы меню, и наоборот
    if(nState == MF_CHECKED)
    {
      pSysMenu->CheckMenuItem(IDM_COLOR,
                              MF_BYCOMMAND | MF_UNCHECKED);
      pTopMenu->EnableMenuItem(iPos,
                               MF_BYPOSITION | MF_GRAYED);
    }
    else
    {
      pSysMenu->CheckMenuItem(IDM_COLOR,
        MF_BYCOMMAND | MF_CHECKED);
      pTopMenu->EnableMenuItem(iPos,
        MF_BYPOSITION | MF_ENABLED);
    }
    // Перерисовываем полосу меню
    DrawMenuBar();
  }
  else
  {
    // Если команда не наша, необходимо
    // передать ее для корректной обработки
    CWnd::OnSysCommand(nID, lParam);
  }
}
```

Контекстное меню

```
/*********************************************************
Файл DynMenuView.h
Copyright (c) 1999 Мешков А., Тихомиров Ю.
*********************************************************/
```

```
class CDynMenuView : public CView
{
protected: // create from serialization only
  CDynMenuView();
  DECLARE_DYNCREATE(CDynMenuView)
// Attributes
public:
  CDynMenuDoc* GetDocument();
// Overrides
// ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CDynMenuView)
  public:
    ...
    // Функция для внедрения в стандартный маршрут команд
    // добавления и удаления элементов меню
    virtual BOOL OnCmdMsg(UINT nID, int nCode, void* pExtra,
                          AFX_CMDHANDLERINFO* pHandlerInfo);
  //}}AFX_VIRTUAL
// Generated message map functions
  protected:
  //{{AFX_MSG(CDynMenuView)
    // Обработчик для создания контекстного меню
    afx_msg void OnRButtonDown(UINT nFlags, CPoint point);
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()
};
/*********************************************************
Файл DynMenuView.cpp
Copyright (c) 1999 Мешков А., Тихомиров Ю.
*********************************************************/
IMPLEMENT_DYNCREATE(CDynMenuView, CView)
BEGIN_MESSAGE_MAP(CDynMenuView, CView)
  //{{AFX_MSG_MAP(CDynMenuView)
    ON_WM_RBUTTONDOWN()
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()
void CDynMenuView::OnRButtonDown(UINT nFlags, CPoint point)
{
  CDynMenuDoc* pDoc = GetDocument();
```

```
  ASSERT_VALID(pDoc);
  CMenu Menu;
  // Для вычисления расположения контекстного меню на экране, нужно
  // преобразовать положение указателя мыши в экранные координаты
  ClientToScreen(&point);
  // Создаем раскрывающееся меню
  Menu.CreatePopupMenu();
  for (int i = 0; i < NUM_COLOR; i++)
  {
    if (pDoc->m_aColor[i].m_btr == TRUE)
    {
      TCHAR strColor[64];
      sprintf(strColor, "Цвет %d", i);
      Menu.AppendMenu(MF_STRING  | MF_OWNERDRAW,
                      pDoc->m_aColor[i].m_nID, strColor);[1]
    }
  }
  // При помощи этого меню будем удалять элементы
  pDoc->bDelete = TRUE;
  // Выводим контекстное меню на экран
  Menu.TrackPopupMenu(TPM_LEFTALIGN | TPM_RIGHTBUTTON,
                      point.x, point.y, this);
  // Как всегда, не мешаем нормальной работе
  CView::OnRButtonDown(nFlags, point);
}
BOOL CDynMenuView::OnCmdMsg(UINT nID, int nCode, void* pExtra,
                            AFX_CMDHANDLERINFO* pHandlerInfo)
{
  CDynMenuDoc* pDoc = GetDocument();
  ASSERT_VALID(pDoc);
  if (pHandlerInfo == NULL)
  {
    // Организуем фильтр для перехвата нужной команды
    for (int i = NUM_COLOR-1; i >= 0; i--)
    {
      if (nID == pDoc->m_aColor[i].m_nID)
      {
```

[1] Для режима самоотображения необходимо изменить эту функцию.

```
        if (nCode == CN_COMMAND)
        {
          // Обрабатываем перехваченную команду.
          // Отдельная обработка для удаления элемента и его вставки
          if(pDoc->bDelete)
          {
            pDoc->bDelete = FALSE;
            pDoc->m_aColor[i].m_btr = FALSE;
          }
          else
            pDoc->DoSelectColor(i);
          // Получаем доступ к фрейму
          CMainFrame* pwnd = (CMainFrame*)GetParentFrame();
          ASSERT_KINDOF(CMainFrame, pwnd);
          // Обновляем состав раскрывающегося меню
          pwnd->ChangeColorMenu();
          pDoc->UpdateAllViews(NULL);
        }
        else if (nCode == CN_UPDATE_COMMAND_UI)
        {
          // Обновляем состояние элементов меню
          pDoc->DoUpdateSelectColor(i, (CCmdUI*)pExtra);
        }
        return TRUE;
      }
    }
  }
  // Не забываем вернуть управление в стандартный маршрут команд
  return CView::OnCmdMsg(nID, nCode, pExtra, pHandlerInfo);
}
```

Создание собственных маркеров состояния

```
/*********************************************************
Файл MainFrm.h
Copyright (c) 1999 Мешков А., Тихомиров Ю.
*********************************************************/
class CMainFrame : public CFrameWnd
{
```

```
...
// Attributes
public:
  CBitmap m_bmpCheck;     // Для отмеченного состояния
  CBitmap m_bmpUnCheck;   // Для неотмеченного состояния
  // ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CMainFrame)
  public:
    ...
    // Изменять вид маркера будем здесь
    virtual void ActivateFrame(int nCmdShow = -1);
  //}}AFX_VIRTUAL
...
  DECLARE_MESSAGE_MAP()
};
/*********************************************************
Файл MainFrm.cpp
Copyright (c) 1999 Мешков А., Тихомиров Ю.
*********************************************************/
void CMainFrame::ActivateFrame(int nCmdShow)
{
 // Загружаем битовые массивы
  m_bmpCheck.LoadBitmap(IDB_MENU_DOWN);
  m_bmpUnCheck.LoadBitmap(IDB_MENU_UP);
  // Получаем системные размеры маркера
  int cx = ::GetSystemMetrics(SM_CXMENUCHECK);
  int cy = ::GetSystemMetrics(SM_CYMENUCHECK);
  // Приводим текущие размеры битовых массивов к системным
  m_bmpUnCheck.SetBitmapDimension(cx, cy);
  m_bmpCheck.SetBitmapDimension(cx, cy);
  CMenu *pViewMenu = NULL;
  CMenu *pTopMenu = AfxGetMainWnd()->GetMenu();
  int iPos;
  // Ищем элемент меню "Добавить цвет" по идентификатору ID_ADD_COLOR
  for (iPos = pTopMenu->GetMenuItemCount()-1; iPos >= 0; iPos--)
  {
    CMenu* pMenu = pTopMenu->GetSubMenu(iPos);
    if (pMenu && pMenu->GetMenuItemID(0) == ID_VIEW_TOOLBAR)
    {
```

```
      pViewMenu = pMenu;
      break;
    }
  }
  ASSERT(pViewMenu != NULL);
  // Устанавливаем новые маркеры для отмеченного
  // и неотмеченного состояний
  pViewMenu->SetMenuItemBitmaps(ID_VIEW_TOOLBAR, MF_BYCOMMAND,
                                &m_bmpUnCheck, &m_bmpCheck);
  pViewMenu->SetMenuItemBitmaps(ID_VIEW_STATUS_BAR, MF_BYCOMMAND,
                                &m_bmpUnCheck, &m_bmpCheck);
...
  // Передаем управление фрейму
  CFrameWnd::ActivateFrame(nCmdShow);
}
```

Самоотображение элементов меню

```
/*********************************************************
Файл DynMenuView.h
Copyright (c) 1999 Мешков А., Тихомиров Ю.
*********************************************************/
// Для реализации режима самоотображения элемента меню необходимо
// создать свой класс на базе CMenu
class CColorMenu : public CMenu
{
public:
  CColorMenu(){}
// Implementation
  // Переопределяем необходимые виртуальные функции
  virtual void MeasureItem(LPMEASUREITEMSTRUCT lpMIS);
  virtual void DrawItem(LPDRAWITEMSTRUCT lpDIS);
};
/*********************************************************
Файл DynMenuView.cpp
Copyright (c) 1997 Мешков А., Тихомиров Ю., Шведов Д.
*********************************************************/
// Размеры для элемента меню в режиме самоотображения
#define COLOR_BOX_WIDTH     30
```

```
#define COLOR_BOX_HEIGHT    20
void CColorMenu::MeasureItem(LPMEASUREITEMSTRUCT lpMIS)
{
  // Все элементы имеют один размер, но
  // при желании можно сделать и разные
  lpMIS->itemWidth = COLOR_BOX_WIDTH;
  lpMIS->itemHeight = COLOR_BOX_HEIGHT;
}
void CColorMenu::DrawItem(LPDRAWITEMSTRUCT lpDIS)
{
  // Для любого графического вывода нужен контекст устройства
  CDC* pDC = CDC::FromHandle(lpDIS->hDC);
  COLORREF cr = (COLORREF)lpDIS->itemData; // Цвет в элементе данных
  if (lpDIS->itemAction & ODA_DRAWENTIRE)
  {
    // Рисуем элемент меню
    CBrush br(cr);
    pDC->FillRect(&lpDIS->rcItem, &br);
  }
  if ((lpDIS->itemState & ODS_SELECTED) &&
      (lpDIS->itemAction & (ODA_SELECT | ODA_DRAWENTIRE)))
  {
    // Текущий элемент выделяем рамкой
    COLORREF crHilite = RGB(255-GetRValue(cr),
                            255-GetGValue(cr), 255-GetBValue(cr));
    CBrush br(crHilite);
    pDC->FrameRect(&lpDIS->rcItem, &br);
  }
  if (!(lpDIS->itemState & ODS_SELECTED) &&
       (lpDIS->itemAction & ODA_SELECT))
  {
    // Невыделенный элемент - без рамки
    CBrush br(cr);
    pDC->FrameRect(&lpDIS->rcItem, &br);
  }
}
```

Полный текст рассмотренного приложения приведен на сопроводительной дискете.

Глава 13

Акселераторы, графические и другие ресурсы

Акселераторы

Для ускорения доступа к элементам меню при помощи клавиатуры, а также назначения тех или иных функций, не связанных с меню, комбинациям клавиш, в Windows используется так называемая *таблица акселераторов* (accelerator table). Она находится в ресурсах приложения и определяет соответствие между *акселератором* — комбинацией клавиш — и значением параметра WM_COMMAND, передающимся в функцию окна при ее нажатии.

Для того чтобы некоторая комбинация клавиш стала работать как акселератор, она должна быть описана в таблице акселераторов и, кроме того, приложение должно загрузить таблицу акселераторов из ресурсов. Таблицу акселераторов можно создавать и изменять двумя путями — в файле ресурсов и с использованием некоторого визуального редактора ресурсов, типа AppStudio.

Таблица акселераторов определяется в файле описания ресурсов приложения в следующем виде:

```
<ID> ACCELERATORS
BEGIN
  Key, ID, [Type], [Modifiers]
  ...
END
```

Идентификатор ID используется для ссылки на таблицу акселераторов, а BEGIN и END определяют операторные скобки, внутри которых располагаются строки описания акселераторов, имеющие следующий формат:

```
Key, ID, [Type], [Modifiers]
```

Key — определяет клавишу, которая будет использована для создания акселератора (можно использовать символ в коде ASCII, заключенный в двойные кавычки, например, "C", комбинацию символа ASCII со знаком "^", например, "^C", что соответствует комбинации клавиш <Ctrl>+<C>, код ASCII клавиши в виде целого значения или виртуальный код клавиши в символьном или цифровом виде).

ID — задает идентификатор команды WM_COMMAND, которая направляется в функцию окна при использовании акселератора.

Type — принимает одно из двух значений: ASCII или VirtKey и определяет клавишу с использованием кода ASCII или с использованием кода виртуальной клавиши.

Modifiers — поле доступно, если в поле Type выбрано значение VirtKey; в этом случае можно указывать параметры Ctrl, Alt или Shift.

На рис. 13.1 изображено окно свойств для формирования таблицы акселераторов с использованием визуального средства построения ресурсов.

Рис. 13.1. Окно свойств, открытое в режиме настройки акселератора

После того как таблица определена, осталось совсем немного — приложение должно в определенном месте загрузить ее из файла ресурсов. Сделать это можно с помощью функции:

BOOL CFrameWnd::**LoadAccelTable**(LPCTSTR lpszResourseName) —

параметр lpszResourceName определяет имя таблицы акселераторов в файле ресурсов. Следует иметь в виду, что одновременно может быть загружена только одна таблица. При использовании числового идентификатора это имя можно получить с помощью макроса MAKEINTRESOURCE(lpszResourceName).

В случае, если создание окна производилось при помощи функции LoadFrame, таблица акселераторов загружается автоматически, одновременно с ресурсами меню и значками, которые должны иметь одинаковые идентификаторы. Ее уничтожение также производится библиотекой MFC.

Как видите, все очень просто. Единственное, что хотелось бы еще раз повторить: акселераторы представляют собой самостоятельную часть Windows и позволяют не только дублировать команды меню, но и вызывать те команды, которые в нем отсутствуют.

Графические ресурсы

К графическим ресурсам Windows относятся *курсоры*, *пиктограммы*, *битовые массивы* и *панели инструментов*. Все они кодируются практически одинаково (различаясь только техникой использования) и являются частным случаем кодировки битовых массивов формата BMP. Первые три ресурса уже были

достаточно подробно рассмотрены в предыдущих главах, поэтому здесь скажем несколько слов только о панелях инструментов. Они представляют собой набор обычных битовых массивов, каждому из которых соответствует определенный идентификатор команды.

На рис. 13.2 показан внешний вид окна свойств для настройки параметров кнопки панели инструментов, определяемых в полях:

- **ID** определяет идентификатор команды, ассоциированной с данной кнопкой
- **Width** и **Height** — задают размеры кнопки (ширина и высота)
- **Prompt** — содержит две строки, разделенные символами "\n" и изображаемые, соответственно, в строке состояния и во всплывающей подсказке

Рис. 13.2. Окно свойств, открытое для настройки кнопки панели инструментов

Подключение данного ресурса к фрейму осуществляется очень просто — в классе фрейма определяется переменная класса CToolBar для создания объекта:

```
class CMainFrame : public CFrameWnd
{
...
protected:  // control bar embedded members
  CToolBar    m_wndToolBar;
...
};
```

а в обработчике сообщения Windows WM_CREATE вызываются функции для создания окна Windows и загрузки панели инструментов из ресурсов:

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
  ...
  if (!m_wndToolBar.Create(this) ||              // Создание окна
      !m_wndToolBar.LoadToolBar(IDR_MAINFRAME)) // Загрузка ресурса
  {
```

```
    ...
    // Обработка ошибки создания или загрузки панели инструментов
    return -1;
  }
  ...
}
```

Как видите, все точно так же, как и при "старом" способе создания панели инструментов. Отличие в том, что теперь в тексте программы не надо привязывать кнопки к соответствующим идентификаторам — это делается в соответствующем ресурсе.

Строковые ресурсы

Это один из простейших ресурсов Windows. Как обычно, удобнее всего строить его с помощью редактора ресурсов, где нужно задать идентификатор строки и собственно ее текст. При этом в файл ресурсов добавляются строки вида:

```
STRINGTABLE DISCARDABLE
BEGIN
  ID_ADD_COLOR            "Добавить новый цвет\nЦвет"
  ID_DELETE_ALL           "Удалить все цвета\nУдалить цвета"
END
```

имеющие два параметра: идентификатор и собственно строка.

При сопоставлении строкового ресурса с некоторой командой она формируется автоматически при записи текста в поле **Prompt** соответствующего окна свойств.

Загрузка этого вида ресурса также очень проста, и заключается в использовании функции:

BOOL CString::**LoadString**(UINT nID) —

в качестве параметра используется идентификатор строкового ресурса.

При работе со строковыми ресурсами рекомендуем придерживаться следующих правил для экономии памяти:

- Группировать строки по принципу совместного использования (например, строки, используемые в одном модуле приложения, имеют идентификаторы 100—107, а в другом модуле — 210—217)
- Загружать строки в автоматические локальные переменные, чтобы используемое ими место автоматически освобождалось по окончании их использования
- Использовать одну и ту же переменную в качестве буфера хранения нескольких строк

Ресурсы, определяемые программистом

Этот вид ресурсов используется для включения в исполняемый файл специальных данных, которые могут иметь произвольный формат. Для подключения таких ресурсов к приложению нужно описать их в файле ресурсов, например, так

```
UserResource USERDATA UserResource.dat
```

В процессе компоновки имя UserResource будет зарегистрировано как ресурс типа USERDATA в таблице ресурсов исполняемого файла. При этом для обозначения типа и имени ресурса могут использоваться числа (больше 255).

В библиотеке MFC отсутствует дополнительная поддержка этого типа ресурсов, поэтому для их использования следует обратиться к Win32 API, где определены следующие функции: LoadResource, LockResource, FindResource, FindResourceEx, FreeResource, LoadLibrary и LoadModule.

При работе с этими функциями есть два тонких момента. Первый заключается в том, что для загрузки ресурса необходимо вызвать функцию LoadResource, которая, однако, не загружает его в память, и работать с ним пока еще нельзя. Собственно загрузка осуществляется при закреплении блока данных в памяти функцией LockResource. А второй связан с тем, что в Windows NT ресурс освобождается операционной системой, а в Windows 95/98 это должен сделать сам программист, явно вызывая функцию FreeResource.

БЛОКИ ДИАЛОГА

Глава 14

Модальные и немодальные блоки диалога

Блоки диалога (dialog boxes) — специальные окна, поддерживающие гибкие средства, используя которые пользователь может взаимодействовать с программой. Как правило, они включают *дочерние* (child) окна, которые в контексте этого типа окон называются *элементами управления*.

Примечание

Как вы прекрасно понимаете, сказанное отнюдь не означает, что элементы управления можно использовать только в блоках диалога.

Блоки диалога бывают двух типов — *модальные* (modal) и *немодальные* (modeless). Разница между ними заключается в способе управления потоками сообщений. Модальные блоки диалога блокируют все остальные окна приложения так, что пользователь не может с ними ничего сделать, пока не закроет его. Другими словами, они отсекают поток сообщений, идущих от мыши и клавиатуры к его родительским окнам или окнам того же уровня (sibling), делая их недоступными. Если пользователь все же пытается взаимодействовать с недоступным окном, то система предупреждает об этом звуковым сигналом. Блоки диалога этого типа обычно не препятствуют переключению с одного приложения на другое. В редких случаях можно использовать системные модальные блоки диалога, которые позволяют пользователю работать внутри лишь одного блока диалога и не позволяют переключаться ни на какие другие приложения.

Немодальные блоки диалога больше похожи на обычные окна, т. к. они дают пользователю возможность доступа к остальным окнам приложения. Это связано с тем, что блоки диалога этого типа не прерывают потоков сообщений, идущих в любую часть вашей программы.

Оба типа блоков диалога являются важнейшей частью пользовательского интерфейса Windows, и уже давно сложился стандартный набор элементов управления, широко используемых в блоках диалога.

Примечание

Наряду с уже привычными элементами управления Windows: кнопками (PUSHBUTTON), флажками (CHECK BOX), переключателями (RADIO BUTTON), списками (LIST BOX), текстовыми полями (EDIT), комбинированными списками (COMBO BOX), полосами прокрутки (SCROLLBAR) и статическими элементами (STATIC) в Win32 есть достаточно большое количество более новых, которые подробно будут рассмотрены в следующих главах.

На первый взгляд немодальные блоки диалога более привлекательны, поскольку дают пользователю свободу выбора способа действия. Однако большинство блоков диалога все-таки модальные. Это связано с тем, что модальный блок диалога идеально концентрирует внимание пользователя на некотором ограниченном наборе действий, которые он может в данный момент выполнить.

Создание и использование блока диалога, модального и немодального, требует одновременного наличия трех компонентов:

- Шаблона блока диалога
- Программного кода, создающего и отображающего его на экране
- Диалоговой процедуры, которая обслуживает взаимодействие пользователя с блоком диалога

Начнем с шаблона, т. к. без него не будет и самого блока диалога.

Шаблон блока диалога представляет собой объект данных, определенный в файле ресурсов (*.rc), аналогичный приведенному ниже:

```
// Блок диалога About
IDD_ABOUTBOX DIALOG DISCARDABLE  0, 0, 217, 55
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "О программе MDI"
FONT 8, "MS Sans Serif"
BEGIN
  ICON  IDR_MAINFRAME,IDC_STATIC,7,7,21,20
  CTEXT "MDI-приложение Версия 1.0",
    IDC_STATIC,51,17,119,8, SS_NOPREFIX
  CTEXT  "Copyright © 1999",IDC_STATIC,127,34,71,8
  DEFPUSHBUTTON   "OK",IDOK,178,7,32,14,WS_GROUP
  LTEXT  "Мешков А., Тихомиров Ю.",IDC_STATIC,35,34,89,8
END
```

Один из способов создания шаблона блока диалога — посредством редактирования файла ресурсов. Другой способ заключается в использовании какого-либо визуального средства, такого как App Studio или ему подобного, которое позволяет изобразить предварительный набросок блока диалога (рис. 14.1).

Рис. 14.1. Эскиз блока диалога

Но какой бы способ вы ни выбрали, помните, что блок диалога это всего-навсего еще один ресурс. Внимательно посмотрев на приведенный текст, вы заметите, что первая строка содержит идентификатор ресурса блока диалога (IDD_ABOUTBOX), за которым следует объявление типа ресурса (DIALOG) и размеров блока диалога. Следующая строка определяет его стиль, объединяя четыре константы, в основном, уже знакомые вам. После того как объявлен стиль, можно указать заголовок блока диалога (CAPTION), который появляется в его полосе заголовка. Если не определить этот оператор, то, естественно, полоса останется пустой. Следующая строчка описания ресурса (FONT) определяет размер и имя шрифта, который будет использоваться в этом блоке диалога.

Примечание

Если при определении блока диалога вы не используете стиль WS_CAPTION, то полосы заголовка не будет.

Описание блока диалога заканчивается несколькими строчками, заключенными между BEGIN и END. В данном случае они определяют значок, две строки текста и кнопку. Как мы говорили, все они являются элементами управления, т. е. окнами. Другими словами, наш блок диалога имеет четыре дочерних окна, размещенных внутри него: кнопку, два статических элемента управления и значок.

Строчка, описывающая кнопку, начинается с оператора, определяющего тип элемента управления (DEFPUSHBUTTON) и имеет 7 параметров: текст, ассоциируемый с элементом управления, идентификатор, сектор размещения (координаты левого верхнего угла, ширина и высота) и стиль.

Поскольку и значок (ICON) и статические элементы (LTEXT и CTEXT) являются окнами, неудивительно, что они имеют те же параметры, что и кнопка. Некоторые стили элементов управления указываются непосредственно в названии. Например, DEFPUSHBUTTON говорит о том, что у кнопки установлен стиль BS_DEFPUSHBUTTON, а LTEXT — о том, что элемент управления имеет установленный флаг стиля ES_LEFT. Вместо конкретных операторов можно указывать также оператор CONTROL, который является общим для различных элементов управления и имеет 8 параметров:

```
CONTROL <текст>, nID, <класс>, x, y, cx, cy, <стили>
```

где к уже перечисленным параметрам добавляется символьная строка, определяющая имя оконного класса, с помощью которого должен быть создан элемент управления блока диалога.

Итак, формирование шаблона блока диалога включает создание объекта ресурсов, который описывает форму, размер и расположение на экране блока диалога и каждого из его внутренних элементов управления. Данный объект помещается в файл ресурсов приложения вместе с другими объектами (значками, меню, битовыми образами и т. д.). После того как шаблон сформирован, можно приступать к созданию программного кода, способного вызывать появление блока диалога на экране и его последующее удаление.

Как обычно, библиотека MFC предоставляет специальный класс для создания и работы с блоками диалогов — CDialog. Следуя принятому нами стилю изложения, рассмотрим этот класс максимально подробно.

Класс CDialog

Основное назначение этого класса состоит в предоставлении интерфейса программирования для управления блоками диалога. Использование класса CDialog позволяет создавать объекты обоих типов блоков диалога — модальные и немодальные. Повторим еще раз: блок диалога подобен любому другому окну Windows, но в отличие от других типов окон, объект этого класса является комбинацией класса CDialog или производного от него и шаблона диалога. Два способа создания шаблона мы уже рассмотрели, а к третьему — созданию шаблона в памяти, вернемся несколько позже. Теперь же познакомимся непосредственно с самим классом CDialog.

Для создания объекта модального блока диалога на основе шаблона, хранящегося в ресурсах, реализованы два варианта конструктора:

```
CDialog::CDialog(
    LPCTSTR lpszTemplateName,
    CWnd *pParentWnd = NULL)
```

и

```
CDialog::CDialog(
    UINT nIDTemplate,
    CWnd *pParentWnd = NULL) —
```

первые параметры определяют шаблон: lpszTemplateName — указатель на символьную строку, содержащую имя шаблона, а nIDTemplate — номер идентификатора ресурса; pParentWnd указывает на объект родителя или владельца, которому принадлежит объект блока диалога (если значение этого параметра NULL, то родительским окном является основное окно приложения).

Следующим шагом является создание собственно блока диалога Windows. Для этого в классе реализована специальная функция

virtual int CDialog::**DoModal**( ) —

создает и активизирует модальный блок диалога Windows, осуществляет полное взаимодействие с пользователем и удаляет блок диалога после завершения работы, возвращая целое значение типа IDOK или IDCANCEL, позволяющее определить конечный результат его работы. Если функция не смогла создать блок диалога, то она возвращает –1.

Помимо использования шаблона, хранящегося в ресурсе, в системе предусмотрена возможность создания шаблона в памяти непосредственно во время выполнения программы. Это необходимо при косвенном создании блока диалога, как модального, так и немодального. Первым шагом на этом пути является определение и заполнение структуры DLGTEMPLATE, которая позволяет задать размеры и стиль блока диалога. Познакомимся с ней поближе:

```
typedef struct {
  DWORD style;
  DWORD dwExtendedStyle;
  WORD cdit;
  short x;
  short y;
  short cx;
  short cy;
} DLGTEMPLATE;
```

Поле style определяет стиль блока диалога. Наряду с уже известными стилями, общими для всех окон (такими как WS_CHILD и WS_CAPTION), для блоков диалога применяются также дополнительные:

DS_3DLOOK	Вывод трехмерной рамки вокруг элементов управления в блоке диалога; этот стиль применим только для Windows 95/98
DS_ABSALIGN	Координаты блока диалога задаются относительно экрана; если этот стиль не определен, то они задаются относительно рабочей области
DS_CENTER	Блок диалога располагается по центру рабочей области; этот стиль применим только для Windows 95/98
DS_CENTERMOUSE	Курсор мыши располагается в центре блока диалога; этот стиль применим только для Windows 95/98
DS_CONTEXTHELP	В полосу заголовка блока диалога добавляется знак вопроса, который подключает контекстно-зависимую справку; на самом деле, после создания блока диалога система проверяет наличие этого стиля, и если он присутствует, то добавляет флаг WS_EX_CONTEXTHELP к расширенному стилю блока диалога
DS_CONTROL	Блок диалога может использоваться как элемент управления в другом блоке диалога; наличие этого стиля позволяет пользователю перемещаться по элементам управления дочернего блока диалога, использовать акселераторы клавиатуры и т. д.

DS_FIXEDSYS	Блок диалога использует SYSTEM_FIXED_FONT вместо SYSTEM_FONT; этот стиль применим только для Windows 95/98
DS_MODALFRAME	Модальный блок диалога; этот стиль часто комбинируют с WS_CAPTION и WS_SYSMENU
DS_NOFAILCREATE	Создает блок диалога, даже в том случае, если произошли ошибки; например, если не может быть создан какой-либо элемент управления
DS_NOIDLEMSG	Блокирует сообщение WM_ENTERIDLE, которое Windows посылает владельцу блока диалога, пока он отображается на экране
DS_SETFONT	Показывает, что шаблон блока диалога (структура DLGTEMPLATE) содержит дополнительный элемент, определяющий имя и размер шрифта; этот шрифт используется для текста в рабочей области блока диалога и для текста внутри элементов управления; при этом сообщение Windows WM_SETFONT посылается в блок диалога и в каждый элемент управления
DS_SETFOREGROUND	Переводит блок диалога на передний план, для чего Windows вызывает функцию SetForegroundWindow
DS_SYSMODAL	Создает системный модальный блок диалога; при этом у него обязательно должен быть установлен расширенный стиль WS_EX_TOPMOST, иначе стиль будет проигнорирован

Поле dwExtendedStyle служит для установки расширенных стилей окна. Для создания блоков диалога это поле не используется, за исключением вышеупомянутых случаев. Оно используется, когда вы создаете другие типы окон, используя шаблон блока диалога.

Поле cdit задает число элементов блока диалога, т. е. число структур DLGITEMTEMPLATE, которые описывают элементы управления блока диалога:

```
typedef struct {
  DWORD style;
  DWORD dwExtendedStyle;
  short x;
  short y;
  short cx;
  short cy;
  WORD id;
  } DLGITEMTEMPLATE;
```

Все поля этой структуры аналогичны только что описанным, кроме поля id, которое определяет идентификатор элемента управления.

При ее использовании необходимо дополнительно описать еще три массива, определяющие класс, заголовок и данные элемента управления. Каждый массив состоит из одного или более 16-битных элементов:

- Первым за структурой следует *массив класса*. Если его первый элемент равен 0xFFFF, то массив имеет один дополнительный элемент, который определяет значение, идентифицирующее зарегистрированный оконный класс. Любое другое значение первого элемента воспринимается в Windows как строка Unicode, заканчивающаяся нулевым символом и определяющая имя зарегистрированного класса окна.
- После массива класса расположен *массив заголовка*. Если его первый элемент равен 0xFFFF, то массив имеет один дополнительный элемент, который определяет идентификатор ресурса для этого элемента управления. Любое другое значение первого элемента воспринимается в Windows как строка Unicode, заканчивающаяся нулевым символом и определяющая начальный текст элемента управления.
- *Массив данных* может содержать произвольное число элементов, которые оконная процедура элемента управления должна "понимать" и уметь обрабатывать. Эти данные передаются элементу управления, когда Windows вызывает функцию CreateWindowEx.

Поля x, y, cx и cy определяют соответственно координаты левого верхнего угла блока диалога, его ширину и высоту. Все размеры задаются в единицах блока диалога, которые можно преобразовать в значения экранных координат при помощи функции MapDialogRect.

При использовании структуры DLGTEMPLATE необходимо дополнительно описать еще три массива переменной длины, которые определяют меню, класс и заголовок блока диалога. Каждый массив содержит один или более 16-битных элементов:

- Первым располагается *массив меню*. Если его первый элемент равен 0x0000, то массив не имеет дополнительных элементов, а блок диалога не имеет меню. Если первый элемент равен 0xFFFF, то массив имеет один дополнительный элемент, который определяет идентификатор ресурса для меню. Любое другое значение первого элемента воспринимается в Windows как строка Unicode, заканчивающаяся нулевым символом и определяющая имя ресурса меню в исполняемом файле.
- За массивом меню следует *массив класса*. Если его первый элемент равен 0x0000, то Windows использует предопределенный оконный класс для блока диалога. Если первый элемент равен 0xFFFF, то массив имеет один дополнительный элемент, который определяет значение атома, идентифицирующее зарегистрированный класс окна блока диалога. Любое другое значение первого элемента воспринимается в Windows как строка Unicode, заканчивающаяся нулевым символом и определяющая имя зарегистрированного оконного класса.

- После массива класса расположен *массив заголовка*. Если его первый элемент равен 0x0000, то блок диалога не имеет заголовка. Любое другое значение первого элемента воспринимается в Windows как строка Unicode, заканчивающаяся нулевым символом и определяющая текст заголовка.

Если для блока диалога установлен стиль DS_SETFONT, то за массивом заголовка располагается 16-битный элемент, определяющий размер шрифта, за которым следует массив, задающий его вид. Массив вида шрифта — строка Unicode, заканчивающаяся нулевым символом и определяющая имя вида. Если эти параметры заданы, то Windows посылает сообщения WM_SETFONT оконным процедурам блока диалога и элементов управления.

После создания и заполнения необходимых структур и массивов можно создавать окно блока диалога. Как обычно, сначала нужно создать объект класса, используя для этого пустой конструктор, а затем вызвать функцию:

BOOL CDialog::**InitModalIndirect**(
LPCDLGTEMPLATE lpDialogTemplate,
CWnd *pParentWnd = NULL)

или

BOOL CDialog::**InitModalIndirect**(
HGLOBAL hDialogTemplate,
CWnd *pParentWnd = NULL) —

оба варианта функции инициализируют объект модального блока диалога, используя шаблон, предварительно созданный в памяти, на который указывает либо lpDialogTemplate — указатель на структуру DLGTEMPLATE, либо hDialogTemplate — дескриптор глобальной памяти, в которой определены структура DLGTEMPLATE и данные для каждого элемента управления.

Примечание

Обратите особое внимание на то, что система Windows после выполнения функции CDialog::InitModalIndirect создает блок диалога, но не отображает его. Он появляется на экране только после непосредственного (явного) вызова функции CDialog::DoModal.

Рассмотрим теперь создание немодального блока диалога. Первый способ предполагает наличие ресурса шаблона. Сначала вызывается пустой конструктор для создания объекта (так же, как при создании модального блока диалога на основе шаблона в памяти). Затем для создания немодального блока диалога и присоединения его к объекту CDialog вызывается функция:

BOOL CDialog::**Create**(
LPCTSTR lpszTemplateName,
CWnd *pParentWnd = NULL)

или

BOOL CDialog::**Create**(
UINT nIDTemplate,
CWnd *pParentWnd = NULL) —

используемые параметры аналогичны параметрам конструктора для модального блока диалога; эту функцию можно вызвать прямо внутри конструктора класса, производного от CDialog.

Второй способ создания немодального блока диалога опирается на шаблон, предварительно созданный в памяти. Этот способ по своей сути ничем не отличается от аналогичного для модального блока диалога, за исключением того, что вместо CDialog::InitModalIndirect вызывается функция:

BOOL CDialog::**CreateIndirect**(
LPCDLGTEMPLATE lpDialogTemplate,
CWnd *pParentWnd = NULL)

или

BOOL CDialog::**CreateIndirect**(
HGLOBAL hDialogTemplate,
CWnd *pParentWnd = NULL)

Еще одним отличием является то, что если при создании модального блока диалога не было необходимости устанавливать стиль WS_VISIBLE, т. к. такой блок диалога становится видимым автоматически, то для немодального блока диалога наличие или отсутствие этого стиля определяет, будет ли он отображаться на экране после своего создания. При отсутствии этого стиля необходим явный вызов функции CWnd::ShowWindow для вывода блока диалога на экран.

В отличие от функции CDialog::DoModal, функции CDialog::Create и CDialog::CreateIndirect завершаются сразу же после создания блока диалога Windows.

После того как работа с немодальным блоком диалога закончена, необходимо вызвать функцию CWnd::DestroyWindow, чтобы уничтожить его.

Несмотря на то что блок диалога представляет собой стандартное окно Windows, для него имеются свои специфические сообщения. Одним из основных является WM_INITDIALOG. Это сообщение посылается процедуре блока диалога во время вызова функций Create, CreateIndirect или DoModal после того, как были созданы окна всех его элементов управления, но до того как они стали видимыми. В ответ на это сообщение процедура блока диалога инициализирует каждый элемент управления. Например, она может заполнить список элементами, с которыми потом будет работать пользователь. В ответ на сообщение WM_INITDIALOG Windows вызывает специальный обработчик:

virtual BOOL CDialog::**OnInitDialog**( )

через стандартную глобальную процедуру блока диалога, общую для всех блоков диалога библиотеки MFC.

Для того чтобы выполнить какую-либо специальную обработку, необходимо переопределить эту функцию в своем классе. При этом не забывайте вызывать функцию OnInitDialog базового класса для корректной инициализации. Функция должна возвращать нулевое значение только в том случае, когда явно устанавливает фокус ввода на какой-либо элемент управления, и ненулевое, когда фокус ввода устанавливается на первый элемент.

После того как работа с модальным блоком диалога закончена, обязательно должна вызываться функция

void CDialog::**EndDialog**(int nResult) —

> она не закрывает блок диалога сама, а только устанавливает флажок, чтобы закрыть его, как только закончится обработка текущего сообщения. Параметр nResult передается в функцию DoModal, которая использует его в качестве возвращаемого значения. Вызывать эту функцию можно в любой момент, даже в функции OnInitDialog. В этом случае блок диалога необходимо закрыть до того, как он будет выведен на экран или до установки фокуса ввода на какой-либо элемент управления.

Чаще всего пользователь ожидает увидеть в блоке диалога две кнопки — **OK** и **Cancel** (Отмена). Поэтому для вас не должно быть неожиданностью, что класс CDialog имеет встроенную поддержку обработки команд от этих кнопок. Начнем с кнопки **OK**. Обработчик:

virtual void CDialog::**OnOK**( ) —

> фиксирует данные блока диалога и обновляет соответствующие переменные вашего приложения. Если вам нужны какие-либо дополнительные действия, то можно переопределить эту функцию для модального блока диалога. В случае немодального блока диалога ее необходимо переопределить. Дело в том, что функция EndDialog, которая вызывается в функции базового класса, делает блок диалога невидимым, но не уничтожает его. Поэтому внутри переопределенной функции необходимо вызвать функцию DestroyWindow, для уничтожения соответствующего окна Windows.

Для кнопки **Cancel** в классе реализован свой обработчик:

virtual void CDialog::**OnCancel**( ) —

> просто завершает работу с модальным блоком диалога, вызывая EndDialog, и возвращает IDCANCEL. Для немодального блока диалога эту функцию необходимо переопределить и вызвать внутри нее функцию DestroyWindow.

Последняя переопределяемая функция, реализованная в классе:

virtual void CDialog::**OnSetFont**(CFont *pFont) —

> вызывается в ответ на сообщение WM_SETFONT и устанавливает шрифт, определенный в редакторе блока диалога. Если вам нужно изменить этот шрифт, то функцию следует переопределить. Установленный шрифт используется по умолчанию всеми элементами управления блока диалога.

При обсуждении способов создания блока диалога мы упомянули о том, что они имеют свою систему координат, которые называются *координатами бло-*

ка диалога и позволяют определять размеры блока диалога независимо от конкретного устройства, поскольку масштаб элементов координатной сетки определяется относительно размеров текущего шрифта. Элементы масштаба оси X координат блока диалога соответствуют примерно 1/4 средней ширины используемого шрифта, а элементы оси Y — 1/8 высоты. Для преобразования координат блока диалога в экранные координаты в классе реализована функция:

void CDialog::**MapDialogRect**(LPRECT lpRect) —

на входе получает размеры и координаты блока диалога через указатель lpRect (в системе координат блока диалога) и заменяет значения в соответствующих полях на экранные координаты.

Для перемещения по элементам управления блока диалога можно использовать клавишу <Tab> или мышь. Однако возникают ситуации, когда такого рода переходы необходимо выполнить программно. Для реализации программных перемещений можно использовать следующие функции:

void CDialog::**NextDlgCtrl**( )

и

void CDialog::**PrevDlgCtrl**( ) —

устанавливают фокус на следующий или, соответственно, на предыдущий элемент управления. Все переходы осуществляются циклически, т. е. с последнего элемента управления фокус устанавливается на первый (функция NextDlgCtrl), а с первого — на последний элемент управления (функция PrevDlgCtrl).

Примечание

Порядковый номер элемента управления задается последовательно при его помещении в блок диалога. В дальнейшем первоначальный порядок может быть изменен.

Для перемещения фокуса на какой-либо определенный элемент управления можно использовать функцию:

void CDialog::**GotoDlgCtrl**(CWnd *pWndCtrl) —

параметр pWndCtrl определяет указатель на объект "элемент управления" (дочернее окно), на который перемещается фокус ввода. Для получения этого указателя можно воспользоваться одной из следующих функций:

CWnd* CWnd::**GetDlgItem**(int nID)

или

void CWnd::**GetDlgItem**(
int nID,
HWND *phWnd) —

по идентификатору элемента управления блока диалога возвращают либо указатель на соответствующий оконный объект, либо указатель на дескриптор окна этого объекта.

В заключение обзора членов класса рассмотрим две функции, которые позволяют узнать или переустановить "кнопку, действующую по умолчанию". При открытии блока диалога одна из кнопок имеет фокус по умолчанию, т. е. действие, приписанное ей, будет выполнено при нажатии пользователем клавиши <Enter> (для нее в шаблоне установлен стиль DEFPUSHBUTTON):

DWORD CDialog::**GetDefID**( ) —

если для кнопки определен идентификатор, то функция возвращает его в младшем слове, старшее слово при этом содержит значение DC_HASDEFID. Обычно возвращается IDOK (кнопка **OK**).

void CDialog::**SetDefID**(UINT nID) —

изменяет кнопку блока диалога, действующую по умолчанию. Параметр nID определяет идентификатор новой кнопки.

Теперь, после того как мы познакомились с возможностями класса CDialog, рассмотрим пример простейшего блока диалога. Он настолько прост, что даже нет необходимости создавать свой класс для его поддержки. Мы все же сформируем класс CAppAbout, чтобы можно было наглядно продемонстрировать разницу между модальным и немодальным блоками диалога.

Такой, или аналогичный ему, блок диалога фактически стал обязательным элементом любого современного приложения, которое выводит его на экран, чтобы сообщить пользователю некоторые сведения о программе, ее авторе, номере версии и т. д. Он обычно появляется при выборе соответствующего элемента в меню справки, как это показано на рис. 14.2. Сам блок диалога имеет вид, показанный на рис. 14.3. Для того чтобы не дублировать код, мы ограничимся только теми добавлениями, которые внесены в приложение MDI.

Рис. 14.2. Элемент **О программе** меню **?**

Рис. 14.3. Блок диалога

```
/////////////////////////////////////////////////////////
// Файл CAppAbout.h
// Copyright (c) 1999 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////
class CAppAbout : public CDialog
```

```
{
// Construction
public:
  CAppAbout(CWnd* pParent=NULL);  // standard constructor
// Dialog Data
  //{{AFX_DATA(CAppAbout)
  enum { IDD = IDD_ABOUTBOX };
  // NOTE: the ClassWizard will add data members here
  //}}AFX_DATA
// Overrides
  // ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CAppAbout)
  protected:
  // DDX/DDV support
  virtual void DoDataExchange(CDataExchange* pDX);
  //}}AFX_VIRTUAL
// Implementation
protected:
// Generated message map functions
  //{{AFX_MSG(CAppAbout)
  // NOTE: the ClassWizard will add member functions here
  //}}AFX_MSG
DECLARE_MESSAGE_MAP()
};
```

Добавления, внесенные в реализацию членов этого класса:

```
/////////////////////////////////////////////////////////
// Файл CAppAbout.cpp
// Copyright (c) 1999 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////
// Блок диалога CAppAbout: реализация класса
CAppAbout::CAppAbout(CWnd* pParent /*=NULL*/) :
  Dialog(CAppAbout::IDD, pParent) // Можно и не делать (см. пример главы 10)
{
  Create(CAppAbout::IDD, pParent);
  //{{AFX_DATA_INIT(CAppAbout)
  //}}AFX_DATA_INIT
}
```

```
void CAppAbout::DoDataExchange(CDataExchange* pDX)
{
  CDialog::DoDataExchange(pDX);
  //{{AFX_DATA_MAP(CAppAbout)
  //}}AFX_DATA_MAP
}
BEGIN_MESSAGE_MAP(CAppAbout, CDialog)
  //{{AFX_MSG_MAP(CAppAbout)
  // No message handlers
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

Изменения в классе CMDIApp:

```
/////////////////////////////////////////////////////////
// Файл MDI.h
// Copyright (c) 1999 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////
// Объект приложения CMDIApp: интерфейс класса
class CMDIApp : public CWinApp
{
...
// Implementation
  //{{AFX_MSG(CMDIApp)
  afx_msg void OnAppAbout();
  //}}AFX_MSG
DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////////
// Файл MDI.cpp
// Copyright (c) 1999 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////
BEGIN_MESSAGE_MAP(CMDIApp, CWinApp)
  //{{AFX_MSG_MAP(CMDIApp)
  ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()
// App command to run the dialog
void CMDIApp::OnAppAbout()
```

```
{
  CAppAbout aboutDlg;
  aboutDlg.DoModal();
}
```

Во включаемый файл добавляется идентификатор:

```
/////////////////////////////////////////////////////////
// Файл resource.h
// Copyright (c) 1999 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////
...
#define IDD_ABOUTBOX                    100
...
```

В файл ресурса добавляется шаблон блока диалога:

```
/////////////////////////////////////////////////////////
// Файл MDI.rc
// Copyright (c) 1999 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////

/////////////////////////////////////////////////////////
// Dialog
IDD_ABOUTBOX DIALOG DISCARDABLE  0, 0, 217, 55
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "О программе MDI"
FONT 8, "MS Sans Serif"
BEGIN
  ICON  IDR_MAINFRAME,IDC_STATIC,7,7,21,20
  CTEXT "MDI-приложение Версия 1.1",
    IDC_STATIC,51,17,119,8, SS_NOPREFIX
  CTEXT  "Copyright © 1999",IDC_STATIC,127,34,71,8
  DEFPUSHBUTTON   "OK",IDOK,178,7,32,14,WS_GROUP
  LTEXT  "Мешков А., Тихомиров Ю.",IDC_STATIC,35,34,89,8
END
```

В шаблон меню необходимо добавить следующий пункт:

```
POPUP "&?"
  BEGIN
    MENUITEM "&О программе ...",              ID_APP_ABOUT
  END
```

Как видите, все достаточно просто. Блок диалога появляется на экране при выборе пользователем пункта **О программе** в меню справки (**?**). Заметим, что такой путь является стандартным для вызова блока диалога **О программе**. После того как данный пункт меню выбран, вызывается обработчик OnAppAbout команды ID_APP_ABOUT, в котором создается модальный блок диалога.

Для того чтобы вместо модального блока диалога создать немодальный, достаточно сделать следующее: изменить конструктор и обработчик команды ID_APP_ABOUT, как показано ниже:

```
CAppAbout::CAppAbout(CWnd* pParent)
{
...
 Create(CAppAbout::IDD, pParent);
...
}
void CMDIApp::OnAppAbout()
{
  CAppAbout *aboutDlg = new CAppAbout();
}
```

Теперь, если вы выведете на экран блок диалога **О программе**, можно продолжать работу с любым окном приложения без каких-либо ограничений.

Приведенный пример полностью иллюстрирует общий подход к созданию блоков диалога на основе шаблона, определенного в файле ресурсов. Теперь, опираясь на полученные знания, рассмотрим пример приложения, который поможет нам изучить еще несколько новых интересных вопросов:

- Создание приложения, в котором в качестве главного окна будет выступать блок диалога
- Обмен данными с блоком диалога
- Создание блока диалога на основе шаблона в памяти

Template — пример приложения

Начнем непосредственно с кода.

```
/////////////////////////////////////////////////////////
// Файл Template.h
// Copyright (c) 1999 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////
#ifndef __AFXWIN_H__
  #error include 'stdafx.h' before including this file for PCH
```

```
#endif
#include "resource.h"  // main symbols
/////////////////////////////////////////////////////////////////////////////
// CTemplateApp:
class CTemplateApp : public CWinApp
{
public:
  CTemplateApp();
// Overrides
  // ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CTemplateApp)
  public:
  virtual BOOL InitInstance();
  //}}AFX_VIRTUAL
// Implementation
  //{{AFX_MSG(CTemplateApp)
  //}}AFX_MSG
DECLARE_MESSAGE_MAP()
};
/////////////////////////////////////////////////////////////////////////////
// Файл Template.cpp
// Copyright (c) 1999 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////////////////////////
#include "stdafx.h"
#include "Template.h"
#include "TemplateDlg.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
// CTemplateApp
BEGIN_MESSAGE_MAP(CTemplateApp, CWinApp)
  //{{AFX_MSG_MAP(CTemplateApp)
  //}}AFX_MSG
END_MESSAGE_MAP()
// Конструктор CTemplateApp
CTemplateApp::CTemplateApp()
{
```

```
}
// The one and only CTemplateApp object
CTemplateApp theApp;
// CTemplateApp initialization
BOOL CTemplateApp::InitInstance()
{
#ifdef _AFXDLL
  Enable3dControls();       // shared DLL
#else
  Enable3dControlsStatic(); // statically
#endif
  // Создаем объект "блок диалога"
  CTemplateDlg dlg;
  // Поскольку блок диалога является главным ресурсом приложения,
  // указатель на этот объект присваивается переменной m_pMainWnd
  m_pMainWnd = &dlg;
  // Отображаем его на экране
  int nResponse = dlg.DoModal();
  return FALSE;
}
/////////////////////////////////////////////////////////
// Файл TemplateDlg.h
// Copyright (c) 1999 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////
// Класс CItem для создания элементов управления
class CItem
{
public:
  DLGITEMTEMPLATE  m_ctrItem;
  enum ctrtype {EDITTEXT = 0x0081, STATICTEXT};
  ctrtype m_ctrtype;
  CString m_strTitle;
public:
  CItem(enum ctrtype cType);
  CItem() {};
  void Initialize(enum ctrtype cType, UINT nID,
    CRect *prect = NULL,
    LPCTSTR pszCaption = NULL);
};
```

```
// Класс CMemory шаблона блока диалога
class CMemory
{
public:
  DLGTEMPLATE m_dlgMemory;
  CItem m_dlgItem[2]; // В шаблоне будет 2 элемента
  CMemory();
  void Show();
};
// Класс CTemplateDlg главного окна приложения
class CTemplateDlg : public CDialog
{
public:
  CTemplateDlg(CWnd* pParent = NULL);
// Dialog Data
  //{{AFX_DATA(CTemplateDlg)
  enum { IDD = IDD_TEMPLATE_DIALOG };
  CString m_strTitle;
  UINT m_x;
  UINT m_y;
  UINT m_nWidth;
  UINT m_nHeight;
  int m_nItem;
  CString m_strItem;
  //}}AFX_DATA
  int m_nActiveItem;
  CMemory m_dlgTemplate;
// ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CTemplateDlg)
  protected:
// Поддержка обмена данными с блоком диалога (DDX/DDV)
  virtual void DoDataExchange(CDataExchange* pDX);
  //}}AFX_VIRTUAL
// Implementation
protected:
  HICON m_hIcon;
  void InfoToDialog(int nItem, LPCTSTR lpszItem);
// Generated message map functions
  //{{AFX_MSG(CTemplateDlg)
```

```
  virtual BOOL OnInitDialog();
  afx_msg void OnShow();
  afx_msg void OnEdit();
  afx_msg void OnStatic();
  //}}AFX_MSG
DECLARE_MESSAGE_MAP()
};
/////////////////////////////////////////////////////////
// Файл TemplateDlg.cpp
// Copyright (c) 1999 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////
#include "stdafx.h"
#include "Template.h"
#include "TemplateDlg.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
// CItem - элементы управления шаблона в памяти
CItem::CItem(enum ctrtype cType)
{
  m_ctrtype = cType;
}
void CItem::Initialize(enum ctrtype cType, UINT nID,
  CRect* prect, LPCTSTR pszCaption)
{
  m_ctrtype = cType;
  // Заполняем поля структуры DLGITEMTEMPLATE
  if (prect != NULL)
  {
    m_ctrItem.x = (short)prect->left;
    m_ctrItem.y = (short)prect->top;
    m_ctrItem.cx = (short)prect->Width();
    m_ctrItem.cy = (short)prect->Height();
  }
  else
  {
    m_ctrItem.x = 0;
```

```
    m_ctrItem.y = 0;
    m_ctrItem.cx = 20;
    m_ctrItem.cy = 20;
  }
  m_ctrItem.dwExtendedStyle = 0;
  m_ctrItem.id = nID;
  switch (m_ctrtype)
  {
    case STATICTEXT:
      m_ctrItem.style = WS_CHILD | WS_VISIBLE | SS_CENTER;
      break;
    case EDITTEXT:
      m_ctrItem.style = WS_CHILD | WS_VISIBLE |
        ES_MULTILINE | ES_LEFT |
        WS_EX_CLIENTEDGE | WS_BORDER;
      break;
    default:
      ASSERT(FALSE);
  }
  m_strTitle = (pszCaption != NULL) ? pszCaption : _T("");
}
// CMemory - шаблон в памяти
#define SETRECT(rect, l, t, r, b) \
  rect.left = l; \
  rect.top = t; \
  rect.right = r; \
  rect.bottom = b;
CMemory::CMemory()
{
  // Инициализация элементов шаблона блока диалога
  CRect rect(40, 10, 200, 35);
  m_dlgItem[0].Initialize(CItem::STATICTEXT,
    IDC_TEXTSTATIC, &rect, _T("Статический текст"));
  SETRECT(rect, 40, 32, 200, 96);
  m_dlgItem[1].Initialize(CItem::EDITTEXT,
    IDC_TEXTEDIT, &rect, _T("Редактируемый текст"));
  // Инициализация структуры DLGTEMPLATE
  m_dlgMemory.cx = 240;
  m_dlgMemory.cy = 104;
```

```
  m_dlgMemory.style = WS_CAPTION  | WS_VISIBLE |
                WS_DLGFRAME | WS_POPUP | DS_MODALFRAME | DS_SETFONT;
  m_dlgMemory.dwExtendedStyle = 0;
  m_dlgMemory.x = 0;
  m_dlgMemory.y = 0;
  m_dlgMemory.cdit = 2;
}
void CMemory::Show()
{
  // Прежде всего, необходимо выделить память под шаблон
  WCHAR szBoxTitle[] =
    L"Шаблон блока диалога в памяти: Выход — <ESC>";
  WCHAR szFontName[] = L"Courier New Cyr";
  // Определяем размер структуры DLGTEMPLATE и заголовка
  // блока диалога
  int nBufferSize = sizeof(DLGTEMPLATE) + // Структура
    (2*sizeof(WORD)) +                    // Меню и класс
    sizeof(szBoxTitle) +                  // Заголовок
    sizeof(WORD)+sizeof(szFontName);      // Шрифт
  nBufferSize = (nBufferSize + 3) & ~3;
  // Добавляем размеры структуры DLGITEMTEMPLATE
  // и заголовков элементов
  for (int i = 0; i < 2; i++)
  {
    int nItemLength = (sizeof(DLGITEMTEMPLATE) +
      3*sizeof(WORD) +
      (m_dlgItem[i].m_strTitle.GetLength()+1)*sizeof(WCHAR));
    if (i != 1)
      nItemLength = (nItemLength + 3) & ~3;
    nBufferSize += nItemLength;
  }
  // Выделяем память
  HLOCAL hLocal = LocalAlloc(LHND, nBufferSize);
  if (hLocal == NULL)
  {
    // Обработка ошибки распределения памяти
  }
  BYTE *pBuffer = (BYTE*)LocalLock(hLocal);
  if (pBuffer == NULL)
```

```
{
  LocalFree(hLocal);
}
BYTE *pDest = pBuffer;
// Переписываем структуру DLGTEMPLATE в буфер
memcpy(pDest, &m_dlgMemory, sizeof(DLGTEMPLATE));
pDest += sizeof(DLGTEMPLATE);
*(WORD*)pDest = 0; // Меню нет
*(WORD*)pDest = 0; // Используем класс по умолчанию
pDest += 2*sizeof(WORD);
memcpy(pDest, szBoxTitle, sizeof(szBoxTitle));
pDest += sizeof(szBoxTitle);
*(WORD*)pDest = 10; // Размер шрифта
pDest += sizeof(WORD);
memcpy(pDest, szFontName, sizeof(szFontName));
pDest += sizeof(szFontName);
// Переписываем информацию о каждом элементе шаблона в буфер
for (i = 0; i < 2; i++)
{
  pDest = (BYTE*)(((DWORD)pDest + 3) & ~3);
  memcpy(pDest, (void*)&m_dlgItem[i].m_ctrItem,
    sizeof(DLGITEMTEMPLATE));
  pDest += sizeof(DLGITEMTEMPLATE);
  *(WORD*)pDest = 0xFFFF; // Есть идентификатор ресурса
  pDest += sizeof(WORD);
  *(WORD*)pDest = m_dlgItem[i].m_ctrtype;
  pDest += sizeof(WORD);
  // Переписываем заголовок, предварительно преобразовав
  // его в строку Unicode
  WCHAR *pchCaption;
  int nChar;
  nChar = m_dlgItem[i].m_strTitle.GetLength() + 1;
  pchCaption = new WCHAR[nChar];
  nChar = MultiByteToWideChar(CP_ACP, 0,
    m_dlgItem[i].m_strTitle, -1, pchCaption, nChar);
  ASSERT(nChar > 0);
  memcpy(pDest, pchCaption, nChar*sizeof(WCHAR));
  pDest += nChar*sizeof(WCHAR);
  delete pchCaption;
```

```
    *(WORD*)pDest = 0;
    pDest += sizeof(WORD);
  }
  ASSERT(pDest - pBuffer == nBufferSize);
  CDialog dlg;
  dlg.InitModalIndirect((DLGTEMPLATE*)pBuffer);
  dlg.DoModal();
  LocalUnlock(hLocal);
  LocalFree(hLocal);
}
// Блок диалога CTemplateDlg
CTemplateDlg::CTemplateDlg(CWnd* pParent) :
  CDialog(CTemplateDlg::IDD, pParent)
{
  m_nActiveItem = -1;
  //{{AFX_DATA_INIT(CTemplateDlg)
  m_strTitle = _T("");
  m_x = 0;
  m_y = 0;
  m_nWidth = 0;
  m_nHeight = 0;
  m_nItem = -1;
  m_strItem = _T("");
  //}}AFX_DATA_INIT
  m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}
void CTemplateDlg::DoDataExchange(CDataExchange* pDX)
{
  CDialog::DoDataExchange(pDX);
  //{{AFX_DATA_MAP(CTemplateDlg)
  DDX_Text(pDX, IDC_TITLE, m_strTitle);
  DDX_Text(pDX, IDC_XPOS, m_x);
  DDX_Text(pDX, IDC_YPOS, m_y);
  DDX_Text(pDX, IDC_WIDTH, m_nWidth);
  DDX_Text(pDX, IDC_HEIGHT, m_nHeight);
  DDX_Radio(pDX, IDC_STATICTEXT, m_nItem);
  DDX_Text(pDX, IDC_CAPTION, m_strItem);
  //}}AFX_DATA_MAP
}
```

```
BEGIN_MESSAGE_MAP(CTemplateDlg, CDialog)
  //{{AFX_MSG_MAP(CTemplateDlg)
  ON_BN_CLICKED(ID_SHOW, OnShow)
  ON_BN_CLICKED(IDC_EDITTEXT, OnEdit)
  ON_BN_CLICKED(IDC_STATICTEXT, OnStatic)
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()
/////////////////////////////////////////////////////////////
// Обработчики сообщений класса CTemplateDlg
BOOL CTemplateDlg::OnInitDialog()
{
 CDialog::OnInitDialog();
  SetIcon(m_hIcon, TRUE);
  m_nItem = 0;
  OnStatic();
  return TRUE;
}
void CTemplateDlg::OnShow()
{
  InfoToDialog(m_nActiveItem, NULL);
  m_dlgTemplate.Show();
}
void CTemplateDlg::OnEdit()
{
  InfoToDialog(1, _T("Поле ввода текста"));
}
void CTemplateDlg::OnStatic()
{
  InfoToDialog(0, _T("Статический текст"));
}
void CTemplateDlg::InfoToDialog(int nItem,
  LPCTSTR lpszItem)
{
  if (m_nActiveItem != -1)
  {
    UpdateData(TRUE);
    m_dlgTemplate.m_dlgItem[m_nActiveItem].m_ctrItem.x = m_x;
    m_dlgTemplate.m_dlgItem[m_nActiveItem].m_ctrItem.y = m_y;
```

```
    m_dlgTemplate.m_dlgItem[m_nActiveItem].m_ctrItem.cx =
      m_nWidth;
    m_dlgTemplate.m_dlgItem[m_nActiveItem].m_ctrItem.cy =
      m_nHeight;
    m_dlgTemplate.m_dlgItem[m_nActiveItem].m_strTitle =
      m_strTitle;
  }
  if (m_nActiveItem != nItem)
  {
    m_x = m_dlgTemplate.m_dlgItem[nItem].m_ctrItem.x;
    m_y = m_dlgTemplate.m_dlgItem[nItem].m_ctrItem.y;
    m_nWidth = m_dlgTemplate.m_dlgItem[nItem].m_ctrItem.cx;
    m_nHeight = m_dlgTemplate.m_dlgItem[nItem].m_ctrItem.cy;
    m_strTitle = m_dlgTemplate.m_dlgItem[nItem].m_strTitle;
    ASSERT(lpszItem != NULL);
    m_strItem = lpszItem;
    m_nItem = nItem;
    m_nActiveItem = nItem;
    UpdateData(FALSE);
  }
}
/////////////////////////////////////////////////////////////
// Файл Template.rc
// Copyright (c) 1999 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////////
//Microsoft Developer Studio generated resource script.
#include "resource.h"
#define APSTUDIO_READONLY_SYMBOLS
/////////////////////////////////////////////////////////////
// Generated from the TEXTINCLUDE 2 resource.
#include "afxres.h"
#undef APSTUDIO_READONLY_SYMBOLS
/////////////////////////////////////////////////////////////
// Russian resources
#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_RUS)
#ifdef _WIN32
LANGUAGE LANG_RUSSIAN, SUBLANG_DEFAULT
#pragma code_page(1251)
```

```
#endif //_WIN32
#ifdef APSTUDIO_INVOKED
/////////////////////////////////////////////////////
// TEXTINCLUDE
1 TEXTINCLUDE DISCARDABLE
BEGIN
  "resource.h\0"
END
2 TEXTINCLUDE DISCARDABLE
BEGIN
  "#include ""afxres.h""\r\n"
  "\0"
END
3 TEXTINCLUDE DISCARDABLE
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 9, 1\r\n"
"#pragma code_page(1252)\r\n"
"#endif\r\n"
// non-Microsoft Visual C++ edited resources\r\n"
"#include ""res\\Template.rc2""
"#include ""afxres.rc"" // Standard components\r\n"
"#endif\0"
END
#endif    // APSTUDIO_INVOKED
/////////////////////////////////////////////////////
// Icon
IDR_MAINFRAME ICON DISCARDABLE  "res\\Template.ico"
/////////////////////////////////////////////////////
// Dialog
IDD_TEMPLATE DIALOGEX 0, 0, 275, 150
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
EXSTYLE WS_EX_APPWINDOW
```

```
CAPTION " Главное окно приложения"
FONT 8, "MS Sans Serif"
BEGIN
  DEFPUSHBUTTON   "Показать",ID_SHOW,218,7,50,14
  DEFPUSHBUTTON   "Выход",IDCANCEL,218,23,50,14
  GROUPBOX        "Тип элемента управления",
                  IDC_STATIC,17,49,102,56
  CONTROL         "Статический текст",
                  IDC_STATICTEXT,"Button",
                  BS_AUTORADIOBUTTON | WS_GROUP |
                  WS_TABSTOP,25,66,90,10
  CONTROL         "Редактируемый текст",
                  IDC_EDITTEXT,"Button",
                  BS_AUTORADIOBUTTON |
                  WS_TABSTOP,25,86,90,10
  ICON            IDR_MAINFRAME,
                  IDC_STATIC,17,7,18,20,SS_SUNKEN |
                  WS_GROUP
  GROUPBOX        "Параметры элемента управления",
                  IDC_CAPTION,132,49,128,82
  EDITTEXT        IDC_XPOS,156,62,30,14,ES_AUTOHSCROLL
  EDITTEXT        IDC_YPOS,156,83,30,14,ES_AUTOHSCROLL
  EDITTEXT        IDC_WIDTH,220,62,30,14,ES_AUTOHSCROLL |
                  ES_NUMBER
  EDITTEXT        IDC_HEIGHT, 220,83,30,14,ES_AUTOHSCROLL
  EDITTEXT        IDC_TITLE,162,111,88,14,ES_AUTOHSCROLL
  LTEXT           "Ширина:",IDC_STATIC,189,65,29,8
  LTEXT           "X:",IDC_STATIC,143,64,8,8
  LTEXT           "Y:",IDC_STATIC,143,87,8,8
  LTEXT           "Высота:",IDC_STATIC,189,86,28,8
  LTEXT           "Текст:",IDC_STATIC,138,113,23,8
  CTEXT           "Выберите тип элемента управления и
↳задайте его размеры и выводимый текст",
                  IDC_STATIC,71,7,111,24
END
/////////////////////////////////////////////////////////
// DESIGNINFO
#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
```

```
BEGIN
  IDD_TEMPLATE, DIALOG
  BEGIN
   LEFTMARGIN, 7
   RIGHTMARGIN, 268
   TOPMARGIN, 7
   BOTTOMMARGIN, 142
  END
END
#endif    // APSTUDIO_INVOKED
#endif    // Russian resources
#ifndef APSTUDIO_INVOKED
/////////////////////////////////////////////////////////
// Generated from the TEXTINCLUDE 3 resource.
//
#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 9, 1
#pragma code_page(1252)
#endif
// non-Microsoft Visual C++ edited resources
#include "res\Template.rc2"
#include "afxres.rc"         // Standard components
#endif
/////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
/////////////////////////////////////////////////////////
// Файл resource.h
// Copyright (c) 1999 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////
//{{NO_DEPENDENCIES}}
// Microsoft Developer Studio generated include file.
// Used by Template.rc
//
#define IDD_TEMPLATE_DIALOG             102
```

```
#define IDD_TEMPLATE                    102
#define IDR_MAINFRAME                   128
#define IDC_STATICTEXT                  1000
#define IDC_EDITTEXT                    1001
#define IDC_XPOS                        1003
#define IDC_YPOS                        1004
#define IDC_WIDTH                       1005
#define IDC_HEIGHT                      1006
#define IDC_TITLE                       1007
#define ID_SHOW                         1008
#define IDC_CAPTION                     1009
#define IDC_TEXTSTATIC                  1010
#define IDC_TEXTEDIT                    1011
// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE        129
#define _APS_NEXT_COMMAND_VALUE         32771
#define _APS_NEXT_CONTROL_VALUE         1010
#define _APS_NEXT_SYMED_VALUE           103
#endif
#endif
/////////////////////////////////////////////////////////
// Файл stdafx.h
// Copyright (c) 1999 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////
// stdafx.h : include file for standard system
// include files,or project specific include files
// that are used frequently, but are changed infrequently
//
// Exclude rarely-used stuff from Windows headers
#define VC_EXTRALEAN
#include <afxwin.h>   // MFC core and standard components
#include <afxext.h>   // MFC extensions
// Для подключения функции MultiByteToWideChar
#include <winnls.h>
```

Рассмотрим основные положения этой программы.

Блок диалога в качестве главного окна приложения

Для реализации этой возможности нужно проделать всего три шага. Прежде всего, в ресурсах определить шаблон блока диалога. Способы, которыми это можно сделать, мы подробно рассмотрели выше. На рис. 14.4 показан блок диалога, созданный для нашего приложения.

Рис. 14.4. Главное окно приложения является блоком диалога

Вторым шагом является создание класса на основе CDialog:

```
class CTemplateDlg : public CDialog
{
public:
  CTemplateDlg(CWnd* pParent = NULL);
...
};
```

Примечание

Мы больше не говорим о необходимости создания класса приложения. Сделать это нужно, но ничего нового в этом нет.

И, наконец, необходимо создать объект этого класса, что мы и делаем в переопределенной функции InitInstance:

```
BOOL CTemplateApp::InitInstance()
{
...
  CTemplateDlg dlg;
```

```
  m_pMainWnd = &dlg;
  int nResponse = dlg.DoModal();
...
}
```

После создания окно сразу выводится на экран. Как видите, все настолько просто, что даже нет смысла останавливаться на этом более подробно. Поэтому мы переходим к новой теме, связанной с взаимодействием между пользователем и блоком диалога.

Обмен данными с блоком диалога

Для того чтобы можно было обмениваться данными с блоком диалога, необходимо прежде всего определить в вашем классе соответствующие члены, которые смогут принимать и передавать эти данные:

Примечание

Рассматриваемый ниже механизм справедлив для любого класса, производного от CWnd, а не только для блоков диалога. Однако поскольку чаще всего он используется именно при работе с блоками диалога, мы позволили себе использовать термин "обмен данными с блоком диалога".

```
class CTemplateDlg : public CDialog
{
...
// Dialog Data
  // Специальный формат AFX комментариев для ClassWizard
  //{{AFX_DATA(CTemplateDlg)
  // Здесь располагаются переменные для обмена данными
  ...
  //}}AFX_DATA
  ...
  //{{AFX_VIRTUAL(CTemplateDlg)
  protected:
  // Поддержка обмена данными с блоком диалога (DDX/DDV)
  virtual void DoDataExchange(CDataExchange* pDX);
 //}}AFX_VIRTUAL
...
DECLARE_MESSAGE_MAP()
};
```

Будет уместно привести список элементов управления и соответствующих им типов переменных:

Текстовое поле (EDIT BOX)	CString, int, UINT, long, DWORD, float, double, short, BOOL
Обычный флажок (CHECK BOX)	BOOL
Флажок с тремя состояниями (CHECK BOX)	int
Первый в группе переключатель (RADIOBUTTON)	int
Неотсортированный список (LISTBOX)	CString, int
Комбинированный список (COMBOBOX)	CString, int
Все остальные виды списков	CString

Начальные значения данных устанавливаются в переопределенной функции OnInitDialog или в конструкторе нашего класса блока диалога:

```
CTemplateDlg::CTemplateDlg(CWnd* pParent):
  CDialog(CTemplateDlg::IDD, pParent)
{
...
// Специальный формат AFX комментариев для ClassWizard
  //{{AFX_DATA_INIT(CTemplateDlg)
...
  // Инициализация данных для обмена с блоком диалога
...
  //}}AFX_DATA_INIT
...
}
```

После того как определены источник и приемник данных, а также их типы, пришло время ознакомиться с тем, каким образом осуществляется обмен. Для организации обмена нужны всего две функции:

BOOL CWnd::**UpdateData**(BOOL bSaveAndValidate = TRUE) —

параметр bSaveAndValidate определяет направленность действий. Если он равен TRUE, то данные, введенные пользователем в блоке диалога, записываются в соответствующие переменные класса, в противном случае данные инициализируют элементы управления блока диалога. Она вызывается в функции CDialog::OnInitDialog с параметром, равным FALSE, для инициализации данных при создании модального блока диалога, и в функции CDialog::OnOK с параметром, равным TRUE, для получения и проверки на допустимость данных из блока диалога, если он закрыт при помощи кнопки OK. Ее основное назначение — инициализация и обработка ошибок при обращении к следующей функции:

virtual void CWnd::**DoDataExchange**(CDataExchange *pDX) —

эта функция, используя параметр pDX (указатель на объект класса CDataExchange), и "занимается" обменом данных и проверкой их корректности. Ее необходимо переопределить для каждого класса блока диалога, который должен обеспечивать взаимодействие с

пользователем. Переопределенная функция имеет следующую стандартную форму, аналогичную приведенной в рассмотренном приложении:

```
void CTemplateDlg::DoDataExchange(CDataExchange* pDX)
{
// Обязательный вызов функции базового класса
  CDialog::DoDataExchange(pDX);
// Специальный формат AFX комментариев для ClassWizard
  //{{AFX_DATA_MAP(CTemplateDlg)
  // Вызовы функций для обмена данными
   ...
  // Вызовы функций для проверки корректности данных
   ...
  //}}AFX_DATA_MAP
}
```

Общая схема обмена данными наглядно представлена на рис. 14.5.

Рис. 14.5. Общая схема обмена данными в приложении

Функция DoDataExchange во многом похожа на функцию Serialize, а ее параметр pDX очень напоминает параметр CArchive функции CObject::Serialize, например, объект pDX класса CDataExchange, так же как объект класса CArchive, имеет флаг направления:

BOOL CDataExchange::**m_bSaveAndValidate** —

указывает направление при операциях обмена данными (Dynamic Data Exchange, DDX). Значение TRUE задает передачу данных от элементов управления блока диалога к членам соответствующего класса, а значение FALSE — в обратном направлении. При операциях проверки корректности данных (Dynamic Data Validation, DDV) всегда установлен в TRUE. Значение флага определяется параметром, передаваемым функцией CWnd::UpdateData.

Кроме этого, в классе CDataExchange имеются еще четыре члена:

BOOL CDataExchange::**m_pDlgWnd** —

указатель на объект класса CWnd (обычно CDialog), для которого осуществляются обмен и проверка корректности введенных данных;

HWND CDataExchange::**PrepareCtrl**(int nIDC)

и

HWND CDataExchange::**PrepareEditCtrl**(int nIDC) —

подготавливают элементы управления к обмену данными. Подготовка заключается в сохранении дескриптора (HWND) элемента управления в классе CDataExchange. В случае сбоя они позволяют восстановить фокус ввода у последнего элемента управления, имевшего фокус. Параметр nIDC задает идентификатор элемента управления, подготавливаемый для операций DDX\DDV. Первая функция применяется для всех элементов управления, за исключением элементов редактирования, вторая работает исключительно с элементами редактирования;

void CDataExchange::**Fail**( ) —

вызывается в случае сбоя при проведении операции проверки корректности данных блока диалога. Она восстанавливает фокус у элемента управления, вызвавшего сбой, и вызывает программу обработки исключения типа CUserException для остановки операции DDV. Исключение выводит окно сообщения с объяснением причины сбоя, после чего пользователь может повторно вводить данные в этот элемент управления.

После того как вы определите свою функцию DoDataExchange, вам необходимо вставить в нее требуемые функции DDX, по одной на каждый элемент управления, участвующий в обмене. Библиотека MFC предоставляет достаточно большое число таких функций для организации автоматического обмена и проверки корректности данных. Полный их список можно посмотреть в файле afxdd_.h. То же самое можно сказать и про функции DDV. Правда, тут есть один маленький нюанс. Вы не можете вызвать функцию проверки корректности введенных пользователем данных (DDV) без вызова соответствующей функции обмена (DDX). Более того, функция DDV должна следовать непосредственно за функцией DDX этого элемента управления. Функции имеют следующий формат вызова:

`DDX_`*`XXX`*`(pDX, nIDC, member)` —

для обмена данными

и

`DDV_`*`XXX`*`(pDX, member, ...)` —

для проверки корректности введенных данных

В этих функциях задействованы следующие параметры: *XXX* — имя соответствующей функции (такой как DDX_Check или DDV_MinMaxInt), pDX — указатель на объект CDataExchange, nIDC — идентификатор элемента управления, с которым производится обмен данными, и member — член класса, участвующий в обмене.

Этот шаг, связанный с работой с блоками диалога, также достаточно прост, и мы переходим к третьей теме.

Создание блока диалога на основе шаблона в памяти

Мы уже говорили, что в этом случае объект класса создается при помощи функций CDialog::InitModalIndirect или CDialog::CreateIndirect. Однако перед их использованием нужно проделать некоторую подготовительную работу. Обратимся к рассматриваемому примеру. Как вы помните, для создания шаблона блока диалога в памяти необходимо определить и заполнить структуру DLGTEMPLATE для блока диалога и некоторое количество структур DLGITEMTEMPLATE (по числу используемых элементов управления). Задачу определения структур у нас решают два класса: CMemory и CItem, которые позволяют создать объект "блок диалога" и необходимое число объектов его элементов управления (в данном случае — "статическое поле" и "текстовое поле"). После того как объекты созданы и проинициализированы, можно создавать соответствующие окна Windows. Эту задачу у нас выполняет функция CMemory::Show:

```
void CMemory::Show()
{
  // При работе с шаблонами блоков диалога в памяти все
  // текстовые строки должны быть представлены в формате Unicode
  WCHAR szBoxTitle[] =
    L"Шаблон блока диалога в памяти: Выход - <ESC>";
  WCHAR szFontName[] = L"Courier New Cyr";

  // Прежде всего, определяем размер памяти, необходимой
  // для размещения шаблона блока диалога, и дополнительной
  // памяти под четыре массива: для меню, класса,
  // заголовка и шрифта
  int nBufferSize = sizeof(DLGTEMPLATE) + // Структура
    (2*sizeof(WORD)) +                    // Меню и класс
    sizeof(szBoxTitle) +                  // Заголовок
    sizeof(WORD)+sizeof(szFontName);      // Шрифт

  // Выравнивание по DWORD
  nBufferSize = (nBufferSize + 3) & ~3;

  // Добавляем размеры структуры DLGITEMTEMPLATE
```

```
// и заголовков элементов
for (int i = 0; i < 2; i++)
{
  int nItemLength=(sizeof(DLGITEMTEMPLATE) + // Структура
    // 3 массива для класса, заголовка и данных
    // элемента управления
    3*sizeof(WORD) +
    // Размер строки Unicode для заголовка
    (m_dlgItem[i].m_strTitle.GetLength()+1)*sizeof(WCHAR));

    // Выравнивание по DWORD
    if (i != 1)
      nItemLength = (nItemLength + 3) & ~3;
    nBufferSize += nItemLength;
}

// Выделяем память под шаблон
HLOCAL hLocal = LocalAlloc(LHND, nBufferSize);
if (hLocal == NULL)
{
// Обработка ошибки распределения памяти
}
BYTE *pBuffer = (BYTE*)LocalLock(hLocal);
if (pBuffer == NULL)
{
  LocalFree(hLocal);
  // Обработка ошибки блокировки памяти
}
BYTE *pDest = pBuffer;

// Переписываем структуру DLGTEMPLATE в буфер
memcpy(pDest, &m_dlgMemory, sizeof(DLGTEMPLATE));
pDest += sizeof(DLGTEMPLATE);
*(WORD*)pDest = 0; // Меню нет
*(WORD*)pDest = 0; // Используем класс по умолчанию
pDest += 2*sizeof(WORD);

// Переписываем заголовок
memcpy(pDest, szBoxTitle, sizeof(szBoxTitle));
```

```
pDest += sizeof(szBoxTitle);
*(WORD*)pDest = 10; // размер шрифта
pDest += sizeof(WORD);

// Переписываем имя шрифта
memcpy(pDest, szFontName, sizeof(szFontName));
pDest += sizeof(szFontName);

// Переписываем информацию о каждом элементе шаблона
for (i = 0; i < 2; i++)
{
  // Выравнивание буфера по DWORD
  pDest = (BYTE*)(((DWORD)pDest + 3) & ~3);
  memcpy(pDest, (void*)&m_dlgItem[i].m_ctrItem,
    sizeof(DLGITEMTEMPLATE));
  pDest += sizeof(DLGITEMTEMPLATE);
  *(WORD*)pDest = 0xFFFF; // Есть идентификатор ресурса
  pDest += sizeof(WORD);
  *(WORD*)pDest = m_dlgItem[i].m_ctrtype;
  pDest += sizeof(WORD);

  // Преобразуем заголовок в строку Unicode
  WCHAR *pchCaption;
  int nChar = m_dlgItem[i].m_strTitle.GetLength() + 1;
  pchCaption = new WCHAR[nChar];

  // Преобразуем строку ANSI в строку Unicode
  nChar = MultiByteToWideChar(
    CP_ACP,       // Кодовая страница ANSI
    0,            // Предварительно сформированная строка
    m_dlgItem[i].m_strTitle, // Адрес строки
    -1,                      // Автоматическое определение длины
    pchCaption, // Адрес буфера строки Unicode
    nChar);     // Размер буфера строки Unicode
  ASSERT(nChar > 0);

  // Переписываем заголовок элемента управления
  memcpy(pDest, pchCaption, nChar*sizeof(WCHAR));
  pDest += nChar*sizeof(WCHAR);
```

```
    delete pchCaption;
    *(WORD*)pDest = 0;
    pDest += sizeof(WORD);
  }

  // Проверка на полноту заполнения буфера
  ASSERT(pDest — pBuffer == nBufferSize);

  // После заполнения буфера информация в нем хранится
  // в следующем виде:
  // Структура DLGTEMPLATE
  // 0x0000 (WORD) - индикатор наличия меню
  // 0x0000 (WORD) - класс блока диалога по умолчанию
  // Заголовок - строка Unicode
  // 0x0000 - размер используемого шрифта
  // "Courier New Cyr" - имя используемого шрифта
  // Структура DLGITEMTEMPLATE для статического текста
  // 0xFFFF - есть идентификатор элемента управления
  // 0x0081 - идентификатор класса Static
  // Заголовок - строка Unicode
  // 0x0000 - дополнительных данных нет
  // Структура DLGITEMTEMPLATE для редактируемого текста
  // 0xFFFF - есть идентификатор элемента управления
  // 0x0082 - идентификатор класса Edit
  // Заголовок - строка Unicode
  // 0x0000 - дополнительных данных нет

  // Создание объекта "блок диалога"
  CDialog dlg;

  // Создание окна Windows блока диалога
  dlg.InitModalIndirect((DLGTEMPLATE*)pBuffer);

  // Отображение модального блока диалога на экране
  dlg.DoModal();

  // Освобождение использованной памяти
  LocalUnlock(hLocal);
  LocalFree(hLocal);
}
```

Функция вызывается всякий раз, когда пользователь нажимает кнопку **Показать** в главном окне приложения. Параметры, которые пользователь вводит в полях **X:**, **Y:**, **Ширина**, **Высота** и **Текст** передаются в соответствующие элементы управления, которые используют их при инициализации. В результате выполнения функции на экран выводится блок диалога, представленный на рис. 14.6.

Рис. 14.6. Блок диалога, созданный функцией CDialog::InitModalIndirect

Понятно, что приведенный пример не претендует на изящество, однако свою роль в иллюстрации рассматриваемых положений он успешно выполняет.

Глава 15

Стандартные блоки диалога

В свое время фирма Microsoft ввела понятие *стандартных блоков диалога* (Common Dialog Box), применяемых для выполнения тех или иных стандартных действий, таких, например, как открытие файла или выбор шрифта. Теперь уже для всех очевидны достоинства такого подхода, ведь использование стандартных блоков диалога означает целостность интерфейса для пользователя и меньший объем работы для программиста. Эти блоки диалога давно уже стали составной частью Windows, и с первых версий библиотеки MFC в нее, естественно, были включены классы, поддерживающие создание *стандартных* блоков диалога. В последней версии библиотека поддерживает следующие классы: CFileDialog, CColorDialog, CFontDialog, CFindReplaceDialog, CPrintDialog и CPageSetupDialog.

Ввиду достаточно частого использования стандартных блоков диалога рассмотрим их основные возможности. Все они базируются на одном классе CCommonDialog, инкапсулирующем функциональные возможности стандартных блоков диалога Windows и имеющем всего одну функцию — конструктор:

CCommonDialog::**CCommonDialog**(CWnd *pParentWnd) —

здесь, как и для класса CDialog, если параметр pParentWnd равен NULL, то родительским окном блока диалога будет главное окно приложения.

Значительно больший интерес представляют производные от него классы. Знакомство с ними начнем с класса CFileDialog, приведя краткое описание членов и функций классов. Для некоторых классов будут рассмотрены примеры, демонстрирующие основные принципы настройки внешнего вида соответствующего блока диалога.

Класс CFileDialog

Этот класс отвечает за создание и работоспособность двух блоков диалога: один позволяет выбрать файл для его последующего открытия, а другой

предназначен для выбора места расположения и имени файла при его сохранении.

Перед тем как начать рассматривать члены этого класса, скажем несколько слов об общих принципах создания этих блоков диалога на базе рассматриваемого класса.

Прежде всего, если вам не нужны никакие дополнительные настройки, вы совершенно спокойно можете использовать прямо этот класс, не создавая своего. Если же ваши запросы несколько шире (надо сказать, что процесс создания "идеального" блока диалога **File Open** (открытие файла), давно превратился в своего рода искусство), то необходимо создать свой класс на базе CFileDialog и спроектировать конструктор таким образом, чтобы удовлетворить все ваши потребности. После того как блок диалога создан, можно выполнять различные настройки его элементов управления, заполняя поля структуры типа OPENFILENAME. И, наконец, вызывается функция DoModal, для отображения блока диалога и работы с ним. Если после закрытия блока диалога функция возвращает значение IDOK, можно использовать специальные функции для получения информации о сделанном выборе.

В классе представлены следующие члены:

OPENFILENAME CFileDialog::**m_ofn** —
структура типа OPENFILENAME:

```
typedef struct tagOFN{
  DWORD lStructSize;
  HWND hwndOwner;
  HINSTANCE hInstance;
  LPCTSTR lpstrFilter;
  LPTSTR lpstrCustomFilter;
  DWORD nMaxCustFilter;
  DWORD nFilterIndex;
  LPTSTR lpstrFile;
  DWORD nMaxFile;
  LPTSTR lpstrFileTitle;
  DWORD nMaxFileTitle;
  LPCTSTR lpstrInitialDir;
  LPCTSTR lpstrTitle;
  DWORD Flags;
  WORD nFileOffset;
  WORD nFileExtension;
  LPCTSTR lpstrDefExt;
  DWORD lCustData;
  LPOFNHOOKPROC lpfnHook;
```

```
  LPCTSTR lpTemplateName;
} OPENFILENAME;
```

Используется для инициализации внешнего вида блоков диалога **File Open** (Открытие файла) и **File Save As** (Сохранение файла) после создания объекта класса, но до вызова функции DoModal. Рассмотрим поля этой структуры, чтобы иметь ясное представление о тех возможностях, которыми мы располагаем:

lStructSize — размер структуры в байтах;

hwndOwner — дескриптор окна, которое владеет блоком диалога, или NULL;

hInstance — определяет идентификатор блока памяти (установлен флаг OFN_ENABLETEMPLATEHANDLE) или дескриптор модуля (установлен флаг OFN_ENABLETEMPLATE), содержащего шаблон блока диалога. Если не установлен ни один из этих флагов, то содержимое поля игнорируется. Если не установлен флаг OFN_EXPLORER, это поле определяет шаблон, который заменяет шаблон, заданный по умолчанию. Если же в Windows 95/98 указанный флаг установлен, то система использует в качестве шаблона стиль блока диалога **Explorer** (Проводник), в котором изображается дерево, а также список файлов и каталогов в виде маленьких значков;

lpstrFilter — определяет указатель на буфер, содержащий одну или больше пар текстовых строк, задающих фильтры для выбора имен файлов (шаблоны имен файлов). Первая строка в каждой паре описывает название фильтра (например, "Исполняемые файлы"), а вторая — образец фильтра (например, "*.EXE; *.COM"). Если в единственной строке фильтра необходимо указать несколько образцов, то они должны отделяться друг от друга нулевым символом. Последняя строка в буфере должна заканчиваться двумя нулевыми символами. Порядок расположения пар строк в буфере задает порядок следования фильтров. Если этот параметр равен NULL, то в блоке диалога не определен никакой фильтр;

lpstrCustomFilter — определяет указатель на статический буфер, содержащий пару текстовых строк, для хранения образца выбранного пользователем фильтра. Первая строка описывает отображаемую строку, а вторая — образец фильтра, выбранный пользователем. При первом создании блока диалога вы определяете первую строку, которая может быть любой непустой строкой. Когда пользователь выбирает какой-либо файл, блок диалога копирует текущий образец фильтра во вторую строку. Им может быть как один из образцов, определенных в буфере lpstrFilter, так и введенный пользователем. Система использует строки для инициализации определенного пользователем фильтра при следующем создании блока диалога. Если поле nFilterIndex содержит нуль, то блок диалога использует этот настраиваемый фильтр. Блок диалога не сохраняет введенный пользователем образец фильтра, если это поле содержит NULL. В противном случае размер этого буфера должен быть указан в поле nMaxCustFilter;

nMaxCustFilter — определяет размер буфера, на который указывает lpstrCustomFilter. При работе со строками ANSI размер указывается в байтах, а для строк Unicode — в символах. Размер буфера должен быть не меньше 40 символов. Значение этого поля игнорируется, если в lpstrCustomFilter записан NULL;

nFilterIndex — определяет индекс (начиная с единицы) текущего выбранного фильтра в элементе управления **File Types** (тип файлов). Нулевой индекс говорит о том, что используется настраиваемый фильтр lpstrCustomFilter. Можно определить значение индекса при открытии блока диалога. При выборе файла в этом поле сохраняется значение индекса текущего фильтра. Если nFilterIndex равен нулю и lpstrCustomFilter равен NULL, то используется первый фильтр из буфера lpstrFilter. Если при этом также lpstrFilter равен NULL, то фильтр не используется вообще;

lpstrFile — указывает на буфер, который содержит имя файла (или путь), используемое по умолчанию при инициализации элемента управления **File Name** (имя файла), или NULL, если инициализация не требуется. При успешном возврате из функций GetOpenFileName или GetSaveFileName этот буфер будет содержать полный путь к выбранному файлу. Если размера буфера не достаточно для хранения строки, то указан-

ные функции возвращают FALSE, и первые два байта в этом поле содержат значение требуемого размера буфера;

nMaxFile — определяет размер в байтах (строки ANSI) или символах (строки Unicode) буфера, на который указывает lpstrFile. Минимальный размер буфера — 256 символов;

lpstrFileTitle — указывает на буфер, в который после выбора будет записано имя файла с расширением (без информации о пути);

nMaxFileTitle — определяет размер в байтах (строки ANSI) или символах (строки Unicode) буфера, на который указывает lpstrFileTitle. Значение в этом поле игнорируется, если lpstrFileTitle равен NULL;

lpstrInitialDir — указывает на строку, которая определяет начальный каталог. Чтобы начать работу в текущем каталоге, в это поле следует записать NULL;

lpstrTitle — указывает на строку, которая определяет заголовок блока диалога. Если это поле содержит NULL, будут использованы стандартные заголовки **Open** и **Save As**;

Flags — это поле позволяет задать различные режимы выбора файла, влияющие на внешний вид блока диалога. Допустимыми являются комбинации следующих флагов:

OFN_ALLOWMULTISELECT	Разрешает выбор нескольких файлов одновременно. При выборе пользователем более одного файла буфер, на который указывает lpstrFile, будет содержать путь к текущему каталогу, за которым через пробел следуют имена выбранных файлов с расширениями, разделенные между собой пробелами. Если блок диалога создан на основе шаблона, определенного пользователем, то элемент управления LIST BOX должен иметь стиль LBS_EXTENDEDSEL
OFN_CREATEPROMPT	Если пользователь определяет несуществующий файл, то выводится блок диалога, в котором предлагается создать файл. Этот флаг устанавливается автоматически при использовании флагов OFN_PATHMUSTEXIST и OFN_FILEMUSTEXIST
OFN_ENABLEHOOK	Разрешает использовать функцию фильтра, адрес которой указан в поле lpfnHook
OFN_ENABLETEMPLATE	Указывает на то, что поле lpTemplateName содержит указатель на имя ресурса шаблона блока диалога в модуле, определяемом hInstance
OFN_ENABLETEMPLATEHANDLE	Указывает на то, что поле hInstance используется для идентификации блока памяти, содержащего предварительно загруженный шаблон блока диалога. В этом случае игнорируется содержимое поля lpTemplateName
OFN_EXPLORER	При установленном флаге функции Windows NT GetOpenFileName и GetSaveFileName возвращают ошибку. Этот флаг может применяться только в Windows 95/98 и указывает на то, что при создании блоков диалога **Open** (открытие) и **Save As** (сохранение) используется новый стиль EXPLORER. Этот стиль используется по умолчанию, независимо от того, установлен ли этот флаг. Его применение необходимо только в том случае, если используется функция фильтра или настраиваемый шаблон. Для того чтобы не использовать новый стиль блоков диалога, необходимо снять этот флаг и обеспечить возврат значения FALSE функцией фильтра

OFN_EXTENSIONDIFFERENT	Устанавливается после возвращения из функции и указывает, что расширение выбранного файла отличается от того, которое было определено в поле lpstrDefExt. Этот флаг не устанавливается, если перед вызовом функции в поле lpstrDefExt было записано значение NULL или файл не имеет расширения
OFN_FILEMUSTEXIST	Определяет, что в поле **File Name** (имя файла) пользователь может вводить только имена существующих файлов. В противном случае на экран выводится предупреждающее сообщение. Если этот флаг определен, то автоматически устанавливается флаг OFN_PATHMUSTEXIST
OFN_HIDEREADONLY	Предписывает убрать из блока диалога флажок **Read Only** (только чтение)
OFN_LONGNAMES	Предписывает блокам диалога **Open** (открытие) и **Save As** (сохранение) отображать длинные имена файлов. В противном случае они выводятся в формате 8.3. Если для Windows 95/98 установлен флаг OFN_EXPLORER, то рассматриваемый флаг игнорируется, т. к. в этом случае всегда отображаются длинные имена файлов
OFN_NOCHANGEDIR	Устанавливает первоначальный текущий каталог, если пользователь изменил его при поиске файлов
OFN_NODEREFERENCELINKS	Предписывает блоку диалога возвратить путь и имя выбранного сокращения (shortcut) — файла с расширением LNK. В противном случае блок диалога возвращает путь и имя файла, на который ссылается сокращатель
OFN_NOLONGNAMES	Определяет, что в списке **File Name** (имя файла) не отображаются длинные имена файлов. Этот флаг игнорируется для Windows 95/98, если установлен флаг OFN_EXPLORER, т. к. в этом случае всегда отображаются длинные имена файлов
OFN_NONETWORKBUTTON	Скрывает и блокирует кнопку **Network** (сеть)
OFN_NOREADONLYRETURN	Определяет, что выбранный файл не имеет атрибута **Read Only** (только чтение) и не располагается в защищенном от записи каталоге
OFN_NOTESTFILECREATE	Определяет, что файл не создается перед закрытием блока диалога, а также отсутствие проверки на защиту от записи, переполнение диска или сетевую защиту
OFN_NOVALIDATE	Определяет, что стандартные блоки диалога допускают наличие неразрешенных символов в именах возвращаемых файлов
OFN_OVERWRITEPROMPT	Предписывает блоку диалога **Save As** (сохранение) выводить окно запроса на перезапись, если выбранный файл уже существует
OFN_PATHMUSTEXIST	Определяет, что пользователь может использовать только имеющиеся путь и имена файлов. В противном случае на экран выводится предупреждающее сообщение

OFN_READONLY	При отображении блока диалога флажок **Read Only** (только чтение) будет находиться в установленном состоянии, а при закрытии блока диалога указывает состояние флажка
OFN_SHAREWARE	Устанавливается после возвращения из функции OpenFile и указывает, что произошла ошибка при совместном доступе к файлу в сети
OFN_SHOWHELP	Предписывает блоку диалога отображать кнопку **Help** (**?**)

nFileOffset — определяет смещение первого символа имени файла относительно начала буфера, на который указывает lpstrFile;

nFileExtension — определяет смещение первого символа расширения имени файла относительно начала буфера, на который указывает lpstrFile. Если пользователь не задал расширение, и lpstrDefExt содержит NULL, то в этом поле указывается смещение до завершающего нулевого символа. Если в качестве последнего символа имени файла пользователь ввел ".", то в это поле записывается нуль;

lpstrDefExt — указатель на буфер, который содержит расширение имени файла, используемое по умолчанию. Эта строка добавляется к имени выбранного файла, если расширение не было указано. Она может иметь произвольную длину, но добавляются только первые три символа, среди которых не должно быть точки;

lCustData — определяет данные, которые система посылает функции фильтра lpfnHook. Когда система посылает сообщение WM_INITDIALOG функции фильтра, параметр lParam этого сообщения указывает на структуру OPENFILENAME, определенную при создании блока диалога. Этот указатель может использоваться для доступа к полю lCustData;

lpfnHook — указатель на функцию фильтра. Это поле игнорируется, если не установлен флаг OFN_ENABLEHOOK;

Если флаг OFN_EXPLORER не установлен, то это поле содержит указатель на функцию фильтра OFNHookProcOldStyle, которая получает сообщения, предназначенные для блока диалога. Она возвращает FALSE, чтобы передать сообщение процедуре блока диалога, выполняемой по умолчанию, и TRUE, чтобы игнорировать его;

Для Windows 95/98, в случае установленного флага OFN_EXPLORER, это поле содержит указатель на функцию фильтра OFNHookProc, которая получает извещения от блока диалога. Кроме того, она получает сообщения для всех дополнительных элементов управления, которые вы определили в шаблоне блока диалога. Для стандартных элементов управления эта функция сообщений не получает;

lpTemplateName — указатель на строку, которая содержит имя ресурса шаблона блока диалога, определенного в модуле hInstance. Для числовых идентификаторов ресурса можно использовать макрос MAKEINTRESOURCE. Это поле игнорируется, если не установлен флаг OFN_ENABLETEMPLATE.

Ниже кратко описаны основные функции класса.

```
CFileDialog::CFileDialog(
    BOOL bOpenFileDialog,
    LPCTSTR lpszDefExt = NULL,
    LPCTSTR lpszFileName = NULL,
    DWORD dwFlags = OFN_HIDEREADONLY |
    OFN_OVERWRITEPROMPT,
    LPCTSTR lpszFilter = NULL,
    CWnd *pParentWnd = NULL) —
```

конструктор класса, создает объект "стандартный блок диалога".

Если параметр bOpenDialog равен TRUE, создается объект "блок диалога **File Open** (Открытие файла)" и объект "блок диалога **File Save As** (Сохранение файла)", если bOpenDialog равен FALSE. Остальные параметры, передаваемые в функцию CFileDialog, вам уже известны из описания структуры OPENFILENAME. Исключение составляет лишь параметр lpszFilter. Здесь, в отличие от соответствующего поля структуры, в качестве разделителя между названиями и образцами фильтров используется символ "|", вместо пробела. Кроме того, строка должна заканчиваться двумя символами "|", за которыми следует завершающий NULL.

virtual int CFileDialog::**DoModal**( ) —

вызов этой функции выводит на экран модальный стандартный блок диалога для открытия или сохранения файла. Если вы хотите самостоятельно настроить параметры блока диалога, устанавливая соответствующие значения в поля структуры m_ofn, то делать это следует после создания объекта "блок диалога", но до вызова функции. При успешном завершении функция возвращает IDOK или IDCANCEL в зависимости от того, при помощи какой кнопки пользователь закрыл блок диалога. В последнем случае можно вызвать функцию Windows CommDlgExtendedError для проверки наличия ошибки.

Для того чтобы получить имя файла, определенное пользователем в блоке диалога, в классе реализованы несколько функций:

virtual CString CFileDialog::**GetPathName**( ) —

возвращает полное имя файла, включая путь.

virtual CString CFileDialog::**GetFileName**( ) —

возвращает имя файла с расширением.

virtual CString CFileDialog::**GetFileExt**( ) —

возвращает расширение имени файла.

virtual CString CFileDialog::**GetFileTitle**( ) —

возвращает имя файла без расширения.

Если при использовании этих функций для блока диалога установлен флаг OFN_ALLOWMULTISELECT, то возвращаемая строка содержит последовательность строк, первая из которых является полным путем к этой группе файлов, а последующие определяют имена выбранных файлов. В этом случае для извлечения имен файлов следует использовать функции:

POSITION CFileDialog::**GetStartPosition**( ) —

возвращает позицию первого имени файла в списке, если для блока диалога разрешен множественный выбор, или NULL, если список пуст.

CString CFileDialog::**GetNextPathName**(POSITION &pos) —

возвращает следующее полное имя файла из группы файлов, выбранных в блоке диалога. В параметре pos возвращается позиция текущего имени файла, либо NULL, если достигнут конец списка.

BOOL CFileDialog::**GetReadOnlyPref**( ) —

позволяет определить, был ли установлен (возвращается TRUE) флажок **Read Only** (только чтение) в блоке диалога.

Оставшиеся обработчики сообщений переопределяются достаточно редко, и мы приводим их для того, чтобы иметь полное представление обо всех возможностях класса:

void CFileDialog::**OnShareViolation**( ) —

вызывается, когда происходит общее нарушение.

void CFileDialog::**OnFileNameOK**( ) —

вызывается, чтобы зафиксировать имя файла, введенное в блоке диалога.

void CFileDialog::**OnLBSelChangedNotify**( ) —

вызывается, когда изменился выбор в поле списка.

void CFileDialog::**OnInitDone**( ) —

вызывается для обработки извещения CDN_INITDONE.

void CFileDialog::**OnFileNameChange**( ) —

вызывается для обработки извещения CDN_SELCHANGE.

void CFileDialog::**OnFolderChange**( ) —

вызывается для обработки извещения CDN_FOLDERCHANGE.

void CFileDialog::**OnTypeChange**( ) —

вызывается для обработки извещения CDN_TYPECHANGE.

В заключение, как и обещали, приведем небольшой пример (приложение DlgSmpl), показывающий принципы настройки блока диалога этого класса. За основу мы взяли уже рассмотренный пример MDI-приложения, поэтому здесь приведены только те добавления, которые иллюстрируют рассматриваемые положения.

```
/////////////////////////////////////////////////////////////
// Файл MainFrm.h
// Copyright (c) 1999 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////////
class CMainFrame : public CMDIFrameWnd
{
...
// Generated message map functions
protected:
  //{{AFX_MSG(CMainFrame)
...
```

```
  afx_msg void OnOpen();
  afx_msg void OnClose();
  //}}AFX_MSG
DECLARE_MESSAGE_MAP()
};
/////////////////////////////////////////////////////////////
// Файл MainFrm.cpp
// Copyright (c) 1999 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////////
// Настройка и вывод на экран стандартного блока диалога
// "Открытие файла" (File Open)
void CMainFrame::OnOpen()
{
  // Заголовок блока диалога
  TCHAR title[] =
                "Пример настройки блока диалога \"Открытие файла\"";

  // Создаем объект блока диалога
  CFileDialog fileDlg(TRUE);

  // Формируем шаблоны имен отображаемых файлов
  CString str("Все файлы (*.*)"); str += (TCHAR)NULL;
  str += "*.*"; str += (TCHAR)NULL;
  str += ("Файлы исходных текстов (cpp, h)"); str += (TCHAR)NULL;
  str += "*.cpp; *.h"; str += (TCHAR)NULL;
  str += "Исполняемые файлы (*.exe)"; str += (TCHAR)NULL;
  str += "*.exe"; str += (TCHAR)NULL;

  // Устанавливаем фильтр
  fileDlg.m_ofn.lpstrFilter = str;

  // При открытии блока диалога отображается список
  // файлов, имеющих расширение .cpp или .h
  fileDlg.m_ofn.nFilterIndex = 2;

  // Готовим буфер для полного имени выбранного файла
  TCHAR strName[_MAX_PATH];

  // Этот буфер должен быть пуст
```

```
  strName[0] = (TCHAR)NULL;
  fileDlg.m_ofn.lpstrFile = strName;
  // Передаем заголовок блока диалога
  fileDlg.m_ofn.lpstrTitle = title;

  // К флагам, устанавливаемым по умолчанию, добавляем возможность
  // выбора нескольких файлов одновременно
  fileDlg.m_ofn.Flags |= OFN_ALLOWMULTISELECT;

  // Выводим блок диалога на экран
  if (fileDlg.DoModal() == IDOK)
  {
    // Если блок диалога закрыт нажатием кнопки OK,
    // в заголовок текущего активного дочернего окна
    // выводим имя первого выбранного файла
    CChildWnd *pWnd = (CChildWnd *)GetActiveFrame();
    pWnd->SetWindowText(fileDlg.m_ofn.lpstrFile);
  }
}
```

При использовании приведенных установок блок диалога будет иметь внешний вид, показанный на рис. 15.1. А после закрытия блока диалога кнопкой **Открыть** получим общий вид главного окна приложения, показанный на рис. 15.2.

```
// Настройка и вывод на экран стандартного блока диалога
// File Save As (Сохранение файла)
void CMainFrame::OnClose()
{
  // Создаем объект блока диалога
  CFileDialog fileDlg(FALSE);

  // Определяем расширение, которое будет присваиваться
  // имени файла по умолчанию
  CString ext("hhp");
  fileDlg.m_ofn.lpstrDefExt = ext;

  // Заголовок блока диалога
  TCHAR title[] =
                "Пример настройки блока диалога \"Сохранить файл\"";
  fileDlg.m_ofn.lpstrTitle = title;
```

```
  // Задаем вывод флажка "Только для чтения" (Read Only) и
  // снимаем режим Проводник (Explorer)
  fileDlg.m_ofn.Flags &= ~(OFN_HIDEREADONLY  | OFN_EXPLORER);

  // Выводим блок диалога на экран
  if(fileDlg.DoModal() == IDOK)
  {
    // Если блок диалога закрыт нажатием кнопки OK,
    // в заголовок текущего активного дочернего окна
    // выводим имя первого выбранного файла
    CChildWnd *pWnd = (CChildWnd *)GetActiveFrame();
    pWnd->SetWindowText(fileDlg.GetFileName());
  }
}
```

Рис. 15.1. Блок диалога типа **File Open**

Рис. 15.2. Главное окно приложения DlgSmpl с окном открытого файла

При использовании приведенных установок блок диалога будет иметь внешний вид показанный на рис. 15.3.

Дальнейшие эксперименты по настройке внешнего вида представленных блоков диалога можно провести самостоятельно, наблюдая за теми изменениями, которые возникают при использовании того или иного параметра. Нам же осталось привести шаблон меню из файла ресурсов, поскольку оно изменилось по сравнению с исходным из приложения MDI:

Рис. 15.3. Блок диалога типа **File Save As**

```
//////////////////////////////////////////////////////////////
// Файл MDI.rc
// Copyright (c) 1999 Мешков А., Тихомиров Ю.
//////////////////////////////////////////////////////////////
...
IDR_CHILD MENU PRELOAD DISCARDABLE
BEGIN
    POPUP "Файл"
    BEGIN
        MENUITEM "Создать\tCtrl+N",             ID_NEW
        MENUITEM "Открыть...\tCtrl+O",          ID_OPEN
        MENUITEM "Закрыть",                     ID_FILE_CLOSE
        MENUITEM "Сохранить\tCtrl+S",           ID_FILE_SAVE
        MENUITEM "Сохранить как...",            ID_CLOSE
        MENUITEM SEPARATOR
        MENUITEM "Список файлов",         ID_FILE_MRU_FILE1, GRAYED
        MENUITEM SEPARATOR
        MENUITEM "Выход",                       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 "Вставить\tCtrl+V",            ID_EDIT_PASTE
    END
    POPUP "Вид"
    BEGIN
        MENUITEM "Набор инструментов",          ID_VIEW_TOOLBAR
        MENUITEM "Строка состояния",            ID_VIEW_STATUS_BAR
    END
    POPUP "Диалоги"
    BEGIN
        POPUP "Цвет"
        BEGIN
            MENUITEM "Черный",                      IDM_BLACK
            MENUITEM "Красный",                     IDM_RED
            MENUITEM "Синий",                       IDM_BLUE
            MENUITEM "Зеленый",                     IDM_GREEN
            MENUITEM "Белый",                       IDM_WHITE
            MENUITEM "Изменить палитру ...",        IDM_CUSTOM
        END
        MENUITEM "Шрифт ...",                   IDM_FONT
    END
    POPUP "Окно"
    BEGIN
        MENUITEM "Новое",                       ID_WINDOW_NEW
        MENUITEM "Каскадом",                    ID_WINDOW_CASCADE
        MENUITEM "Без перекрытия",              ID_WINDOW_TILE_HORZ
        MENUITEM "Упорядочить значки",          ID_WINDOW_ARRANGE
    END
    POPUP "?"
    BEGIN
        MENUITEM "О программе",                 ID_APP_ABOUT
    END
END
...
```

Этот же шаблон используется в двух следующих примерах, рассматриваемых в этой главе.

Класс CColorDialog

Этот класс отвечает за создание модального блока диалога для выбора и/или создания цветов, которые определены для вашего дисплея.

Основным членом класса, позволяющим осуществлять настройку объекта этого класса, является структура типа CHOOSECOLOR:

CHOOSECOLOR CColorDialog::**m_cc**

```
typedef struct {
  DWORD lStructSize;
  HWND hwndOwner;
  HWND hInstance;
  COLORREF rgbResult;
  COLORREF *lpCustColor;
  DWORD Flags;
  LPARAM lCustData;
  LPCCHOOKPROC lpfnHook;
  LPCTSTR lpTemplateName;
} CHOOSECOLOR;
```

Структура служит для настройки внешнего вида блока диалога и имеет следующие поля:

lStructSize — размер структуры в байтах;

hwndOwner — дескриптор окна, которое владеет блоком диалога, или NULL;

hInstance — определяет идентификатор блока памяти (установлен флаг CC_ENABLETEMPLATEHANDLE) или дескриптор модуля (установлен флаг CC_ENABLETEMPLATE), содержащего шаблон блока диалога. Если не установлен ни один из этих флагов, то содержимое поля игнорируется;

rgbResult — при создании блока диалога определяет первоначально выбранный цвет, если при этом в параметре Flags установлен флаг CC_RGBINIT. В противном случае выбирается черный цвет. При завершении работы блока диалога в это поле записывается цвет, выбранный пользователем;

lpCustColors — указатель на массив, содержащий 16 значений COLORREF для настраиваемых цветов блока диалога. Если пользователь изменяет значения этих цветов, то система сама модифицирует содержимое массива. Для того чтобы иметь возможность в дальнейшем использовать эти новые значения, массив должен быть объявлен как статический;

lCustData — определяет данные, которые система посылает функции фильтра lpfnHook. Когда система посылает сообщение WM_INITDIALOG функции фильтра, параметр lParam этого сообщения указывает на структуру CHOOSECOLOR, определенную при создании блока диалога и может использоваться для доступа к полю lCustData;

lpfnHook — указатель на функцию фильтра CCHookProc, которая может обрабатывать сообщения блока диалога. Это поле игнорируется, если не установлен флаг CC_ENABLEHOOK;

lpTemplateName — указатель на строку, которая содержит имя ресурса шаблона блока диалога, определенного в модуле hInstance, подставляемого вместо стандартного шаблона блока диалога. Для числовых идентификаторов ресурса можно использовать мак-

рос MAKEINTRESOURCE. Это поле игнорируется, если не установлен флаг CC_ENABLETEMPLATE;

Flags — это поле определяет множество битовых флагов, которые можно использовать при инициализации блока диалога. Допустимыми являются комбинации следующих флагов:

CC_ENABLEHOOK	Разрешает использовать функцию фильтра, адрес которой указан в поле lpfnHook. Используется только при инициализации блока диалога
CC_ENABLETEMPLATE	Указывает на то, что поле lpTemplateName содержит указатель на имя ресурса шаблона блока диалога в модуле, определяемом полем hInstance. Используется только при инициализации блока диалога
CC_ENABLETEMPLATEHANDLE	Указывает на то, что поле hInstance используется для идентификации блока памяти, содержащего предварительно загруженный шаблон блока диалога. В этом случае игнорируется содержимое поля lpTemplateName. Используется только при инициализации блока диалога
CC_FULLOPEN	Предписывает блоку диалога отображать дополнительные элементы управления, необходимые для создания произвольных цветов. Если этот флаг не установлен, то для отображения дополнительных элементов управления пользователь должен использовать кнопку **Define Custom Color** (определить цвет)
CC_PREVENTFULLOPEN	Блокирует кнопку **Define Custom Color**, т. е. пользователь не может определить произвольный цвет
CC_RGBINIT	Устанавливает цвет, заданный в поле rgbResult, в качестве используемого по умолчанию
FR_SHOWHELP	В блоке диалога отображается кнопка **Help** (**?**)

Ниже описаны основные функции класса.

CColorDialog::**CColorDialog**(
COLORREF clrInit = 0,
DWORD dwFlags = 0,
CWnd *pParentWnd = NULL) —

конструктор объекта "блок диалога". Параметр clrInit определяет первоначально выбранный цвет в формате RGB (Red — красный, Green — зеленый, Blue — синий).

virtual int CColorDialog::**DoModal**( ) —

выводит на экран модальный стандартный блок диалога Windows для выбора цвета. Если вы хотите самостоятельно настроить параметры блока диалога, устанавливая соответствующие значения в поля структуры m_cc, то делать это следует после создания объекта "блок диалога", но до вызова функции. При успешном завершении функция возвращает IDOK или IDCANCEL в зависимости от кнопки, при помощи которой пользователь закрыл блок диалога. В последнем случае можно вызвать функцию Windows CommDlgExtendedError для проверки наличия ошибки.

COLORREF CColorDialog::**GetColor**( ) —

позволяет получить информацию о цвете, выбранном пользователем. Вызывается после того, как завершилась функция CColorDialog::DoModal.

static COLORREF* CColorDialog::**GetSavedCustomColors**( ) —

возвращает указатель на массив, содержащий 16 дополнительных цветов, созданных пользователем. Вызывается после завершения функции CColorDialog::DoModal. Те цвета (из 16), которые пользователь не настраивал, определяют белый цвет.

void CColorDialog::**SetCurrentColor**(COLORREF clr) —

вызывается из функции CColorDialog::OnColorOK для принудительного выбора цвета, заданного в параметре clr.

virtual BOOL CColorDialog::**OnColorOK**( ) —

переопределение этой функции практически не требуется, т. к. реализация по умолчанию обеспечивает полную проверку правильности введенного цвета. Если вы переопределяете эту функцию, то для нормального завершения работы блока диалога она должна возвращать FALSE.

В заключение обещанный пример.

```
//////////////////////////////////////////////////////////////
// Файл ChildWnd.h
// Copyright (c) 1999 Мешков А., Тихомиров Ю.
//////////////////////////////////////////////////////////////
class CChildWnd : public CMDIChildWnd
{
...
// Implementation
protected:
  static CMenu menu;
  static COLORREF aCustom[16];
  UINT nIDColor;
  COLORREF clrText;
 ...
// Generated message map functions
  //{{AFX_MSG(CChildWnd)
...
  afx_msg void OnCustom();
  afx_msg void OnColor(UINT nID);
  afx_msg void OnUpdateColor(CCmdUI *pCmdUI);
  afx_msg void OnFont();
  //}}AFX_MSG
DECLARE_MESSAGE_MAP()
};

//////////////////////////////////////////////////////////////
// Файл ChildWnd.cpp
```

```
// Copyright (c) 1999 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////////
// В качестве дополнительного примера здесь используются
// макросы для обработки ряда идентификаторов команд
// ON_COMMAND_RANGE и ON_UPDATE_COMMAND_UI_RANGE
BEGIN_MESSAGE_MAP(CChildWnd, CMDIChildWnd)
  //{{AFX_MSG_MAP(CChildWnd)
  ON_WM_PAINT()
  ON_COMMAND(IDM_CUSTOM, OnCustom)
  ON_COMMAND(IDM_FONT, OnFont)
  //}}AFX_MSG_MAP
  ON_COMMAND_RANGE(IDM_BLACK, IDM_WHITE, OnColor)
  ON_UPDATE_COMMAND_UI_RANGE(IDM_BLACK, IDM_WHITE, OnUpdateColor)
END_MESSAGE_MAP()
...
// Соответствующие обработчики
void CChildWnd::OnColor(UINT nID)
{
  nIDColor = nID;
  clrText = colorArray[nIDColor — IDM_BLACK];
  Invalidate();
}
void CChildWnd::OnUpdateColor(CCmdUI* pCmdUI)
{
  pCmdUI->SetCheck(pCmdUI->m_nID == nIDColor);
}
// Обработчик для настройки цветов
void CChildWnd::OnCustom()
{
// Создаем объект "блока диалога"
  CColorDialog clrDlg;

  clrDlg.m_cc.lStructSize = sizeof(clrDlg.m_cc);
  clrDlg.m_cc.rgbResult = colorArray[nIDColor - IDM_BLACK];
  clrDlg.m_cc.lpCustColors = aCustom;

  // Если не используются установки по умолчанию, не забывайте
  // устанавливать флаг CC_ENABLEHOOK для подключения
  // функции обратного вызова, необходимой для обработки
  clrDlg.m_cc.Flags = CC_ENABLEHOOK; // CC_FULLOPEN | CC_RGBINIT
```

```
  // Выводим блок диалога на экран
  if(clrDlg.DoModal() == IDOK)
  {
    // Изменение цвета выводимого текста при закрытии
    // блока диалога кнопкой OK
    nIDColor = IDM_CUSTOM;
    colorArray[nIDColor - IDM_BLACK] = clrDlg.GetColor();
    OnColor(nIDColor);
  }
}
```

Рис. 15.4. Блок диалога для выбора цвета

Рис. 15.5. Расширенный вариант блока диалога для выбора цвета

На рис. 15.4 представлен внешний вид рассматриваемого блока диалога.

При используемых параметрах по умолчанию выбирается черный цвет. Если же при создании использовать флаги CC_FULLOPEN и CC_RGBINIT, то внешний вид существенно изменится (рис. 15.5).

Класс CFontDialog

Этот класс позволяет создать стандартный модальный блок диалога, предназначенный для выбора одного из зарегистрированных в системе шрифтов, а также его стиля, размера, цвета и т. д.

За внешний вид стандартного блока диалога, создаваемого на основе этого класса, отвечает структура типа CHOOSEFONT:

CHOOSEFONT CFontDialog::**m_cf**

```
typedef struct {
  DWORD lStructSize;
  HWND hwndOwner;
  HDC hDC;
  LPLOGFONT lpLogFont;
  int iPointSize;
  DWORD Flags;
  DWORD rgbColors;
  LPARAM lCustData;
  LPCFHOOKPROC lpfnHook;
  LPCTSTR lpTemplateName;
  HINSTANCE hInstance;
  LPTSTR lpszStyle;
  WORD nFontType;
  WORD __MISSING ALIGNMENT__;
  int nSizeMin;
  int nSizeMax;
} CHOOSEFONT;
```

Структура служит для настройки внешнего вида блока диалога и имеет следующие поля:

lStructSize — размер структуры в байтах;

hwndOwner — дескриптор окна, которое владеет блоком диалога, или NULL;

hDC — дескриптор контекста устройства (или информационного контекста) принтера, шрифты которого должны быть отражены в блоке диалога. Это поле используется только в случае, если в поле Flags установлен один из флагов CF_BOTH или CF_PRINTERFONTS;

lpLogFont — указатель на структуру LOGFONT, которая содержит значения, определяющие параметры шрифта. Если при создании блока диалога установлен флаг

CF_INITTOLOGFONTSTRUCT, то параметры этой структуры будут использоваться в качестве начальных. При завершении работы с блоком диалога поля этой структуры будут заполнены параметрами шрифта, выбранного пользователем;

Примечание

Сама структура будет описана ниже при рассмотрении вопросов работы со шрифтами.

iPointSize — определяет размер выбранного шрифта в десятых долях пункта. Поле заполняется после закрытия блока диалога;

rgbColors — цвет символов шрифта, который будет выбран при отображении блока диалога. При этом должен быть установлен флаг CF_EFFECTS;

lCustData — определяет данные, которые приложение может передать функции фильтра lpfnHook. Когда система посылает сообщение WM_INITDIALOG функции фильтра, параметр lParam этого сообщения указывает на структуру CHOOSEFONT, определенную при создании блока диалога. Этот указатель может использоваться для доступа к полю lCustData;

lpfnHook — указатель на функцию фильтра CFHookProc, которая может обрабатывать сообщения блока диалога. Это поле игнорируется, если не установлен флаг CF_ENABLEHOOK;

lpTemplateName — указатель на строку, которая содержит имя ресурса шаблона блока диалога, определенного в модуле hInstance. Для числовых идентификаторов ресурса можно использовать макрос MAKEINTRESOURCE. Это поле игнорируется, если не установлен флаг CF_ENABLETEMPLATE;

hInstance — определяет идентификатор блока памяти (установлен флаг CF_ENABLETEMPLATEHANDLE) или дескриптор модуля (установлен флаг CF_ENABLETEMPLATE), содержащего шаблон блока диалога. Если не установлен ни один из этих флагов, то содержимое поля игнорируется;

lpszStyle — указатель на буфер, содержащий данные стиля. Если установлен флаг CF_USESTYLE, эта строка используется для инициализации стиля шрифта;

__MISSING ALIGNMENT__ — используется для выравнивания;

nSizeMin — определяет минимальный размер шрифта, который пользователь может выбрать. При этом должен быть установлен флаг CF_LIMITSIZE;

nSizeMax — определяет максимальный размер шрифта, который пользователь может выбрать. При этом должен быть установлен флаг CF_LIMITSIZE;

Flags — флаги инициализации, которые также определяют выбор пользователя после закрытия блока диалога. Можно использовать комбинации следующих значений:

CF_APPLY	Предписывает блоку диалога отобразить кнопку **Apply** (применить). Для обработки действия этой кнопки необходимо определить функцию фильтра, которая может обработать сообщение WM_CHOOSEFONT_GETLOGFONT для получения адреса структуры LOGFONT, содержащей текущие параметры шрифта
CF_ANSIONLY	Для Windows 95/98 практически не используется. В списке доступных шрифтов отображаются только шрифты в кодировке ANSI. Для Windows NT определяет, что в списке отображаются все шрифты, которые используют наборы символов Windows и Unicode

CF_BOTH	В списке шрифтов отображаются и экранные, и принтерные шрифты
CF_TTONLY	В списке шрифтов отображаются только масштабируемые шрифты TrueType
CF_EFFECTS	Предписывает блоку диалога отображать элементы управления, с помощью которых можно задавать цвет букв, создавать подчеркнутые и перечеркнутые шрифты. В этом случае необходимо проинициализировать поле rgbColor этой структуры и поля lfStrikeOut и lfUnderLine структуры LOGFONT
CF_ENABLEHOOK	Разрешает использовать функцию фильтра, адрес которой указан в поле lpfnHook
CF_ENABLETEMPLATE	Указывает на то, что поле lpTemplateName содержит указатель на имя ресурса шаблона блока диалога **Font** (шрифт) в модуле, определяемом hInstance
CF_ENABLETEMPLATEHANDLE	Указывает на то, что поле hInstance используется для идентификации блока памяти, содержащего предварительно загруженный шаблон блока диалога **Font**. В этом случае игнорируется содержимое поля lpTemplateName
CF_FIXEDPITCHONLY	Разрешает выбирать только шрифты с фиксированной шириной символов (моноширинные)
CF_FORCEFONTEXIST	Определяет, что если пользователь пытается выбрать несуществующий шрифт, то выдается сообщение об ошибке
CF_INITTOLOGFONTSTRUCT	При инициализации блока диалога используется содержимое структуры LOGFONT, адрес которой передается через поле lpLogFont
CF_LIMITSIZE	При выборе шрифта учитывается содержимое полей nSizeMin и nSizeMax
CF_NOOEMFONT	Запрещает выбирать векторные шрифты. Этот флаг аналогичен флагу CF_VEKTORFONTS
CF_NOFACESEL	При использовании структуры LOGFONT для инициализации блока диалога не отображается имя шрифта
CF_NOSCRIPTSEL	Блокирует раскрывающийся список SCRIPT COMBO BOX (Подчеркивание). Используется только в Windows 95/98 и при инициализации блока диалога
CF_NOSTYLESEL	При использовании структуры LOGFONT для инициализации блока диалога не отображается стиль шрифта.
CF_NOSIZESEL	Интерфейсу GDI запрещается эмуляция шрифтов
CF_VECTORFONTS	Запрещает выбирать векторные шрифты
CF_NOVERTFONTS	Используется только в Windows 95/98 и предписывает блоку диалога отображать только горизонтально ориентированные шрифты
CF_PRINTERFONTS	Предписывает блоку диалога отображать только шрифты, поддерживаемые принтером, контекст которого задан в поле hDC

CF_SCALABLEONLY	Разрешает выбор только масштабируемых шрифтов, которые включают векторные, масштабируемые шрифты принтера, шрифты TrueType
CF_SCREENFONTS	Предписывает блоку диалога отображать только экранные шрифты, поддерживаемые системой
CF_SCRIPTSONLY	Используется только в Windows 95/98 и разрешает выбор всех шрифтов, кроме OEM и Symbol. Используется вместо CF_ANSIONLY
CF_SELECTSCRIPT	Используется только в Windows 95/98 и разрешает отображать только шрифты, характеристики которых определены в поле lfCharSet структуры LOGFONT
CF_SHOWHELP	В блоке диалога отображается кнопка **Help** (**?**)
CF_USESTYLE	Определяет, что поле lpszStyle указывает на буфер, содержащий данные стиля, который отображается при инициализации блока диалога
CF_WYSIWYG	Разрешает выбирать только те шрифты, которые доступны и для отображения на экране, и для печати на принтере. При этом должны быть также установлены флаги CF_BOTH и CF_SCALABLEONLY

nFontType — определяет тип выбираемого шрифта. Можно использовать одно из следующих значений:

BOLD_FONTTYPE	Полужирное начертание. Эта информация продублирована в поле lfWeight структуры LOGFONT и эквивалентна FW_BOLD
ITALIC_FONTTYPE	Курсивное начертание. Эта информация продублирована в поле lfWeight структуры LOGFONT и эквивалентна FW_ITALIC
REGULAR_FONTTYPE	Обычное начертание (не полужирное и не курсивное). Эта информация продублирована в поле lfWeight структуры LOGFONT и эквивалентна FW_REGULAR
PRINTER_FONTTYPE	Принтерный шрифт
SCREEN_FONTTYPE	Экранный шрифт
SIMULATED_FONTTYPE	GDI может эмулировать этот шрифт

Ниже приведено описание основных функций этого класса.

CFontDialog::**CFontDialog**(
LPLOGFONT lplfInitial = NULL,
DWORD dwFlags = CF_EFFECTS | CF_SCREENFONTS,
CDC *pdcPrinter = NULL,
CWnd *pParentWnd = NULL) —

конструктор объекта для стандартного модального блока диалога Windows **Font** (Шрифт). Автоматически записывает значения, принятые по умолчанию, в структуру CHOOSEFONT.

virtual int CFontDialog::**DoModal**( ) —

выводит на экран модальный стандартный блок диалога для выбора шрифта. Если вы хотите самостоятельно настроить параметры блока диалога, устанавливая соответствующие значения в поля структуры m_cf, то делать это следует после создания объекта

"блок диалога", но до вызова функции DoModal. При успешном завершении функция возвращает IDOK или IDCANCEL в зависимости от кнопки, при помощи которой пользователь закрыл блок диалога. В последнем случае можно вызвать функцию Windows CommDlgExtendedError для проверки наличия ошибки. Если функция возвратила IDOK, то можно вызывать другие функции класса для получения информации о выборе пользователя.

void CFontDialog::**GetCurrentFont**(LPLOGFONT lplf) —

заполняет поля структуры LOGFONT характеристиками текущего выбранного шрифта.

CString CFontDialog::**GetFaceName**( ) —

позволяет получить имя выбранного шрифта.

CString CFontDialog::**GetStyleName**( ) —

позволяет получить стиль выбранного шрифта.

int CFontDialog::**GetSize**( ) —

позволяет получить размер (в десятых долях пункта) выбранного шрифта.

COLORREF CFontDialog::**GetColor**( ) —

позволяет получить цвет выбранного шрифта.

int CFontDialog::**GetWeight**( ) —

позволяет получить "вес" выбранного шрифта.

Примечание

Подробнее об этом и других параметрах шрифта читайте в главе 19.

BOOL CFontDialog::**IsStrikeOut**( ) —

определяет, будет ли выбранный шрифт зачеркнутым.

BOOL CFontDialog::**IsUnderLine**( ) —

определяет, будет ли выбранный шрифт подчеркнутым.

BOOL CFontDialog::**IsItalic**( ) —

определяет, будет ли выбранный шрифт наклонным.

Приведенный ниже пример иллюстрирует основные принципы настройки блока диалога этого класса.

(Описание класса CChildWnd см. в разд. "Класс CColorDialog").

```
/////////////////////////////////////////////////////////////
// Файл ChildWnd.cpp
// Copyright (c) 1999 Мешков А., Тихомиров Ю.
/////////////////////////////////////////////////////////////
// Обработчик для настройки блока диалога выбора шрифта
```

```
void CChildWnd::OnFont()
{
  CFontDialog fontDlg;

  fontDlg.m_cf.Flags = CF_BOTH | CF_APPLY | CF_ENABLEHOOK;

  // Инициализация начального цвета
  fontDlg.m_cf.rgbColors = RGB(255, 255, 255);

  // Эти установки становятся доступны только при
  // использовании флага CF_EFFECTS
  fontDlg.m_cf.Flags |= CF_EFFECTS;

  // Выводим блок диалога на экран
  if (fontDlg.DoModal() == IDOK)
  {
    // Выводим имя выбранного шрифта в текущее
    // активное дочернее окно
    CClientDC dc(this);

    // Устанавливаем цвет для выводимого текста
    // в соответствии со сделанным выбором
    dc.SetTextColor(fontDlg.GetColor());
    dc.SetBkColor(::GetSysColor(COLOR_WINDOW));

    // Выводим текст
    TCHAR strText[256];
    strcpy(strText, "Выбран шрифт ");
    strcat(strText, fontDlg.GetFaceName());
    TCHAR str[32];
    sprintf(str, " размером %d пунктов", fontDlg.GetSize()/10);
    strcat(strText, str);
    dc.TextOut(10, 10, strText, strlen(strText));
  }
}
```

Внешний вид блока диалога изображен на рис. 15.6. При выборе какого-либо шрифта его имя и размер отображаются в текущем активном дочернем окне, как показано на рис. 15.7.

Дальнейшие эксперименты по настройке внешнего вида представленных блоков диалога можно провести самостоятельно, наблюдая за изменениями, которые возникают при использовании того или иного параметра.

Рис. 15.6. Блок диалога для настройки шрифта

Рис. 15.7. Результат настройки шрифта

Класс CFindReplaceDialog

Этот класс отвечает за создание двух немодальных блоков диалога: **Find** (найти) для поиска строк и **Find/Replace** (найти/заменить) для поиска и/или замены. Собственно функции поиска и замены вы должны добавить сами.

Как мы уже отмечали, для немодальных блоков диалога лучше использовать динамическое создание объекта при помощи оператора new, вызывая затем функцию Create. Поскольку этот процесс нами уже многократно рассматривался, мы переходим непосредственно к описанию членов этого класса.

FINDREPLACE CFindReplaceDialog::**m_fr** —

структура типа FINDREPLACE, которая содержит характеристики объекта блока диалога:

```
typedef struct {
  DWORD lStructSize;
  HWND hwndOwner;
  HINSTANCE hInstance;
  DWORD Flags;
  LPTSTR lpstrFindWhat;
  LPTSTR lpstrReplaceWith;
  WORD wFindWhatLen;
  WORD wReplaceWithLen;
  LPARAM lCustData;
  LPFRHOOKPROC lpfnHook;
  LPCTSTR lpTemplateName;
 } FINDREPLACE; —
```

Структура служит для настройки внешнего вида блока диалога и имеет следующие поля:

lStructSize — размер структуры в байтах;

hwndOwner — дескриптор окна, которое владеет блоком диалога. Этот параметр не может быть равен NULL;

hInstance — определяет идентификатор блока памяти (установлен флаг FR_ENABLETEMPLATEHANDLE) или дескриптор модуля (установлен флаг FR_ENABLETEMPLATE), содержащего шаблон блока диалога. Если не установлен ни один из этих флагов, то содержимое поля игнорируется;

lpstrFindWhat — указывает на буфер, который содержит строку, введенную пользователем в поле **Find What** (что). Место под буфер (не менее 80 символов) должно быть выделено либо динамически, либо в глобальном или статическом массиве. Если буфер содержит строку при инициализации блока диалога, то она будет отображена в поле **Find What**;

lpstrReplaceWith — указывает на буфер, который содержит строку, введенную пользователем в поле **Replace With** (заменить на). Место под буфер должно быть выделено либо динамически, либо в глобальном или статическом массиве. Если буфер содержит строку при инициализации блока диалога, то она будет отображена в поле **Replace With**;

wFindWhatLen — определяет длину буфера (в байтах), на который указывает поле lpstrFindWhat;

wReplaceWithLen — определяет длину буфера (в байтах), на который указывает поле lpstrReplaceWith;

lCustData — определяет данные, которые система посылает функции фильтра lpfnHook. Когда система посылает сообщение WM_INITDIALOG функции фильтра, параметр lParam этого сообщения указывает на структуру FINDREPLACE, определенную при создании блока диалога и может использоваться для доступа к полю lCustData;

lpfnHook — указатель на функцию фильтра FRHookProc, которая может обрабатывать сообщения блока диалога. Это поле игнорируется, если не установлен флаг FR_ENABLEHOOK;

lpTemplateName — указатель на строку, которая содержит имя ресурса шаблона блока диалога, определенного в модуле hInstance. Для числовых идентификаторов ресурса можно использовать макрос MAKEINTRESOURCE. Это поле игнорируется, если не установлен флаг FR_ENABLETEMPLATE;

Flags — это поле определяет множество битовых флагов, которые можно использовать при инициализации блока диалога. Допустимы комбинации следующих флагов:

FR_DIALOGTERM	Сообщает о закрытии блока диалога
FR_DOWN	При инициализации блока диалога переключатель направления поиска устанавливается в положение **Down** (вниз), определяя, что поиск будет осуществляться от положения курсора до конца документа. В противном случае переключатель устанавливается в положение **Up** (вверх) — поиск осуществляется от позиции курсора к началу документа
FR_ENABLEHOOK	Разрешает использовать функцию фильтра, адрес которой указан в поле lpfnHook. Используется только при инициализации блока диалога
FR_ENABLETEMPLATE	Указывает на то, что поле lpTemplateName содержит указатель на имя ресурса шаблона блока диалога в модуле, определяемом hInstance
FR_ENABLETEMPLATEHANDLE	Указывает на то, что поле hInstance используется для идентификации блока памяти, содержащего предварительно загруженный шаблон блока диалога. В этом случае игнорируется содержимое поля lpTemplateName
FR_FINDNEXT	Указывает, что пользователь нажал кнопку **Next** (следующий) в блоке диалога. При этом поле lpstrFindWhat содержит строку, которую нужно найти
FR_HIDEUPDOWN	Предписывает убрать из блока диалога переключатель направления поиска
FR_HIDEMATCHCASE	Предписывает убрать из блока диалога флажок **Match Case** (учитывать регистр)
FR_HIDEWHOLEWORD	Предписывает убрать из блока диалога флажок **Match Whole Word Only** (только слово целиком)
FR_MATCHCASE	При инициализации устанавливает в блоке диалога флажок **Match Case**, указывая, что при поиске необходимо учитывать регистр
FR_NOMATCHCASE	При инициализации сбрасывает в блоке диалога флажок **Match Case**, указывая, что при поиске не надо учитывать регистр
FR_NOUPDOWN	При инициализации блокирует в блоке диалога переключатель направления поиска
FR_NOWHOLEWORD	При инициализации блокирует флажок **Match Whole Word Only** в блоке диалога

FR_REPLACE	Указывает, что пользователь нажал кнопку **Replace** (Заменить) в блоке диалога. При этом поле lpstrFindWhat указывает на искомую строку, а поле lpstrReplaceWith определяет строку, на которую будет произведена замена
FR_ REPLACEALL	Указывает, что пользователь нажал кнопку **Replace All** (Заменить Все) в блоке диалога. При этом поле lpstrFindWhat указывает на искомую строку, а поле lpstrReplaceWith определяет строку, на которую будет произведена замена
FR_SHOWHELP	В блоке диалога отображается кнопка **Help** (**?**)
FR_WHOLEWORD	При инициализации блока диалога устанавливает флажок **Match Whole Word Only**, указывая, что будет производиться поиск только целых слов. В противном случае можно искать отдельные фрагменты слов

Ниже описаны основные функции класса.

CFindReplaceDialog::**CFindReplaceDialog**( ) —

конструктор класса, который, как и другие немодальные блоки диалога, необходимо создавать динамически.

BOOL CFindReplaceDialog::**Create**(
BOOL bFindDialogOnly,
LPCTSTR lpszFindWhat,
LPCTSTR lpszReplaceWith = NULL,
DWORD dwFlags = FR_DOWN,
CWnd *pParentWnd = NULL) —

создает объект "стандартный блок диалога". Если параметр bFindDialogOnly равен TRUE, создается объект "блок диалога **Find** (Найти)", в противном случае — объект "блок диалога **Find/Replace** (Найти/Заменить)". Остальные параметры вам уже известны из описания полей структуры FINDREPLACE. Если параметр pParentWnd равен NULL, то функция в качестве родительского окна устанавливает главное окно приложения.

BOOL CFindReplace::**FindNext**( ) —

вызывается из вашей функции обратного вызова (callback), если требуется продолжать поиск.

static CFindReplaceDialog* CFindReplaceDialog::**GetNotifier**(
LPARAM lParam) —

позволяет из вашей функции обратного вызова получить указатель на текущий блок диалога **Find/Replace** для получения возможности доступа к его функциям и структуре m_fr.

CString CFindReplaceDialog::**GetFindString**( ) —

возвращает заданную по умолчанию строку для поиска.

CString CFindReplaceDialog::**GetReplaceString**( ) —

возвращает заданную по умолчанию строку, на которую будут заменяться найденные строки.

BOOL CFindReplaceDialog::**IsTerminating**( ) —

при вызове из вашей функции обратного вызова позволяет определить, решил ли пользователь завершить работу с блоком диалога. Если функция возвращает TRUE, то необходимо вызвать DestroyWindow и установить указатель на объект "блок диалога" в NULL. Перед завершением можно сохранить текущие установки блока диалога.

BOOL CFindReplaceDialog::**MatchCase**( ) —

позволяет определить, хочет ли пользователь осуществлять поиск точно, или можно не учитывать регистр.

BOOL CFindReplaceDialog::**MatchWholeWord**( ) —

позволяет определить, хочет ли пользователь осуществлять поиск слова целиком, или только его части.

BOOL CFindReplaceDialog::**ReplaceAll**( ) —

позволяет определить, хочет ли пользователь осуществлять автоматическую замену всех найденных строк.

BOOL CFindReplaceDialog::**ReplaceCurrent**( ) —

позволяет определить, хочет ли пользователь заменить найденную строку.

BOOL CFindReplaceDialog::**SearchDown**( ) —

позволяет определить направление поиска. Если возвращается TRUE, поиск осуществляется к концу документа; в противном случае — к началу.

Подробно создание и работа с блоком диалога этого класса была рассмотрена при описании архитектуры "документ/представление" библиотеки классов MFC.

Класс CPrintDialog

Этот класс отвечает за создание двух модальных блоков диалога: **Print** (печать) и **Print Setup** (настройка печати), с помощью которых пользователь может напечатать документ, выбрать нужный принтер или изменить его параметры. В большинстве случаев достаточно тех функциональных возможностей, которые заложены в этом классе. Если вас не устраивает внешний вид этих стандартных блоков диалога, вы можете образовать из него свой класс для использования собственных шаблонов и функции фильтра для обеспечения дополнительной обработки сообщений.

Настройка объекта этого класса осуществляется с помощью структуры типа PRINTDLG:

PRINTDLG CPrintDialog::**m_pd**

```
typedef struct tagPD{
  DWORD lStructSize;
  HWND hwndOwner;
```

```
  HANDLE hDevMode;
  HANDLE hDevNames;
  HDC hDC;
  DWORD Flags;
  WORD nFromPage;
  WORD nToPage;
  WORD nMinPage;
  WORD nMaxPage;
  WORD nCopies;
  HINSTANCE hInstance;
  DWORD lCustData;
  LPPRINTHOOKPROC lpfnPrintHook;
  LPSETUPHOOKPROC lpfnSetupHook;
  LPCTSTR lpPrintTemplateName;
  LPCTSTR lpSetupTemplateName;
  HANDLE hPrintTemplate;
  HANDLE hSetupTemplate;
} PRINTDLG;
```

Структура служит для настройки внешнего вида блока диалога и содержит поля:

lStructSize — размер структуры в байтах;

hwndOwner — дескриптор окна, которое владеет блоком диалога, или NULL;

hDevMode — определяет идентификатор глобального перемещаемого блока памяти, который содержит структуру типа DEVMODE, используемую для инициализации параметров принтера. Если содержимое этого поля указать как NULL, то функции библиотеки сами запросят глобальную память под эту структуру и, после завершения работы с блоком диалога, запишут туда выбранные параметры принтера. Поля структуры DEVMODE мы опишем позже;

hDevNames — определяет идентификатор глобального перемещаемого блока памяти, который содержит структуру типа DEVNAMES, определяющую имя драйвера принтера, его имя и порт вывода, к которому подключен принтер. Если содержимое этого поля указать как NULL, то функции библиотеки сами запросят глобальную память под эту структуру и, после завершения работы с блоком диалога, запишут туда строки, соответствующие выбранному принтеру. Эти значения приложение использует для создания контекста устройства или информационного контекста. Поля структуры DEVNAMES мы опишем позже;

hDC — определяет идентификатор контекста устройства или информационного контекста, в зависимости от установленного флага PD_RETURNDC или PD_RETURNIC. В первом случае в это поле записывается контекст принтера, который можно использовать для печати, а во втором — информационный контекст, который используется для получения разнообразной информации о принтере;

nFromPage — определяет начальное значение для инициализации элемента управления **From** (от). Используется только в случае, если установлен флаг PD_PAGENUMS. После завершения работы с блоком диалога это поле содержит номер страницы, с которой должна начинаться печать;

nToPage — определяет начальное значение для инициализации элемента управления **To** (до). Используется только в случае, если установлен флаг PD_PAGENUMS. После за-

вершения работы с блоком диалога это поле содержит номер страницы, до которой должна выполняться печать;

nMinPage — определяет минимальное количество страниц, которое можно задать при помощи элементов управления **From** и **To**;

nMaxPage — определяет максимальное количество страниц, которое можно задать при помощи элементов управления **From** и **To**;

nCopies — определяет начальное число копий, записываемое в элемент управления **Copies** (Число копий), если поле hDevMode содержит NULL. В противном случае это число задается в поле dmCopies структуры DEVMODE. При завершении работы с блоком диалога здесь содержится фактическое число копий, которые нужно напечатать;

hInstance — дескриптор модуля, содержащего шаблон блока диалога. Это поле используется только в случае, если установлен хотя бы один из флагов PD_ENABLEPRINTTEMPLATE или PD_ENABLESETUPTEMPLATE;

lCustData — определяет данные, которые приложение может передать функции фильтра lpfnPrintHook или lpfnSetupHook. Когда система посылает сообщение WM_INITDIALOG функции фильтра, параметр lParam этого сообщения указывает на структуру PRINTDLG, определенную при создании блока диалога. Этот указатель может использоваться для доступа к полю lCustData;

lpfnPrintHook — указатель на функцию фильтра PrintHookProc, которая может обрабатывать сообщения блока диалога. Это поле игнорируется, если не установлен флаг PD_ENABLEPRINTHOOK;

lpfnSetupHook — указатель на функцию фильтра SetupHookProc, которая может обрабатывать сообщения блока диалога. Это поле игнорируется, если не установлен флаг PD_ENABLESETUPHOOK.

lpPrintTemplateName — указатель на строку, которая содержит имя ресурса шаблона блока диалога, определенного в модуле hInstance. Это поле игнорируется, если не установлен флаг PD_ENABLEPRINTTEMPLATE.

lpSetupTemplateName — указатель на строку, которая содержит имя ресурса шаблона блока диалога, определенного в модуле hInstance. Это поле игнорируется, если не установлен флаг PD_ENABLESETUPTEMPLATE;

hPrintTemplate — определяет дескриптор блока памяти, содержащий предварительно загруженный шаблон блока диалога **Print** (печать). Для использования этого поля необходимо установить флаг PD_ENABLEPRINTTEMPLATEHANDLE;

hSetupTemplate — определяет дескриптор блока памяти, содержащий предварительно загруженный шаблон блока диалога **Print Setup** (настройка печати). Для использования этого поля необходимо установить флаг PD_ENABLESETUPTEMPLATEHANDLE;

Flags — определяет множество битовых флагов, используемых для инициализации блока диалога **Print**. При завершении работы с блоком диалога это поле отражает выбор пользователя. Допустимыми являются комбинации следующих флагов:

PD_ALLPAGES	Этот флаг используется по умолчанию и устанавливает переключатель **All** (все) во включенное состояние. При этом предполагается, что надо распечатать весь текст, а не отдельные страницы или выделенный фрагмент
PD_COLLATE	При инициализации блока диалога устанавливает переключатель **Collate** (разобрать по копиям) в выбранное состояние. При завершении работы установленный флаг сообщает о том, что драйвер принтера не поддерживает этот режим (значение поля dmCollate структуры DEVMODE не определено), и приложение должно обеспечить его само

PD_DISABLEPRINTTOFILE	Блокирует флажок **Print to File** (печать в файл)
PD_ENABLEPRINTHOOK	Разрешает использовать функцию фильтра, адрес которой указан в поле lpfnPrintHook, для блока диалога **Print**
PD_ENABLEPRINTTEMPLATE	Указывает на то, что поле lpPrintTemplateName содержит указатель на имя ресурса шаблона блока диалога **Print** в модуле, определяемом hInstance
PD_ENABLEPRINTTEMPLATEHANDLE	Указывает на то, что поле hInstance используется для идентификации блока памяти, содержащего предварительно загруженный шаблон блока диалога **Print**. В этом случае игнорируется содержимое поля lpPrintTemplateName
PD_ENABLESETUPHOOK	Разрешает использовать функцию фильтра, адрес которой указан в поле lpfnSetupHook, для блока диалога **Print Setup**
PD_ENABLESETUPTEMPLATE	Указывает на то, что поле lpSetupTemplateName содержит указатель на имя ресурса шаблона блока диалога **Print Setup** в модуле, определяемом hInstance
PD_ENABLESETUPTEMPLATEHANDLE	Указывает на то, что поле hInstance используется для идентификации блока памяти, содержащего предварительно загруженный шаблон блока диалога **Print Setup**. В этом случае игнорируется содержимое поля lpSetupTemplateName
PD_HIDEPRINTTOFILE	Предписывает убрать из блока диалога флажок **Print to File** (печать в файл)
PD_NOPAGENUMS	Блокирует переключатель **Pages** (страницы) и связанный с ним элемент управления
PD_NOSELECTION	Блокирует переключатель **Selection** (выделенный фрагмент)
PD_NOWARNING	Отменяет вывод сообщения о том, что в системе не установлен принтер по умолчанию
PD_PAGENUMS	Устанавливает переключатель **Pages** во включенное состояние. При этом предполагается, что надо распечатать отдельные страницы текста, а не выделенный фрагмент или весь текст
PD_PRINTSETUP	Определяет, что на экран будет выведен блок диалога **Print Setup**, а не блок диалога **Print**
PD_PRINTTOFILE	При инициализации блока диалога устанавливает флажок **Print to File**. При завершении работы с блоком диалога установленный флаг сообщает о том, что поле wOutputOffset структуры DEVNAMES содержит строку "FILE:". Это предписывает подсистеме печати запросить у пользователя имя выходного файла
PD_RETURNDC	Определяет, что в поле hDC будет записан идентификатор контекста принтера

PD_RETURNIC	В поле hDC будет записан идентификатор информационного контекста, который можно использовать для получения информации о принтере
PD_RETURNDEFAULT	В поля hDevNames и hDevMode записываются идентификаторы структур DEVNAMES и DEVMODE, которые заполнены параметрами принтера, установленного по умолчанию. Исходно эти поля должны содержать NULL, в противном случае будет возвращен признак ошибки
PD_SELECTION	Устанавливает переключатель **Selection** в выбранное состояние. При этом предполагается, что надо распечатать выделенный фрагмент, а не весь текст или отдельные страницы
PD_SHOWHELP	В блоке диалога отображается кнопка **Help** (**?**)
PD_USEDEVMODECOPIESANDCOLLATE	Блокирует поле **Copies**, если драйвер принтера не поддерживает печать нескольких копий, и снимает флажок **Collate**, если драйвер не поддерживает этот режим

Чтобы иметь полное представление об установках, необходимых для настройки принтера, приведем краткое описание полей структур DEVMODE и DEVNAMES.

Структура DEVMODE содержит разнообразную информацию о конфигурации и параметрах принтера:

```
typedef struct devicemode{
  BCHAR dmDeviceName[CCHDEVICENAME];
  WORD dmSpecVersion;
  WORD dmDriverVersion;
  WORD dmSize;
  WORD dmDriverExtra;
  DWORD dmFields;
  short dmOrientation;
  short dmPaperSize;
  short dmPaperLength;
  short dmPaperWidth;
  short dmScale;
  short dmCopies;
  short dmDefaultSource;
  short dmPrintQuality;
  short dmColor;
  short dmDuplex;
  short dmYResolution;
```

```
  short dmTTOption;
  short dmCollate;
  BCHAR dmFormName[CCHFORMMNAME];
  WORD dmLogPixels;
  DWORD dmBitsPerPel;
  DWORD dmPelsWidth;
  DWORD dmPelsHeight;
  DWORD dmDisplayFlags;
  DWORD dmDisplayFrequency;
 #if (WINVER >= 0x0400)
  DWORD dmICMMetod;   // только для Windows 95/98
  DWORD dmICMIntent;  // только для Windows 95/98
  DWORD dmMediaType;  // только для Windows 95/98
  DWORD dmDitherType; // только для Windows 95/98
  DWORD dmReserved1;  // только для Windows 95/98
  DWORD dmReserved2;  // только для Windows 95/98
 #endif /* WINVER >=0x0400 */
} DEVMODE;
```

dmDeviceName — имя драйвера принтера;

dmSpecVersion — номер версии структуры DEVMODE;

dmDriverVersion — номер версии драйвера принтера;

dmSize — размер структуры в байтах;

dmDriverExtra — размер в байтах дополнительной структуры данных, которая может находиться в памяти сразу за структурой DEVMODE. Если дополнительные данные не используются, сюда должен быть записан нуль;

dmFields — набор флагов, каждый из которых отвечает за определенное поле структуры DEVMODE. Если флаг установлен, инициализируется соответствующее поле: DM_ORIENTATION, DM_PAPERSIZE, DM_PAPERLENGTH, DM_PAPERWIDTH, DM_COPIES, DM_SCALE, DM_DEFAULTSOURCE, DM_PRINTQUALITY, DM_COLOR, DM_DUPLEX, DM_YRESOLUTION, DM_TTOPTION, DM_COLLATE, DM_FORMNAME, DM_LOGPIXELS, DM_BITSPERPEL, DM_PELSWIDTH, DM_PELSHEIGHT, DM_DISPLAYFLAGS, DM_DISPLAYFREQUENCY, DM_ICMMETHOD, DM_ICMINTENT, DM_MEDIATYPE, DM_DITHERTYPE. Четыре последние флага используются только в Windows 95/98;

dmOrientation — определяет ориентацию бумаги. Может принимать одно из следующих значений: DMORIENT_PORTRAIT или DMORIENT_LANDSCAPE;

dmPaperSize — определяет размер бумаги. Например, для бумаги формата A4 применяется константа DMPAPER_A4. В это поле можно записать нуль, если определены значения обоих полей dmPaperLength и dmPaperWidth;

dmPaperLength — переопределяет длину бумаги, заданную в поле dmPaperSize. Длина задается в десятых долях миллиметра;

dmPaperWidth — переопределяет ширину бумаги, заданную в поле dmPaperSize. Ширина задается в десятых долях миллиметра;

dmScale — определяет коэффициент (в процентах) масштабирования при печати;

dmCopies — определяет число печатаемых копий;

dmDefaultSource — зарезервировано. Должно быть установлено в нуль;

dmPrintQuality — определяет код разрешения принтера: DMRES_HIGH, DMRES_MEDIUM, DMRES_LOW, DMRES_DRAFT или положительное число, равное количеству точек на дюйм;

dmColor — определяет режим печати для цветного принтера: DMCOLOR_COLOR — цветная печать, DMCOLOR_MONOCHROME — черно-белая печать;

dmDuplex — определяет возможность печати с двух сторон листа: DMDUP_SIMPLEX — односторонняя печать, DMDUP_HORIZONTAL — двусторонняя печать по горизонтали, DMDUP_VERTICAL — двусторонняя печать по вертикали;

dmYResolution — определяет вертикальное разрешение принтера в точках на дюйм. Если установлено значение в этом поле, то в поле dmPrintQuality указывается горизонтальное разрешение в точках на дюйм;

dmTTOption — определяет способ печати шрифтов TrueType: DMTT_BITMAP — печать в графическом режиме (используется по умолчанию для матричных принтеров), DMTT_DOWNLOAD — загрузка шрифтов в память принтера (используется по умолчанию для лазерных принтеров), DMTT_SUBDEV — замена шрифтов на принтерные (используется по умолчанию для PostScript-принтеров);

dmCollate — определяет, следует ли упорядочивать копии при печати. Возможные значения: DMCOLLATE_TRUE — сначала печатаются все копии одной страницы; DMCOLLATE_FALSE — сначала печатается весь документ, а потом его копии. Первый режим обеспечивает более быстрый способ печати, т. к. в этом случае данные в принтер посылаются только один раз;

dmFormName — для Windows NT определяет имя используемой формы, например, "Letter" или "Legal". Полный набор допустимых имен можно получить при помощи функции EnumForm. В Windows 95/98 это поле не используется;

dmLogPixels — определяет число пикселей на логический дюйм. Драйверы принтера не используют это значение;

dmBitsPerPel — определяет цветную разрешающую способность, в битах на пиксель. Драйверы принтера не используют это значение;

dmPelsWidth — определяет ширину (в пикселях) видимой поверхности устройства. Драйверы принтера не используют это значение;

dmPelsHeight — определяет высоту (в пикселях) видимой поверхности устройства. Драйверы принтера не используют это значение;

dmDisplayFlags — определяет режим работы дисплея. Драйверы принтера не используют это значение;

dmDisplayFrequency — определяет частоту работы дисплея (в герцах). Драйверы принтера не используют это значение;

Последняя группа полей используется только в Windows 95/98:

dmICMMethod — определяет режим обработки ICM (Image Color Matching). Может принимать одно из следующих предопределенных значений: DMICMMETHOD_NONE — режим ICM заблокирован, DMICMMETHOD_SYSTEM — режим ICM поддерживается системой, DMICMMETHOD_DRIVER — режим ICM поддерживается драйвером устройства, DMICMMETHOD_DEVICE — режим ICM поддерживается адресуемым устройством. Большинство драйверов принтера поддерживают только значения DMICMMETHOD_SYSTEM или DMICMMETHOD_NONE;

dmICMIntent — определяет, какой из трех возможных методов отображения цветов используется по умолчанию. Может принимать одно из следующих предопределенных значений: DMICM_SATURATE — соответствие цвета оптимизируется для насыщенности (деловая графика), DMICM_CONTRAST — соответствие цвета оптимизируется для контраста (сканеры, фотографии), DMICM_COLORMETRIC — соответствие цвета оптимизируется точно по запрошенному цвету;

dmMediaType — определяет тип носителя, на который будет осуществляться печать: DMMEDIA_STANDARD — простая бумага, DMMEDIA_GLOSSY — глянцевая бумага, DMMEDIA_TRANSPARENCY — прозрачная пленка;

dmDitherType — определяет как выполняется случайная добавка: DMDITHER_NONE — добавка отсутствует, DMDITHER_LINEART — специальный метод добавки, при котором проводится четкая граница между черным, белым и серым масштабированием, DMDITHER_COARSE — грубая добавка, DMDITHER_FINE — тонкая добавка, DMDITHER_GRAYSCALE — добавка градаций серого.

Структура DEVNAMES содержит имена драйвера, принтера и порта вывода, к которому подключен принтер:

```
typedef struct tagDEVNAMES{
  WORD wDriverOffset;
  WORD wDeviceOffset;
  WORD wOutputOffset;
  WORD wDefault;
} DEVNAMES;
```

Первые три поля этой структуры содержат смещения текстовых строк с именами соответственно драйвера, принтера и порта вывода. Строки располагаются в памяти непосредственно за этой структурой. Поле wDefault определяет, описывают ли предыдущие поля принтер, установленный по умолчанию. В этом случае оно должно содержать флаг DN_DEFAULTPRN.

Ниже описаны основные функции класса CPrintDialog.

CPrintDialog::**CPrintDialog**(
BOOL bPrintSetupOnly,
DWORD dwFlags = PD_ALLPAGES | PD_USEDEVMODECOPIES |
PD_NOPAGENUM | PD_HIDEPRINTTOFILE |
PD_NOSELECTION,
CWnd *pParentWnd = NULL) —

конструктор объекта для стандартных модальных блоков диалога Windows: **Print** (печать) — bPrintSetupOnly = FALSE; **Print Setup** (настройка печати) — bPrintSetupOnly = TRUE. В первом случае в блоке диалога отображается кнопка **Print Setup**, и к заданным флагам автоматически добавляется флаг PD_RETURNDC.

virtual int CPrintDialog::**DoModal**( ) —

выводит на экран модальный стандартный блок диалога печати. Если вы хотите самостоятельно настроить параметры блока диалога, устанавливая соответствующие значения в поля структуры m_pd, то делать это следует после создания объекта "блок диалога", но до вызова функции. При успешном завершении функция возвращает IDOK или IDCANCEL в зависимости от кнопки, при помощи которой пользователь закрыл блок диалога. В последнем случае можно вызвать функцию Windows CommDlgExtendedError для

проверки наличия ошибки. После вызова этой функции можно вызывать другие функции класса для получения информации о выборе пользователя.

HDC CPrintDialog::**CreatePrinterDC**( ) —

создает и делает текущим контекст устройства принтера на базе структур DEVMODE и DEVNAMES. Использование этой функции не связано с открытием блока диалога.

HDC CPrintDialog::**GetPrinterDC**( ) —

если при создании объекта класса конструктор вызывался с параметром CPrintDialog::bPrintSetupOnly = FALSE, функция возвращает дескриптор контекста устройства принтера. После использования вы должны удалить его, вызвав функцию DeleteDC.

BOOL CPrintDialog::**GetDefaults**( ) —

позволяет получить параметры установленного по умолчанию принтера без открытия блока диалога. Эти параметры записываются в структуру m_pd.

CString CPrintDialog::**GetDeviceName**( ) —

вызывается после DoModal и возвращает имя текущего выбранного принтера.

CString CPrintDialog::**GetDriverName**( ) —

вызывается после DoModal и возвращает имя драйвера текущего выбранного принтера.

CString CPrintDialog::**GetPortName**( ) —

вызывается после DoModal и возвращает имя порта вывода текущего выбранного принтера.

LPDEVMODE CPrintDialog::**GetDevMode**( ) —

вызывается после DoModal и возвращает указатель на структуру типа DEVMODE, которая содержит информацию о конфигурации и параметрах принтера.

int CPrintDialog::**GetCopies**( ) —

вызывается после DoModal и возвращает затребованное число копий, которые нужно напечатать.

int CPrintDialog::**GetFromPage**( ) —

вызывается после DoModal и возвращает номер первой печатаемой страницы.

int CPrintDialog::**GetToPage**( ) —

вызывается после DoModal и возвращает номер последней печатаемой страницы.

BOOL CPrintDialog::**PrintAll**( ) —

вызывается после DoModal и возвращает информацию о том, будут ли печататься все страницы документа (TRUE).

BOOL CPrintDialog::**PrintCollate**( ) —

вызывается после DoModal и возвращает информацию о том, выбрал ли пользователь опцию для печати документа по копиям (TRUE).

BOOL CPrintDialog::**PrintRange**( ) —

вызывается после DoModal и возвращает информацию о том, выбрал ли пользователь опцию для печати определенных страниц документа (TRUE).

BOOL CPrintDialog::**PrintSelection**( ) —

вызывается после DoModal и возвращает информацию о том, выбрал ли пользователь опцию для печати только выделенного фрагмента документа (TRUE).

Подробно создание и работа с блоком диалога этого класса будет рассмотрена в главе 21 при описании вывода документов на печать.

Класс CPageSetupDialog

Этот класс отвечает за создание и функционирование стандартного модального блока диалога **Windows Page Setup** (Параметры страницы) с дополнительной поддержкой установки и модификации полей печати.

Принципы работы с классом такие же, как и для рассмотренных выше стандартных блоков диалога. Но есть одно отличие. После того как пользователь закончил работу с блоком диалога, все изменения, которые он сделал, не будут сохранены. Программист должен позаботиться о сохранении настроек пользователя либо в документе, либо в классе приложения.

Настройка объекта этого класса осуществляется с помощью структуры типа PAGESETUPDLG:

PAGESETUPDLG CPageSetupDialog::**m_psd**

```
typedef struct tagPSD{
 DWORD lStructSize;
 HWND hwndOwner;
 HGLOBAL hDevMode;
 HGLOBAL hDevNames;
 DWORD Flags;
 POINT ptPaperSize;
 RECT rtMinMargin;
 RECT rtMargin;
 HINSTANCE hInstance;
 LPARAM lCustData;
 LPPAGESETUPHOOK lpfnPageSetupHook;
 LPPAGEPAINTHOOK lpfnPagePaintHook;
 LPCTSTR lpPageSetupTemplateName;
 HGLOBAL hPageSetupTemplate;
} PAGESETUPDLG, *LP PAGESETUPDLG;
```

Структура служит для настройки внешнего вида блока диалога.

lStructSize — размер структуры в байтах;

hwndOwner — дескриптор окна, которое владеет блоком диалога, или NULL;

hDevMode — определяет идентификатор глобального блока памяти, который содержит структуру типа DEVMODE, используемую для инициализации элементов управления блока диалога. После завершения работы с блоком диалога, в нее будут записаны значения, выбранные пользователем. В случае, когда выбор пользователя недоступен, в это поле записывается NULL;

Примечание

Описание полей структуры см. в предыдущем разделе при описании класса CPrintDialog.

hDevNames — определяет идентификатор глобального блока памяти, который содержит структуру типа DEVNAMES, определяющую имя драйвера принтера, имя принтера и порт вывода, к которому подключен принтер. Эта структура используется для инициализации элементов управления блока диалога. После завершения работы с блоком диалога в этом поле будет указан дескриптор глобального блока памяти для структуры DEVNAMES, куда записаны строки, определяющие выбор пользователя. В случае, когда выбор пользователя недоступен, в это поле записывается NULL;

ptPaperSize — определяет размеры страницы, установленные пользователем. Флаги PSD_INTHOUSANDTHSOFINCHES и PSD_INHUNDREDTHSOFMILLIMETERS задают единицы измерения;

rtMinMargin — определяет минимально допустимую ширину левого, верхнего, правого и нижнего полей. Значения в этом поле игнорируются, если не установлен флаг PSD_MINMARGINS. Они не должны превышать значения, установленные в поле rtMargin. Единицы измерения задаются флагами PSD_INTHOUSANDTHSOFINCHES и PSD_INHUNDREDTHSOFMILLIMETERS;

rtMargin — определяет ширину левого, верхнего, правого и нижнего полей. Если установлен флаг PSD_MARGINS, то эти значения задают начальные параметры при инициализации блока диалога. При завершении работы с ним в это поле записываются настройки пользователя. Единицы измерения задаются флагами PSD_INTHOUSANDTHSOFINCHES и PSD_INHUNDREDTHSOFMILLIMETERS;

hInstance — дескриптор модуля, содержащего шаблон блока диалога, определенный в поле lpPageSetupTemplateName. Это поле используется только в том случае, если установлен флаг PSD_ENABLEPAGESETUPTEMPLATE;

lCustData — определяет данные, которые приложение может передать функции фильтра lpfnPageSetupHook. Когда система посылает сообщение WM_INITDIALOG функции фильтра, параметр lParam этого сообщения указывает на структуру PAGESETUPDLG, определенную при создании блока диалога. Этот указатель может использоваться для доступа к полю lCustData;

lpfnPageSetupHook — указатель на функцию фильтра PageSetupHook, которая может обрабатывать сообщения блока диалога. Это поле игнорируется, если не установлен флаг PSD_ENABLEPAGESETUPHOOK;

lpfnPagePaintHook — указатель на функцию фильтра PagePaintHook, которая получает сообщения WM_PSD_* из блока диалога, когда нужно перерисовать образец страницы. В процессе обработки функция фильтра может настраивать вид образца. Это поле игнорируется, если не установлен флаг PSD_ENABLEPAGEPAINTHOOK;

lpPageSetupTemplateName — указатель на строку, которая содержит имя ресурса шаблона блока диалога, определенного в модуле hInstance. Для числовых идентификаторов

ресурса можно использовать макрос MAKEINTRESOURCE. Это поле игнорируется, если не установлен флаг PSD_ENABLEPAGESETUPTEMPLATE;

hPageSetupTemplate — определяет дескриптор блока памяти, содержащий шаблон блока диалога **Page Setup** (Параметры страницы). Для использования этого поля необходимо установить флаг PSD_ENABLEPAGESETUPTEMPLATEHANDLE;

Flags — определяет множество битовых флагов, используемых для инициализации блока диалога **Page Setup**. При завершении работы с блоком диалога это поле отражает выбор пользователя. Допустимыми являются комбинации следующих флагов:

PSD_DEFAULTMINMARGINS	Устанавливает минимальные значения, используемые по умолчанию, которые пользователь может определять для полей страницы при печати. Этот флаг игнорируется, если определены флаги PSD_MINMARGINS или PSD_MARGINS
PSD_DISABLEMARGINS	Блокирует элементы управления в блоке диалога, отвечающие за установку полей
PSD_DISABLEORIENTATION	Блокирует элементы управления в блоке диалога, отвечающие за ориентацию страницы
PSD_DISABLEPAGEPAINTING	Блокирует вывод образца страницы в блоке диалога
PSD_DISABLEPAPER	Блокирует элементы управления в блоке диалога, отвечающие за параметры страницы, такие как ее размер и источник
PSD_DISABLEPRINTER	Блокирует кнопку **Printer** (принтер) в блоке диалога
PSD_ENABLEPAGEPAINTHOOK	Разрешает использовать функцию фильтра, адрес которой указан в поле lpfnPagePaintHook
PSD_ENABLEPAGESETUPHOOK	Разрешает использовать функцию фильтра, адрес которой указан в поле lpfnPageSetupHook
PSD_ENABLEPAGESETUPTEMPLATE	Указывает на то, что поле lpPageSetupTemplateName содержит указатель на имя ресурса шаблона блока диалога **Page Setup** в модуле, определяемом hInstance
PSD_ENABLEPAGESETUPTEMPLATEHANDLE	Указывает на то, что поле hPageSetupTemplate используется для идентификации блока памяти, содержащего предварительно загруженный шаблон блока диалога. В этом случае игнорируется содержимое поля lpPageSetupTemplateName
PSD_INHUNDREDTHSOFMILLIMETERS	Указывает, что единицей измерения размеров бумаги и полей являются сотые доли миллиметра (значения в полях rtMargin, rtMinMargin и ptPaperSize)

PSD_INTHOUSANDTHSOFINCHES	Указывает, что единицей измерения размера бумаги и полей являются тысячные доли дюйма (значения в полях rtMargin, rtMinMargin и ptPaperSize)
PSD_MARGINS	Предписывает использовать значения, определенные в поле rtMargin, в качестве начальных для левого, верхнего, правого и нижнего полей. Если этот флаг не установлен, по умолчанию эти размеры инициализируются один дюйм
PSD_MINMARGINS	Предписывает использовать значения, определенные в поле rtMinMargin, в качестве минимально допустимых для левого, верхнего, правого и нижнего полей. Если этот флаг не установлен, минимальные размеры определяются параметрами принтера
PSD_NOWARNING	Отменяет вывод сообщения о том, что в системе не установлен принтер по умолчанию
PSD_RETURNDEFAULT	Определяет, что структуры, на которые указывают поля hDevNames и hDevMode инициализируются параметрами принтера, установленного по умолчанию. Блок диалога на экран не выводится
PSD_SHOWHELP	В блоке диалога отображается кнопка **Help** (**?**)

Ниже описаны основные функции этого класса.

CPageSetupDialog::**CPageSetupDialog**(
DWORD dwFlags = PSD_MARGINS | PSD_INWININIINTLMEASURE, CWnd *pParentWnd = NULL) —

конструктор объекта "блок диалога". Используемый по умолчанию флаг PSD_INWININIINTLMEASURE в настоящее время не реализован, и мы рекомендуем явно задавать единицы измерения, указав один из допустимых флагов: PSD_INHUNDREDTHSOFMILLIMETERS или PSD_INTHOUSANDTHSOFINCHES.

virtual int CPageSetupDialog::**DoModal**( ) —

выводит на экран модальный стандартный блок диалога для настройки полей, размеров и ориентации страницы, а также параметров принтера. Если вы хотите самостоятельно задать параметры инициализации блока диалога, устанавливая соответствующие значения в поля структуры m_psd, то делать это следует после создания объекта "блок диалога", но до вызова функции. При успешном завершении функция возвращает IDOK или IDCANCEL в зависимости от кнопки, при помощи которой пользователь закрыл блок диалога. В последнем случае можно вызвать функцию Windows CommDlgExtendedError для проверки наличия ошибки. Если функция возвратила IDOK, то можно вызывать другие функции класса для получения информации о выборе пользователя.

Для того чтобы зафиксировать текущие установки пользователя, необходимо вызвать функцию:

void CWinApp::**SelectPrinter**(
HANDLE hDevNames,
HANDLE hDevMode,
BOOL bFreeOld = TRUE)

заменяет (bFreeOld = TRUE) текущий принтер на новый, указанный в параметрах hDevNames и hDevMode. Если оба эти параметра равны NULL, то используется принтер по умолчанию. После вызова этой функции приложение получает новый контекст устройства для принтера с соответствующими атрибутами.

HDC CPageSetupDialog::**CreatePrinterDC**( ) —

создает и делает текущим контекст устройства принтера на базе структур DEVMODE и DEVNAMES.

CString CPageSetupDialog::**GetDeviceName**( ) —

вызывается после DoModal и возвращает имя текущего выбранного принтера.

LPDEVMODE CPageSetupDialog::**GetDevMode**( ) —

вызывается после DoModal и возвращает указатель на структуру типа DEVMODE, которая содержит информацию о конфигурации и параметрах принтера.

CString CPageSetupDialog::**GetDriverName**( ) —

вызывается после DoModal и возвращает имя драйвера текущего выбранного принтера.

CString CPageSetupDialog::**GetPortName**( ) —

вызывается после DoModal и возвращает имя порта вывода текущего выбранного принтера.

void CPageSetupDialog::**GetMargins**(
LPRECT lpRectMargins,
LPRECT lpRectMinMargins) —

вызывается после DoModal и возвращает значения размеров полей печати текущего выбранного принтера. Если какой-либо параметр вам не нужен, вместо указателя следует передавать NULL. Размеры возвращаются в сотых долях миллиметра или тысячных долях дюйма, в зависимости от установленных единиц измерения.

CSize CPageSetupDialog::**GetPaperSize**( ) —

вызывается после DoModal и возвращает размер бумаги, выбранной для текущего принтера. Размеры возвращаются в сотых долях миллиметра или тысячных долях дюйма, в зависимости от установленных единиц измерения.

virtual UINT CPageSetupDialog::**OnDrawPage**(
CDC *pDC,
UINT nMessage,
LPRECT lpRect) —

вызывается фреймом и рисует экранный образ страницы печати, который затем отображается в блоке диалога **Page Setup** (параметры страницы); по умолчанию выводит изображение страницы текста. В качестве параметров в функцию передаются: pDC — указатель на контекст устройства принтера, lpRect — указатель на объект CRect или RECT, содержащий координаты перерисовываемой области, nMessage — сообщение, опреде-

ляющее область выводимой страницы. Параметр nMessage может принимать одно из следующих значений:

WM_PSD_FULLPAGERECT	Страница целиком
WM_PSD_MINMARGINRECT	Область, ограниченная текущими минимальными полями
WM_PSD_MARGINRECT	Область, ограниченная текущими полями
WM_PSD_GREEKTEXTRECT	Содержание страницы
WM_PSD_ENVSTAMPRECT	Область, зарезервированная для штампа
WM_PSD_YAFULLPAGERECT	Область, простирающаяся до границ образца страницы

Примечание

Эту функцию следует переопределить, если нужно организовать рисование своего изображения страницы. Не забывайте вызывать функцию базового класса для необрабатываемых вами сообщений.

virtual UINT CPageSetupDialog::**PreDrawPage**(
WORD wPaper,
WORD wFlags,
LPPAGESETUPDLG pPSD) —

вызывается фреймом перед рисованием экранного образа страницы печати. Переопределите эту функцию, если вам нужно организовать рисование собственного изображения страницы. Если переопределенная функция возвращает TRUE, то вы должны сами организовать вывод изображения, а если FALSE, то будет выведено изображение по умолчанию. Параметры: wPaper — определяет размер бумаги, может принимать одно из значений DM_PAPER, перечисленных в описании структуры DEVMODE, pPSD — указатель на структуру PAGESETUPDLG; wFlags — определяет ориентацию бумаги и тип принтера (матричный или Hewlett Packard — HPPCL) и может принимать одно из следующих значений:

0x001	Матричный принтер, горизонтальная ориентация бумаги
0x003	Принтер HPPCL, горизонтальная ориентация бумаги
0x005	Матричный принтер, вертикальная ориентация бумаги
0x007	Принтер HPPCL, вертикальная ориентация бумаги
0x00d	Матричный принтер, горизонтальная ориентация конверта
0x00b	Принтер HPPCL, горизонтальная ориентация конверта
0x01f	Матричный принтер, вертикальная ориентация конверта
0x019	Принтер HPPCL, вертикальная ориентация конверта

Подробно создание и работа с блоком диалога этого класса будет рассмотрена при описании архитектуры "документ/представление" библиотеки классов MFC.

ГРАФИЧЕСКИЙ ИНТЕРФЕЙС УСТРОЙСТВ (GDI)

Глава 16

Классы графического интерфейса

В начале книги мы подробно описали механизм графического вывода. Здесь лишь напомним, что весь вывод в системе Windows реализован на принципе унификации работы с такими физически различными устройствами, как экран дисплея, принтеры, плоттеры и т. п. Для всех этих устройств приложение использует одни и те же функции и, в общем случае, может "не задумываться" над тем, куда будет выведена строка текста или какой-либо графический объект. Система сама распознает устройство вывода и активизирует соответствующий драйвер. Такой подход, с одной стороны, обеспечивает универсальность процесса графического вывода, а с другой — позволяет, например, создавать платы графических акселераторов, которые самостоятельно, без использования центрального процессора осуществляют преобразование команд рисования, существенно разгружая тем самым всю систему в целом.

Для реализации такого подхода в Windows предусмотрен специальный объект, называемый *контекстом устройства*. Именно он хранит необходимую информацию как об устройстве вывода, так и о параметрах собственно рисования.

Win32 API имеет в своем составе большое количество функций графического вывода и настройки режимов рисования, однако их использование требует соблюдения некоторого набора правил, поскольку в противном случае могут возникнуть ошибки времени выполнения, сбои при рисовании, потеря части содержимого окон и постепенное исчерпание системных ресурсов, что может сказаться не только на самом неправильно созданном приложении, но и на других приложениях и всей системе в целом. Последняя неприятность сплошь и рядом случалась в 16-разрядных системах Windows, где не освобожденные в явном виде графические ресурсы не освобождались и после завершения приложения. В 32-разрядных версиях, где более строг учет и распределение ресурсов, такое может произойти в случае продолжения использования неверно написанного приложения. После его заверше-

ния, даже аварийного, система за редким исключением самостоятельно освобождает захваченные ранее ресурсы.

Реализация работы с графикой на языке C++ с использованием классов библиотеки MFC позволяет в большинстве случаев гарантировать правильное использование ресурсов (захват и освобождение) в неявном для разработчика виде. С одной стороны, библиотека обеспечивает программиста не менее мощными средствами графического вывода, но уже в виде функций-членов соответствующих классов, причем имеются и такие, у которых нет аналогов в Win32 API. С другой стороны, библиотечные классы берут на себя большую часть работы по созданию, инициализации и корректному освобождению графических ресурсов, что необходимо для правильной и эффективной реализации графического вывода. Теперь появилась возможность больше сконцентрироваться собственно на самом процессе рисования, а не на подготовке к нему.

Перед тем как перейти к описанию тех возможностей, которые предоставляет библиотека классов MFC, скажем, что для организации графического вывода приложению необходимо работать не только с контекстами устройств, но и с рядом объектов GDI: карандашами, кистями, шрифтами и т. д. Однако функции рисования связаны именно с контекстом устройства.

Контексты устройств

В этом разделе мы проведем краткий обзор классов, инкапсулирующих различные типы контекстов устройств, поддерживаемых системой Windows. Контекст устройства — объект Windows, содержащий информацию о параметрах рисования для некоторого графического устройства, такого как экран дисплея или принтер. Физически весь графический вывод происходит, естественно, на конкретное устройство, представленное в системе контекстом, описываемым, в свою очередь, дескриптором. Последний используют (в качестве параметра) все функции графического вывода Win32 API, т. е. при этом задействуется объект "контекст устройства", до чего MFC-приложению нет никакого дела — оно работает с объектами классов библиотеки MFC, что существенно удобнее и надежнее.

CDC — базовый класс для всех классов, инкапсулирующих контексты устройств Windows. Объекты этого класса используются для работы со всем экраном дисплея или с таким устройством, как принтер. Кроме того, именно указатель на базовый класс передается в такие функции, как, например, CView::OnDraw, хотя может фактически указывать на объекты других производных классов контекстов устройств.

Примечание

Параметр CDC *pDC функции CView::OnDraw на самом деле указывает на объект класса CPaintDC, созданный в функции CView::OnPrepareDC, которая вызывается библиотечными функциями MFC. (В отличие от класса CPaintDC, для объектов класса CMetaFileDC функция OnPrepareDC не вызывается автоматически — за это отвечает разработчик приложения).

CPaintDC — объекты этого класса используются *только* в обработчике сообщения WM_PAINT, генерируемого в ответ на вызов функций UpdateWindow или RedrawWindow. Обработчиком этого сообщения обычно служит переопределяемая функция-член окна CWnd::OnPaint. При создании объекта класса CPaintDC в конструкторе вызывается функция Win32 API BeginPaint, а при разрушении, в деструкторе, — функция EndPaint, также из Win32 API, обеспечивая тем самым необходимые операции для подготовки и завершения процесса графического вывода. За создание объекта класса CPaintDC отвечает разработчик приложения.

CClientDC — объекты этого класса обеспечивают доступ к клиентской части окна. Используется для графического вывода в любой функции, в отличие от объектов класса CPaintDC. При создании объекта класса CClientDC в конструкторе вызывается функция Win32 API GetDC, а при разрушении, в деструкторе, — функция ReleaseDC, также из Win32 API, обеспечивая необходимые операции для подготовки и завершения процесса графического вывода *именно* и *только* в клиентскую часть окна. За создание объекта класса CClientDC отвечает разработчик приложения.

CWindowDC — объекты этого класса обеспечивают доступ ко всему окну. При создании объекта класса CWindowDC в конструкторе вызывается функция Win32 API GetWindowDC, а при разрушении, в деструкторе, — функция ReleaseDC, также из Win32 API, обеспечивая необходимые операции для подготовки и завершения процесса графического вывода, как в *клиентскую*, так и в *неклиентскую* части окна. За создание объекта класса CWindowDC отвечает разработчик приложения.

CMetaFileDC — объекты этого класса обеспечивают доступ к метафайлам Windows. Вызовы функций-членов класса CMetaFileDC записываются в связанном с соответствующим объектом файле. Возможно создание двух типов объектов класса CMetaFileDC. Объекты первого типа поддерживают только функции, осуществляющие исключительно графический вывод. Такие метафайлы являются *контекстно-зависимыми*, т. к. впоследствии можно выводить их содержимое только с учетом настроек контекста того устройства, на которое осуществляется собственно вывод. Метафайлы второго типа (*контекстнонезависимые*) могут содержать, в дополнение к функциям графического вывода, и функции настройки параметров рисования. Естественно, что информация из таких файлов может быть отображена на различных устройствах одинаково.

Примечание

Метафайлы, в отличие от других графических файлов, содержат *команды* рисования. Вывод содержимого файла состоит в выполнении записанных команд. По умолчанию имена метафайлов имеют расширение WMF (Windows MetaFile).

Имена расширенных метафайлов имеют расширение EMF (Enhanced MetaFiles).

Графические объекты

Настройка параметров графического вывода проводится двумя путями. Одни параметры настраиваются при помощи вызова функций Win32 API или функций-членов класса CDC и производных от него, другие — путем замены так называемых *графических объектов*. Библиотека MFC обеспечивает разработчиков всеми необходимыми классами, которые инкапсулируют соответствующие графические объекты Windows. Кроме того, библиотека имеет в своем составе дополнительные классы, значительно облегчающие решение ряда задач (классы CPoint, CSize, CRect, CRectTracker) или унифицирующие работу с графическими объектами как таковыми (CGdiObject). Ниже приведен их краткий обзор.

Примечание

Объекты класса CPoint служат для хранения пары координат (x,y).

Объекты класса CSize служат для хранения размеров, относительного положения или любых других парных значений.

Объекты класса CRect служат для хранения координат прямоугольных областей, обычно левого верхнего и правого нижнего углов.

Объекты класса CrectTracker отображают и предоставляют интерфейс пользователя для перемещения и изменения размеров прямоугольных объектов.

CGdiObject — базовый класс для всех классов, обеспечивающих интерфейс с графическими объектами Windows. Хотя этот класс и не является абстрактным, нет никакого смысла в непосредственном создании его объектов, т. к. он не ассоциирован ни с каким графическим объектом Windows.

CPen — инкапсулирует объект Windows "карандаш", который может быть выбран (назначен) в контекст устройства и использоваться для определения типа и цвета линий или границ фигур.

CBrush — инкапсулирует объект Windows "кисть", который может быть выбран (назначен) в контекст устройства и использоваться для определения типа и цвета заливки внутренних областей замкнутых фигур.

CFont — инкапсулирует объект Windows "шрифт", который может быть выбран (назначен) в контекст устройства и использоваться при операциях вывода текстовой информации.

CBitmap — инкапсулирует объект Windows "битовый массив", который обеспечивает приложение рядом возможностей. Используется, в том числе, для работы с контекстами устройств, расположенными в памяти.

CPalette — инкапсулирует объект Windows "цветовая палитра" (или просто "палитра"), который обеспечивает интерфейс между приложением и цветным устройством вывода, таким как экран дисплея. Этот интерфейс позволяет приложению в полной мере использовать возможности устройства вывода, не нарушая при этом работу (графический вывод) других приложений. "Палитра" служит в Windows для определения используемых приложением цветов из набора доступных в системе.

CRgn — инкапсулирует объект Windows "регион", который хотя и относится к графическим объектам, на самом деле является скорее дополнительным объектом или, точнее, средством управления графическим выводом, определяя сложные (не только прямоугольные) области.

Как и все функции Win32 API, работающие с различными объектами Windows, функции графического вывода используют соответствующие дескрипторы. Функции классов графических объектов и контекстов устройств работают также с дескрипторами. В табл. 16.1 показано соответствие представленных классов библиотеки MFC и типов дескрипторов объектов Windows.

Таблица 16.1

Объект Windows	Класс MFC	Тип дескриптора
"Карандаш"	CPen	HPEN
"Кисть"	CBrush	HBRUSH
"Шрифт"	CFont	HFONT
"Битовый массив"	CBitmap	HBITMAP
"Палитра"	CPalette	HPALETTE
"Регион"	CRgn	HRGN

Помимо того что каждый из вышеперечисленных классов имеет конструктор, в нем определена специальная функция (функции) инициализации, имеющая префикс Create.

Создать объект класса CPen или CBrush можно двумя способами:

- Конструктор используется как для создания собственно объекта, так и для его инициализации
- Помимо использования конструктора, только создающего объект класса, дополнительно требуется вызов инициализирующей функции

Второй способ предпочтительнее, т. к. невозможность инициализации, например, в случае исчерпания соответствующих системных ресурсов, легче определить при вызове именно функции инициализации. В противном случае для гарантии защиты от ошибок, вернее, для их своевременного обнаружения потребуется механизм обработки исключений.

Чтобы настроить параметры рисования с помощью соответствующего графического объекта, необходимо выполнить следующие действия:

- Создать графический объект одним из двух описанных способов
- Заменить в контексте устройства соответствующий текущий графический объект вновь созданным, сохранив при этом указатель на "старый" объект

Примечание

Естественно, что при использовании функций Win32 API приложение манипулирует не указателями, а дескрипторами, не имеющими, в частности, деструктора, и освободить ресурс необходимо в явном виде с применением соответствующей функции API.

- Закончив операции рисования, восстановить графический объект в контексте устройства при помощи сохраненного указателя
- Обеспечить удаление созданного объекта при выходе из области видимости, что происходит автоматически для объектов, созданных в стеке приложения

В случае, если один и тот же объект используется приложением достаточно часто, и его создание/удаление существенно сказывается на производительности приложения в целом, можно создать и хранить такой объект во время его многократного использования. Однако перед завершением работы приложения следует обеспечить его корректное удаление, и, как следствие, освобождение ресурса, т. е. конкретного графического объекта Windows.

В следующем примере иллюстрируется методика создания и использования графических объектов при помощи классов и функций-членов библиотеки MFC.

```
void CSunView::OnDraw(CDC* pDC)
{
  // Создание с одновременной инициализацией
  CPen pen_1(PS_DOT, 1, RGB(0, 0, 0));
  try
  {
    // Создание с одновременной инициализацией
    CBrush brush(HS_CROSS, RGB(0, 0, 0));
    // Создание
    CPen pen_2;
```

```
    // Инициализация
    if (pen_2.CreatePen(PS_SOLID, 2, RGB(192, 192, 192)))
    {
      // Устанавливаем новый карандаш,
      // сохраняя указатель на текущий
      CPen* pOldPen = pDC->SelectObject(&pen_2);
      // Рисуем с использованием нового карандаша
      pDC->Rectangle(...);
      // Восстанавливаем старый карандаш
      pDC->SelectObject(pOldPen);
    }
    else
    {
      // Обработка невозможности инициализации pen_2
    }
  }
  catch (CResourceException *e)
  {
    // Обработка невозможности инициализации brush
  }
}
```

Обратите внимание, что по завершении выполнения функции все созданные объекты классов CBrush и CPen будут автоматически уничтожены, и их деструкторы выполнят действия, необходимые для освобождения захваченных ресурсов. Кроме того, в примере не проводится какая-либо диагностика возможной ошибки при инициализации объекта pen_1, а для объекта brush выполняется проверка возникновения исключительной ситуации, которая возбуждается при невозможности создания объекта и имеет тип указателя на объект CResourceException. Причем при возникновении исключения сразу же после выполнения конструктора управление будет передано на оператор catch, а промежуточные операторы будут проигнорированы.

Теперь, после того как мы обрисовали общую картину графического вывода с точки зрения библиотеки MFC, перейдем к подробному рассмотрению перечисленных классов.

Глава 17

Классы контекстов устройств

Как вы узнали из предыдущей главы, в библиотеке MFC реализованы пять классов контекстов устройств — базовый CDC и четыре производных: CPaintDC, CClientDC, CWindowDC и CMetaFileDC. При этом все основные функции инкапсулированы в базовом классе, оставив на долю остальных фактически один конструктор. Именно по этой причине мы подробно рассмотрим только класс CDC.

Класс CDC

Класс CDC — базовый класс для всех классов, представляющих в библиотеке MFC контексты устройств. Класс CDC обеспечивает разработчика набором функций-членов для работы с такими устройствами, как экран дисплея или принтер, а также позволяющих работать с окнами Windows в целом или только с их клиентской частью. Еще раз подчеркнем — все операции рисования выполняются только для объектов класса CDC. Из всех классов, предоставляющих интерфейс с контекстами устройств, только класс CDC обеспечивает разработчиков всем необходимым для реализации правильного и эффективного графического вывода, остальные классы, в основном отличающиеся конструктором и деструктором, можно считать вспомогательными.

Система Windows имеет в своем составе, пожалуй, только два вида объектов, для взаимодействия с которыми число функций несоизмеримо с количеством функций для всех других объектов. Речь, как вы понимаете, идет об окнах и контекстах устройств. Естественно, что в библиотеке классов MFC реализованы соответствующие классы с не менее широким набором функций-членов. Об окнах и классе CWnd мы подробно уже говорили, поэтому сейчас проведем обзор функциональных возможностей класса CDC.

Создание объекта класса и его связь с контекстом устройства

Все классы библиотеки MFC, непосредственно связанные с объектами Windows, предоставляют программистам более удобный и понятный интер-

фейс. Часто класс имеет в своем составе конструктор по умолчанию, т. е. без параметров, который только создает объект класса, никак не связанный с объектом Windows. Поскольку работа с объектами Windows возможна только через дескрипторы, класс должен иметь возможность для их хранения. Теперь подробнее рассмотрим упомянутые механизмы и поддержку их в реализации класса CDC.

Данные класса

Класс CDC имеет в своем составе даже не один, а два члена данных для хранения дескрипторов, что обеспечивает известную гибкость в различных ситуациях.

HDC CDC::**m_hDC** —

дескриптор контекста устройства графического вывода.

HDC CDC::**m_hAttribDC** —

дескриптор контекста устройства, используется для получения параметров устройства.

По умолчанию m_hDC и m_hAttribDC "равны". В принципе контекст устройства вывода (m_hDC) используется только для выполнения операций рисования, а через второй контекст запрашиваются параметры. Например, функция SetTextColor работает с m_hDC, а GetTextColor — с m_hAttribDC. Однако этот механизм имеет смысл только для объектов класса CMetaFileDC, когда m_hDC настроен на метафайл, а m_hAttribDC настроен на физическое устройство вывода, параметры которого используются для рисования. Этот же механизм используется и для реализации режима *предварительного просмотра печати* (print preview), который реализован в большом количестве приложений. Для этих целей могут быть использованы функции SetOutputDC и SetAttribDC, которые позволяют приложению связывать объект класса с различными контекстами устройств для графического вывода и получения текущих параметров.

В ряде случаев, тем не менее, может возникнуть необходимость получения параметров вывода текста от обоих контекстов. В табл. 17.1 показаны пары таких функций.

Таблица 17.1

m_hAttribDC	m_hDC
GetTextExtent	GetOutputTextExtent
GetTabbedTextExtent	GetOutputTabbedTextExtent
GetTextMetrics	GetOutputTextMetrics
GetCharWidth	GetOutputCharWidth

Конструктор

Создает объект класса, не связывая его с конкретными контекстами устройств. Для дальнейшей работы необходимо осуществить связывание объекта Windows, который, возможно, понадобится еще и создать, с объектом класса.

Инициализация

Функции этой группы либо непосредственно создают контексты для конкретных устройств и инициализируют данные класса соответствующими дескрипторами, либо только осуществляют связывание или, наоборот, разрывают связь с существующими контекстами устройств.

Из таких функций стоит выделить следующие.

Основная функция создания контекста устройства и инициализации объекта класса CDC:

virtual BOOL CDC::**CreateDC**(
  LPCTSTR lpszDriverName,
  LPCTSTR lpszDeviceName,
  LPCTSTR lpszOutput,
  const void* lpInitData) —

> lpszDriverName — указатель на строку, содержащую имя драйвера, например, "EPSON"; lpszDeviceName — указатель на строку, содержащую имя устройства, например, "EPSON FX-1050" (этот параметр используется, если драйвер фактически поддерживает несколько устройств); lpszOutput — указатель на строку, определяющую собственно приемник, — файл, символьное имя устройства или порт вывода; lpInitData — указатель на структуру DEVMODE, содержащую установки устройства, в случае, если значение параметра равно NULL, используются установки, заданные в системе по умолчанию.

Microsoft рекомендует имя устройства завершать символом ":", а имена устройств и портов не должны содержать ни ведущих, ни завершающих пробелов. В случае успешного завершения функция возвращает TRUE, иначе FALSE.

virtual BOOL CDC::**CreateCompatibleDC**(CDC* pDC) —

> создает контекст устройства, расположенный в памяти, совместимый (подобный) заданному параметром pDC. Такие контексты позволяют выполнять операции рисования без непосредственной визуализации. Если pDC = NULL, функция создает контекст устройства, совместимый с экраном дисплея.

В момент создания контекста GDI автоматически устанавливает для него системный монохромный битовый массив размером 1×1 пиксель. Таким образом, операции графического вывода могут производиться только, если будет создан и установлен новый битовый массив для данного контекста устройства.

Функция может создать совместимый контекст устройства только для контекстов, поддерживающих растровые операции. Определить, поддерживает

устройство растровые операции или нет, можно с помощью функции GetDeviceCaps.

Совместимые контексты используются для улучшения характеристик вывода графических объектов на экран дисплея, т. к. позволяют заранее подготовить "картинку" в памяти, а затем ее вывести полностью на экран, для чего можно воспользоваться функцией BitBlt, осуществляющей перенос битовых массивов между контекстами. Чтобы было более понятно, о чем идет речь, проиллюстрируем сказанное на примере.

```
void CMercuryFrame::OnPaint()
{
  CPaintDC dc(this);
  ...
  // Создание совместимого контекста устройства
  CDC dcMem;
  dcMem.CreateCompatibleDC(&dc);
  // Определение размеров клиентской области окна
  CRect rect;
  GetClientRect(&rect);
  // Создание и установка битового массива нужного размера
  // и требуемых свойств
  CBitmap bitmap, *pBitmap;
  bitmap.CreateCompatibleBitmap(&dc, rect.right, rect.bottom);
  pBitmap = dcMem.SelectObject(&bitmap);
  // Вызов функций рисования, например
  dcMem.Rectangle(...)
  // Копирование содержимого одного контекста в другой
  dc.BitBlt(0, 0, rect.right, rect.bottom,
    &dcMem, 0, 0, SRCCOPY);
  // Восстановление первоначально созданного битового массива
  if (pBitmap)
    dcMem.SelectObject(pBitmap);
  // Удаление совместимого контекста устройства
  dcMem.DeleteDC();
}
```

static CDC* PASCAL CDC::**FromHandle**(HDC hDC) —

статическая функция, возвращает указатель на объект класса CDC, связанный с задаваемым параметром hDC контекстом устройства. Обычно возвращаемое значение указывает на временный объект, т. к. функция, скорее всего, используется в нестандартной ситуации. Действительно, если приложение создано с использованием библиотеки классов MFC, то вместо дескрипторов используются все-таки объекты классов.

Поскольку существует вероятность того, что объект может быть временным, его нельзя использовать иначе как немедленно, т. е. в обработчике текущего сообщения. Дело в том, что временные объекты контекстов устройств удаляются автоматически объектом "приложение", созданным на базе CWinApp, в момент *простоя* или, иначе, *отсутствия сообщений* (idle time) посредством вызова другой статической функции DeleteTempMap.

Доступ к объектам рисования

Функции с префиксом GetCurrent возвращают указатели на установленные (текущие) графические объекты — карандаши, кисти, битовые массивы, шрифты и палитры:

CPen* CDC::**GetCurrentPen**()

CBrush* CDC::**GetCurrentBrush**()

CBitmap* CDC::**GetCurrentBitmap**()

CFont* CDC::**GetCurrentFont**()

CPalette* CDC::**GetCurrentPalette**()

Все представленные функции возвращают указатели на временные объекты соответствующих классов.

Функции контекста устройства

Функции этой группы обеспечивают доступ к дескрипторам контекстов (GetSafeHdc), а также сохраняют и восстанавливают параметры контекстов устройств (SaveDC и RestoreDC). К функциям этой группы относится функция GetDeviceCaps, которая может быть использована для получения исчерпывающей информации о самом устройстве и соответствующем драйвере.

int CDC::**GetDeviceCaps**(int nIndex) —

параметр nIndex определяет тип требуемой информации.

Для того чтобы создать действительно универсальное приложение с точки зрения графического вывода на различные устройства, просто необходимо использовать эту функцию. Различие устройств, прежде всего, в предоставлении тех или иных возможностей графического вывода, может потребовать поддержки со стороны GDI или даже самого приложения.

Настройка процесса рисования

Графический вывод можно разделить на две составляющие: вывод текста и вывод графических примитивов. Функции обеих групп используют в качестве параметров (да и то не всегда) только координаты вывода. Все остальные атрибуты графики, такие как цвет, стиль и т. д. задаются отдельно и действуют до их смены. Таким образом, задавать параметры вывода рисования можно для групп последовательно вызываемых функций. Альтернативой

такому способу могло бы быть наличие функций со многими параметрами, что затруднило бы сам процесс программирования. К тому же при желании всегда можно написать свою функцию, которая в зависимости от параметров, во-первых, осуществляет настройку режима рисования, и, во-вторых, производит собственно рисование. Однако в рамках Windows это вряд ли оправданно, т. к. установка ряда параметров рисования осуществляется созданием специальных объектов рисования, максимальное число которых в Windows 95/98 и Windows NT хоть и велико, но все же ограниченно. Создание таких объектов требует определенных дополнительных накладных расходов, возможность же использования текущих настроек для нескольких функций рисования, естественно, такие расходы снижает.

Класс CDC предоставляет программистам следующие группы функций настройки.

Функции средств рисования

Функции этой группы определяют базовые координаты для закрашивания объектов при помощи текущей установленной кисти (GetBrushOrg и SetBrushOrg) и позволяют получить информацию о системных карандашах и кистях (EnumObjects).

Примечание

Поскольку кисти представляют собой прямоугольную область 8×8 пикселей, закрашивание с их помощью областей большего размера требует определенного согласования, т. к. в противном случае пространство не будет закрашено ровно. Например, линии штриховки будут иметь размер не больше 8 пикселей. Для осуществления такого согласования для кистей как раз и определяются базовые координаты.

Установка объектов рисования

Функции этой группы используются для установки объектов рисования, определяющих такие параметры рисования, как цвет и стиль линий, вид закрашивания, используемый шрифт и т. д.

Примечание

Напомним еще раз, что под установкой объекта рисования мы понимаем его выбор в контекст устройства.

Основная функция установки графических объектов — SelectObject. Библиотека MFC в классе CDC определяет пять функций — по одной для каждого из графических объектов:

CPen* CDC::**SelectObject**(CPen* pPen)

CBrush* CDC::**SelectObject** (CBrush* pBrush)

virtual CFont* CDC::**SelectObject** (CFont* pFont)

CBitmap* CDC::**SelectObject** (CBitmap* pBitmap)

int CDC::**SelectObject** (CRgn* pRgn) —

все эти функции, за исключением последней, в случае успешной установки возвращают указатель на графический объект (более точно — на объект класса графического объекта) соответствующего типа, ранее установленный для контекста устройства, и NULL в противном случае. Причем указатель может указывать на временный объект и может использоваться во время обработки одного сообщения Windows. Более подробно о временных объектах рассказано ниже при описании функции FromHandle.

Последняя версия функции SelectObject по своему действию аналогична функции SelectClipRgn.

Битовые массивы можно устанавливать только для контекстов устройств в памяти. Причем один битовый массив не может быть установлен одновременно больше чем для одного такого контекста. Битовый массив может быть либо монохромным, либо совместимым с контекстом устройства, т. е. созданным с помощью функции CBitmap::CreateCompatibleBitmap.

Кроме графических объектов, созданных в приложении, можно использовать и предопределенные системные. Для работы с такими объектами следует использовать функцию SelectStockObject, которая как раз и устанавливает в контексте устройства системные графические объекты следующих различных типов:

virtual CGdiObject* CDC::**SelectStockObject**(int nIndex) —

возвращает указатель на соответствующий использованный ранее объект в случае успешной замены и NULL в противном случае. Новый объект задается параметром nIndex:

BLACK_BRUSH	Черная кисть
DKGRAY_BRUSH	Темно-серая кисть
GRAY_BRUSH	Серая кисть
LTGRAY_BRUSH	Светло-серая кисть
HOLLOW_BRUSH	Пустая или прозрачная кисть
NULL_BRUSH	То же, что и HOLLOW_BRUSH
WHITE_BRUSH	Белая кисть
BLACK_PEN	Черный карандаш
NULL_PEN	Пустой или прозрачный карандаш
WHITE_PEN	Белый карандаш
ANSI_FIXED_FONT	Непропорциональный системный шрифт ANSI
ANSI_VAR_FONT	Пропорциональный системный шрифт ANSI
DEVICE_DEFAULT_FONT	Шрифт, зависящий от устройства

OEM_FIXED_FONT	Непропорциональный шрифт OEM
SYSTEM_FONT	Системный шрифт, используемый по умолчанию для отображения пунктов меню, текста элементов управления и т. д.; в версиях Windows, начиная с 3.0, системный шрифт пропорциональный
SYSTEM_FIXED_FONT	Непропорциональный системный шрифт, используется для совместимости с предыдущими версиями Windows
DEFAULT_PALETTE	Основная цветовая палитра, содержащая 20 цветов

Системные кисти DKGRAY_BRUSH, GRAY_BRUSH и LTGRAY_BRUSH можно использовать только для окон, имеющих стили CS_HREDRAW и CS_VREDRAW. Связано это с тем, что базовые координаты системных кистей нельзя настраивать, и при изменении размеров или положения окон их содержимое может быть испорчено.

Функции настройки цветов и палитр

Функции этой группы позволяют установить новую цветовую палитру (SelectPalette), перерисовать окно, заменяя ранее установленные цвета (UpdateColors), а также создать полутоновую кисть (GetHalftoneBrush), цвет которой получается в результате смешивания текущих цвета фона (background) и цвета, используемого для вывода текстовых строк (foreground).

Режимы рисования

Функции этой группы используются для настройки различных режимов рисования и закрашивания замкнутых областей. Основные функции мы рассмотрим подробнее, т. к. их значение для реализации графического вывода чрезвычайно важно. Все функции реализованы парами: с префиксом Get — позволяет получить текущее значение параметра, с префиксом Set — устанавливает новое значение и возвращает предыдущее. Ниже описаны наиболее часто используемые функции.

int CDC::**SetBkMode**(int nBkMode) —

устанавливает режим закрашивания фона. Возможные режимы задаются параметром nBkMode, который может принимать следующие значения:

OPAQUE	Фон (непрозрачный) закрашивается текущим цветом фона при выводе текста, не сплошных линий или при использовании штриховых кистей; задано по умолчанию
TRANSPARENT	Фон (прозрачный) не меняется

int CDC::**SetPolyFillMode**(int nPolyFillMode) —

устанавливает режим закрашивания многоугольников. Возможные режимы задаются параметром nPolyFillMode (рис. 17.1):

ALTERNATE	Режим закрашивания внутренних областей, находящихся между первой и второй, третьей и четвертой и т. д. линиями контура; задан по умолчанию
WINDING	Режим, при котором закрашиваются все внутренние области

Рис. 17.1. Отличие режимов закрашивания: ALTERNATE — слева, WINDING — справа

Для установки режима рисования или заливки используется функция SetROP2. Название функции расшифровывается как "установка режима растровых (пиксельных) операций, использующих в качестве параметров два пикселя". Такими пикселями являются соответствующие друг другу точки экрана и отображаемого объекта.

int CDC::**SetROP2**(int nDrawMode) —

возможные режимы задаются параметром nDrawMode, который может принимать следующие значения:

R2_BLACK	Пиксели будут иметь черный цвет
R2_WHITE	Пиксели будут иметь белый цвет
R2_NOP	Пиксели останутся без изменений
R2_NOT	Пиксели будут иметь инверсный по отношению к экрану цвет
R2_COPYPEN	Пиксели будут иметь цвет, заданный для текущего карандаша
R2_NOTCOPYPEN	Пиксели будут иметь инверсный цвет по отношению к цвету текущего карандаша
R2_MERGEPENNOT	Цвет пикселей получается в результате комбинирования цвета карандаша и инверсии цвета экрана ("цвет пикселя" = (NOT "цвет пикселя экрана") OR "цвет карандаша")
R2_MASKPENNOT	Цвет пикселей получается в результате комбинирования цвета карандаша и инверсии цвета экрана ("цвет пикселя" = (NOT "цвет пикселя экрана") AND "цвет карандаша")
R2_MERGENOTPEN	Цвет пикселей получается в результате комбинирования цвета экрана и инверсии цвета карандаша ("цвет пикселя" = (NOT "цвет карандаша") OR "цвет пикселя экрана")
R2_MASKNOTPEN	Цвет пикселей получается в результате комбинирования цвета экрана и инверсии цвета карандаша ("цвет пикселя" = (NOT "цвет карандаша") AND "цвет пикселя экрана")
R2_MERGEPEN	Цвет пикселей получается в результате комбинирования цвета экрана и цвета карандаша ("цвет пикселя" = "цвет карандаша" OR "цвет пикселя экрана")

R2_NOTMERGEPEN	Цвет пикселей получается в результате инверсии R2_MERGEPEN цвета ("цвет пикселя" = NOT("цвет карандаша" OR "цвет пикселя экрана"))
R2_MASKPEN	Цвет пикселей получается в результате комбинирования цвета экрана и цвета карандаша ("цвет пикселя" = "цвет карандаша" AND "цвет пикселя экрана")
R2_NOTMASKPEN	Цвет пикселей получается в результате инверсии R2_MASKPEN цвета ("цвет пикселя" = NOT("цвет карандаша" AND "цвет пикселя экрана"))
R2_XORPEN	Цвет пикселей получается в результате комбинирования цвета экрана и цвета карандаша ("цвет пикселя" = "цвет карандаша" XOR "цвет пикселя экрана")
R2_NOTXORPEN	Цвет пикселей получается в результате инверсии R2_XORPEN цвета ("цвет пикселя" = NOT("цвет карандаша" XOR "цвет пикселя экрана"))

Примечание

Вышеперечисленные режимы рисования линий поддерживаются только растровыми устройствами.

Для выбора режима сжатия битовых массивов в классе реализована функция SetStrechBltMode:

int CDC::**SetStretchBltMode**(int nStretchMode) —

задает режим сжатия/растяжения изображений при использовании функции StretchBlt, в результате выполнения которой возможна потеря фрагментов изображения или их искажение. Возможные режимы задаются параметром nStretchMode, который может принимать следующие значения:

STRETCH_ANDSCANS или BLACKONWHITE	Цвет пикселей получается в результате комбинирования цвета удаляемых пикселей и остающихся с использованием операции AND; для монохромных изображений черные пиксели имеют приоритет над белыми
STRETCH_DELETESCANS или COLORONCOLOR	Режим простого удаления пикселей без проведения попыток сохранить информацию об их цвете
STRETCH_HALFTONE или HALFTONE	При сжатии цвет пикселей результирующего блока "в среднем" сохраняется
STRETCH_ORSCANS или WHITEONBLACK	Цвет пикселей получается в результате комбинирования цвета удаляемых пикселей и остающихся с использованием операции OR; для монохромных изображений белые пиксели имеют приоритет над черными

К функциям этой же группы относятся функции для назначения и запроса цветов. Одни используются для цвета символов текста (функции SetTextColor и GetTextColor), а другие — для цвета фона (функции SetBkColor и GetBkColor).

Примечание

Цвет для символов текста используется только в случае, если текстовая строка выводится при помощи функций CDC::TextOut или CDC::ExtTextOut.

Цвет фона используется для заливки промежутков несплошных линий, областей, закрашенных штрихованными кистями, а также в просветах между символами текста. Однако для линий, выведенных с использованием карандашей, созданных с флагами PS_GEOMETRIC или PS_COSMETIC, цвет фона не используется.

Режимы отображения

До сих пор, говоря о задании координат, мы подразумевали, что речь идет о смещении относительно левого верхнего угла экрана (screen coordinates) или левого верхнего угла клиентской области окна (client coordinates). В принципе для всего, что касается окон в целом и оконных сообщений, например WM_MOVE, WM_MOUSEMOVE, это правильно. Однако всегда ли удобно рисовать линию "по точкам", когда, во-первых, все "перевернуто с ног на голову" — координаты по вертикали растут вниз, во-вторых, приходится постоянно заботиться о масштабе, и, в-третьих, приходится помнить формулы вычисления координат точек при повороте и смещении, учитывая недостатки такой системы координат?

ОС Windows приходит на помощь, предоставляя программисту возможность создания своей системы координат. Для этого можно, манипулируя различными параметрами, задавать направление и масштаб осей, а также начало координат.

Настройка режимов отображения

Функции этой группы устанавливают и настраивают систему координат, которая используется всеми функциями вывода — координаты вывода задаются именно в логических единицах. Кроме единиц по осям X и Y, функции этой группы задают направление осей и начало координат. Практически все функции группы реализованы парами. Функции с префиксом Get позволяют получить текущее значение параметра, а с префиксом Set устанавливают новое значение и возвращают предыдущее. По умолчанию начало координат находится в левом верхнем углу области вывода, (прежде всего окна). При этом ось X направлена слева направо, а ось Y — сверху вниз. Единица измерения — один пиксель (режим MM_TEXT). Кстати сказать, именно поэтому на различных устройствах, вследствие различий в размерах пикселя, фактический (визуальный) размер по оси X может не соответствовать размеру по оси Y.

Для настройки системы координат в Windows используются два понятия — *физическая область вывода* (viewport), координаты и размеры которой зада-

ются в физических единицах — пикселях, и *логическая область вывода* (window), координаты и размеры которой задаются в логических единицах, определяемых режимом рисования.

Теперь давайте ознакомимся с режимами отображения, которые поддерживает Windows. Всего их восемь, и основная функция, которая позволяет выбрать режим, а значит, и систему координат, — SetMapMode:

virtual int CDC::**SetMapMode**(int nMapMode) —

устанавливает направления осей и определяет логические единицы, т. е. единицы измерения для задаваемой системы координат. Возможные режимы задаются параметром nMapMode, который может принимать следующие значения:

MM_TEXT	Одна логическая единица равна одному пикселю, ось X направлена вправо, ось Y направлена вниз; режим задан по умолчанию
MM_HIENGLISH	Одна логическая единица равна 0,001 дюйма, ось X направлена вправо, ось Y направлена вверх
MM_HIMETRIC	Одна логическая единица равна 0,01 миллиметра, ось X направлена вправо, ось Y направлена вверх
MM_LOENGLISH	Одна логическая единица равна 0,01 дюйма, ось X направлена вправо, ось Y направлена вверх
MM_LOMETRIC	Одна логическая единица равна 0,1 миллиметра, ось X направлена вправо, ось Y направлена вверх
MM_TWIPS	Одна логическая единица — твипс (twips) (1/20 пункта или 1/1440 дюйма), ось X направлена вправо, ось Y направлена вверх
MM_ANISOTROPIC	Режим позволяет настраивать с помощью функций SetWindowExt и SetViewportExt размерность (отдельно для каждой) осей, их направления и начало отсчета
MM_ISOTROPIC	Режим позволяет настраивать с помощью функций SetWindowExt и SetViewportExt размерность осей, их направления и начало отсчета, однако единица оси X равна единице оси Y, таким образом, при необходимости GDI настраивает размерность по осям, обеспечивая соотношение 1:1

Примечание

Режим MM_TEXT назван так не потому, что он наиболее удобен для вывода текста, а потому, что направление осей — суть его системы координат — соответствует способу чтения, принятому в европейских странах — слева направо и сверху вниз.

Примечание

Исторически в полиграфии используются такие единицы измерения, как *пункты* (points) и *твипсы* (twips), размеры которых соответственно равны (приблизительно) 1/72 и 1/1440 дюйма.

Отметим также, что только режимы MM_HIENGLISH, MM_HIMETRIC, MM_LOENGLISH, MM_LOMETRIC и MM_TWIPS позволяют оперировать в приложениях физически понятными единицами измерения с одной стороны, а с другой — определенной независимостью от устройств вывода.

Для первых шести режимов можно изменять только начало координат, которое по умолчанию соответствует левому верхнему углу области вывода. Для режимов MM_ANISOTROPIC и MM_ISOTROPIC дополнительно можно настраивать направление осей и их масштаб.

И физическая и логическая области вывода характеризуются точкой, определяющей начало координат. Для настройки начала координат физической области вывода используются функции:

virtual CPoint CDC::**SetViewportOrg**(
    int x,
    int y)

или

virtual CPoint CDC::**SetViewportOrg**(POINT point)

Для настройки начала координат логической области вывода используются следующие функции:

CPoint CDC::**SetWindowOrg**(
    int x,
    int y)

или

CPoint CDC::**SetWindowOrg**(POINT point)

Все четыре функции в качестве параметров получают новые значения для установки начала координат, а возвращают предыдущие значения. Здесь есть одна тонкость: если для физической области вывода указываются координаты, которые в дальнейшем будут являться началом координат, то для логической области вывода указываются *логические* координаты, которые в дальнейшем будут соответствовать левому верхнему углу логической области вывода.

Из вышесказанного следует, что есть смысл настраивать начало координатной сетки только для одной из областей, т. к. система координат логической области вывода относительна системе координат физической области. Теперь, как обычно, поясним сказанное на примерах.

```
void CVenusFrame::OnPaint()
{
  CPaintDC dc(this);

  CRect rect;
  GetClientRect(rect);
  dc.SetViewportOrg(rect.Width() / 2, rect.Height() / 2);
```

```
  ...
}

void CEarthFrame::OnPaint()
{
  CPaintDC dc(this);

  CRect rect;
  GetClientRect(rect);

  dc.DPtoLP(rect);
  dc.SetWindowOrg(-rect.Width() / 2, -rect.Height() / 2);

  ...
}
```

После таких настроек функции OnPaint классов CVenusFrame и CEarthFrame будут работать с координатной сеткой, начало координат которой располагается в середине клиентской области (рис. 17.2).

Рис. 17.2. Перемещение начала координат в середину клиентской области окна

В следующем примере попытка настроить системы координат обеих областей вывода в классе CMarsFrame приводит к несколько неожиданным результатам, так что, скорее всего, такую последовательность действий можно считать ошибочной (рис. 17.3).

```
void CMarsFrame::OnPaint()
{
  CPaintDC dc(this);

  CRect rect;
  GetClientRect(rect);

  dc.SetViewportOrg(rect.Width() / 2, rect.Height() / 2);
  //
```

```
  // Вызовы следующих двух функций необходимы в случае,
  // если текущий режим отличен от MM_TEXT (см. далее),
  // т. к. функция SetWindowOrg работает с логическими
  // координатами, а в результате преобразования из-за
  // несоответствия направления осей длина и ширина области
  // могут быть вычислены неверно
  //
  dc.DPtoLP(rect);
  rect.NormalizeRect();

  dc.SetWindowOrg(-rect.Width() / 2, -rect.Height() / 2);

  ...
}
```

Рис. 17.3. Перемещение начала координат в правый нижний угол клиентской области окна

Кроме начала координат, каждая из областей вывода характеризуется еще, что вполне естественно, *размерами* (extents), которые используются *только* для определения коэффициента масштабирования. Коэффициент для каждой из осей ищется как отношение соответствующих размеров областей вывода. Однако для большинства режимов коэффициент масштабирования изменить невозможно.

Примечание

Не следует забывать, что размеры области, куда будет осуществляться вывод, определяются размерами клиентской области окна.

Для настройки размеров физической области вывода используются функции:

virtual CSize CDC::**SetViewportExt**(
    int cx,
    int cy)

или

virtual CSize CDC::**SetViewportExt**(SIZE size)

Для настройки размеров логической области вывода используются функции:

virtual CSize CDC::**SetWindowExt**(
    int cx,
    int cy)

или

virtual CSize CDC::**SetWindowExt**(SIZE size)

Все четыре функции в качестве параметров получают новые значения-размеры по осям, которые используются для преобразования логических координат в физические. Кроме того, настройка размеров для физического окна вывода позволяет управлять направлением осей. Поскольку после выбора одного из режимов ось X направлена слева направо, а ось Y — сверху вниз, для изменения направления, например, оси Y, размер должен быть задан с отрицательным значением. Все функции возвращают предыдущие значения, причем они могут отличаться от тех, что были фактически заданы соответствующими функциями, т. к. Windows настраивает их значения в зависимости от конкретных возможностей и/или типа устройства.

Ниже приведен фрагмент кода, который помещает начало координат в середину клиентской области окна, ось Y направлена вверх, а минимальный из размеров по горизонтали или вертикали равен 200 логическим единицам (рис. 17.4). Причем вывод может осуществляться и по координатам, большим 100 или меньшим −100 соответственно.

Примечание

Режим MM_ISOTROPIC предполагает соотношение единиц по обоим направлениям как 1:1. Таким образом, задание одинаковых размеров области вывода в окне, имеющем прямоугольную, а не квадратную, форму, приводит к тому, что по одной из осей размер области вывода фактически будет больше заданного. С другой стороны, никто не запрещает вызывать функции рисования вне клиентской области окна — это не окажет никакого влияния на другие окна.

```
void CSaturnFrame::OnPaint()
{
  CPaintDC dc(this);

  CRect rect;
  GetClientRect(rect);

  dc.SetMapMode(MM_ISOTROPIC);
  dc.SetWindowExt(100, 100);
  dc.SetViewportExt(rect.Width(), -rect.Height());
  dc.SetViewportOrg(rect.Width() / 2, rect.Height() / 2);

  ...
}
```

Рис. 17.4. Перемещение начала координат в середину клиентской области окна и изменение направления оси y

Кроме вышеперечисленных функций установки начала координат, размеров и направлений осей, класс CDC содержит ряд функций для настройки текущих установок.

Преобразование координат

В описании большинства функций рисования указано, что координаты задаются в логических единицах, величина которых зависит от установленного режима. Однако ряд функций "понимают" только физические единицы устройства — пиксели. Для таких функций необходимо осуществлять преобразование координат.

Функции этой группы предназначены для преобразования логических координат в физические и наоборот (LPtoDP и DPtoLP). Кроме того, четыре функции DPtoHIMETRIC, LPtoHIMETRIC, HIMETRICtoDP и HIMETRICtoLP служат для преобразования координат при передаче размеров объекта OLE и при их получении.

Функции рисования

Итак, параметры заданы, объекты рисования выбраны, режимы установлены — можно рисовать. Давайте рассмотрим, какие возможности предоставляет нам класс CDC для рисования различных графических объектов.

Отображение регионов

Функции этой группы позволяют рисовать рамки (FrameRgn), закрашивать (FillRgn и PaintRgn) и инвертировать цвета регионов (InvertRgn).

Отсечения

Функции этой группы добавляют/исключают области отсечения, которые используются для ограничения области перерисовки (SelectClipRgn, ExcludeClipRect, ExcludeUpdateRgn и IntersectClipRect). Кроме того, исполь-

зование части функций группы, например, GetClipBox и PtVisible, позволяет определять текущую область отсечения для программного исключения перерисовки части рисунка, что в ряде случаев существенно сокращает время вывода. Для повторного вывода сложного объекта могут потребоваться не только многократные вызовы функций настройки различных параметров и функций собственно вывода, но и длительный по времени процесс вычисления, например, для вывода контуров трехмерных объектов.

Рисование линий

Функции этой группы предназначены для рисования прямых (LineTo), гладких (Arc, ArcTo и AngleArc) и ломаных линий (Polyline, PolylineTo, PolyPolyline, PolyBezier, PolyBezierTo и PolyDraw), установленным на момент рисования карандашом. Часть функций меняют при этом текущее положение карандаша (LineTo, ArcTo, PolylineTo и PolyBezierTo), другие — нет. Даже если в результате рисования контур получается замкнутым, закрашивание не осуществляется.

Примечание

На сегодняшний день функции ArcTo, AngleArc, PolyDraw поддерживаются все еще только в рамках Windows NT.

Базовые функции рисования

Функции этой группы осуществляют рисование закрашенных простейших объектов, прежде всего, таких как прямоугольники (FillRect, FillSolidRect и InvertRect), с возможностью задания различного типов рамок (FrameRect, Draw3dRect и DrawEdge). К группе относятся также функции рисования пиктограмм и битовых массивов (DrawIcon и DrawState) и элементов управления — кнопок, меню, заголовков окон и полос прокрутки (DrawFrame Control). Большинство функций используют для рисования текущие карандаш и кисть.

Остановимся на функции, вернее, на функциях, DrawState подробнее. Вот определения всех версий функции.

Отображение битовых массивов

```
BOOL CDC::DrawState(
    CPoint pt,
    CSize size,
    HBITMAP hBitmap,
    UINT nFlags,
    HBRUSH hBrush = NULL)
```

```
BOOL CDC::DrawState(
    CPoint pt,
    CSize size,
    CBitmap* pBitmap,
    UINT nFlags,
    CBrush* pBrush = NULL)
```

Отображение значков

```
BOOL CDC::DrawState(
    CPoint pt,
    CSize size,
    HICON hIcon,
    UINT nFlags,
    HBRUSH hBrush = NULL)
```

```
BOOL CDC::DrawState(
    CPoint pt,
    CSize size,
    HICON hIcon,
    UINT nFlags,
    CBrush* pBrush = NULL)
```

Отображение строк

```
BOOL CDC::DrawState(
    CPoint pt,
    CSize size,
    LPCTSTR lpszText,
    UINT nFlags,
    BOOL bPrefixText = TRUE,
    int nTextLen = 0,
    HBRUSH hBrush = NULL)
```

```
BOOL CDC::DrawState(
    CPoint pt,
    CSize size,
    LPCTSTR lpszText,
    UINT nFlags,
    BOOL bPrefixText = TRUE,
    int nTextLen = 0,
    CBrush* pBrush = NULL)
```

"Управляемый вывод" изображений

```
BOOL CDC::DrawState(
    CPoint pt,
    CSize size,
    DRAWSTATEPROC lpDrawProc,
    LPARAM lData,
    UINT nFlags,
    HBRUSH hBrush = NULL)
```

```
BOOL CDC::DrawState(
    CPoint pt,
    CSize size,
    DRAWSTATEPROC lpDrawProc,
    LPARAM lData,
    UINT nFlags,
    CBrush* pBrush = NULL)
```

Общие параметры функций DrawState

Первые два параметра (общие для всех функций) определяют точку привязки (левый верхний угол прямоугольной области вывода — параметр pt) и размеры области вывода (параметр size).

Параметр nFlags определяет тип изображения и может принимать одно из следующих значений:

DST_BITMAP	Битовый массив; в последних двух версиях функции младшее слово параметра lData содержит дескриптор битового массива
DST_COMPLEX	Пользовательский тип изображения; для вывода изображения используется функция обратного вызова (callback), заданная параметром lpOutputFunc
DST_ICON	Значок; в последних двух версиях функции младшее слово параметра lData содержит дескриптор значка

DST_PREFIXTEXT	Символьная строка; если текст содержит символ "&", при выводе следующий за ним символ выводится с подчеркиванием, символ "&" не выводится; в последних двух версиях функции параметр lData содержит адрес символьной строки, а параметр wData содержит ее длину (количество символов), если этот параметр равен 0, то для определения длины строки она должна завершаться нуль-символом
DST_TEXT	Символьная строка; в последних двух версиях параметр lData содержит адрес символьной строки, а параметр wData содержит ее длину (количество символов), если этот параметр равен 0, то для определения длины строки она должна завершаться нуль-символом

Из названия функции не совсем ясно, причем тут "состояние" (state). Возможно, кое-что прояснится после знакомства с параметром nFlags, который может принимать следующие значения, задающие вид модификации изображения:

DSS_NORMAL	Модификации отсутствуют
DSS_UNION	Объединение части изображения с фоном, т. е. частичное разрушение (dithering) изображения (например, строки выводятся серым цветом)
DSS_DISABLED	Выдавливание (embossing) изображения — эффект рельефной каменной плиты
DSS_DEFAULT	Вывод изображения в более отчетливой (контрастной) форме
DSS_MONO	Область вывода заполняется при помощи заданной кисти (параметры hBrush или pBrush)

Примечание

В данном случае состояние DSS_DEFAULT названо так не с точки зрения программирования: по умолчанию используется флаг DSS_NORMAL, значение которого равно нулю и, следовательно может в явном виде не задаваться.

Для всех состояний, кроме DSS_NORMAL, изображение конвертируется в монохромное перед тем, как к нему применяются эффекты.

Отображение эллипсов и многоугольников

Функции этой группы применяются для рисования сложных фигур, таких как эллипсы (Ellipse), сегменты (Chord), сектора (Pie), многоугольники (Rectangle, RoundRect, Polygon и PolyPolygon) и т. д. Функции используют для рисования текущие карандаш и кисть. В случае необходимости выделения прямоугольной области можно использовать функцию DrawFocusRect, которая выполняет над содержимым области операцию XOR.

Контуры

Контур представляет собой совокупность прямых и изогнутых линий, которая хранится в системе и может быть использована для создания сложных графических объектов. В то же время контур может быть использован для создания регионов, например, для ограничения области отсечения.

Для того чтобы создать контур, необходимо сообщить системе о том, что все последующие вызовы функций рисования должны быть использованы для создания контура. Для этого необходимо вызвать функцию CDC::BeginPath, а по завершении создания контура — CDC::EndPath. Между вызовами этих двух функций реальный вывод не осуществляется. Для того чтобы обвести или закрасить контур, используются функции CDC::StokePath и CDC::FillPath, а чтобы обвести и одновременно закрасить замкнутый контур, необходимо воспользоваться функцией CDC::StrokeAndFillPath.

Для создания контура может использоваться только часть функций рисования:

AngleArc	Arc	ArcTo
Chord	CloseFigure	Ellipse
ExtTextOut	LineTo	MoveTo
Pie	PolyBezier	PolyBezierTo
PolyDraw	Polygon	Polyline
PolylineTo	PolyPolygon	PolyPolyline
Rectangle	RoundRect	TextOut

К сожалению, не все эти функции влияют на содержимое контура так, как хотелось бы, — кроме функций вывода текста, это только функции рисования линий, ломаных и кривых Безье. Связано это с тем, что контур хранит только координаты вершин составляющих его линий. Например, дуга при прорисовке контура будет заменена прямой линией. Кривые Безье, по сути, являются совокупностями небольших прямых линий или, иначе, ломаными.

Приведем пример создания и использования контуров для закраски символов текста, что может оказаться весьма полезно, т. к. существенно упрощает сложные операции, реализованные только в мощных графических редакторах.

```
void CClipPathFrame::OnPaint()
{
  CPaintDC dc(this);
  CFont font;
  font.CreateFont(80, 30, 0, 0, FW_BOLD, 0, 0, 0, DEFAULT_CHARSET,
    OUT_DEFAULT_PRECIS, CLIP_DEFAULT_PRECIS, PROOF_QUALITY,
    VARIABLE_PITCH | FF_DONTCARE, "Arial");
```

```
  // Назначаем текущим ранее подготовленный шрифт
  CFont *pFont = dc.SelectObject(&font);
  // Поскольку строка нам понадобится несколько раз,
  // создадим объект типа CString, что позволит,
  // кроме всего прочего, использовать версии
  // функций с меньшим количеством параметров
  CString csText = _T("SelectClipPath");
  // Узнаем размеры строки, которая будет
  // выведена с использованием текущего шрифта
  CSize sz = dc.GetTextExtent(csText);
  // Сообщаем системе о создании контура
  dc.BeginPath();
  // Строка не будет выведена на экран -
  // на основе ее очертания будет создан контур
  dc.TextOut(10, 10, csText);
  // Завершаем создание контура
  dc.EndPath();
  // На основе контура создаем и устанавливаем
  // область отсечения
  dc.SelectClipPath(RGN_XOR);
  // Выводим горизонтальные линии с изменяемым
  // расстоянием между ними
  double delta = 2.0;
  for (int i = 0; i < sz.cy; i += (int)delta)
  {
    dc.MoveTo(10, 10+i);
    dc.LineTo(10+sz.cx, 10+i);
    delta *= 1.025;
  }

  // Устанавливаем ранее использованный шрифт
  dc.SelectObject(pFont);
  // Сообщаем системе о том, что далее областью
  // отсечения будет вся клиентская область окна
  dc.SelectClipRgn(NULL);
}
```

В результате выполнения этого кода мы получим строку, заливка символов которой не может быть получена обычными средствами (рис. 17.5).

Рис. 17.5. Результат работы функции CClipPathFrame::OnPaint

Несколько подробнее опишем функцию CDC::SelectClipPath, использованную в примере:

BOOL CDC::**SelectClipPath**(int nMode) —

параметр nMode задает один из пяти режимов создания новой области отсечения на основе текущей области и текущего контура. Далее перечислены возможные режимы (они также используются и в функции создания новой области отсечения с использованием регионов CDC::SelectClipRgn):

RGN_AND	Новая область отсечения включает пересечение текущей области и текущего контура
RGN_COPY	Новая область отсечения совпадает с текущим контуром
RGN_DIFF	Новая область отсечения состоит из текущей, исключая пересечение с текущим контуром
RGN_OR	Новая область отсечения является объединением текущей области и текущего контура
RGN_XOR	Новая область отсечения является объединением текущей области и текущего контура, исключая область пересечения

Отображение битовых массивов

Если можно было бы рисовать только математически заданные фигуры, графические возможности Windows были бы существенно ограниченны (конечно, реализации отображения и копирования только растровых (точечных) изображений тоже недостаточно). Система Windows, и вместе с ней класс CDC, предоставляют набор достаточно мощных функций, которые позволяют выводить рисунки, дополнительно масштабируя их, или осуществляя их вращение. К сожалению, последняя возможность реализована только в рамках Windows NT (функция PlgBlt).

Функции, использующие битовые массивы

Функции этой группы используются для отображения отдельных точек (GetPixel, SetPixel и SetPixelV) или массивов точек, задаваемых битовыми массивами. Отдельно отметим функции копирования и преобразования битовых массивов, а также функции закрашивания замкнутых областей.

Функция PatBlt используется для создания массива точек, который формируется как результат комбинации текущей кисти и содержимого области вывода (текущего заполнения):

BOOL CDC::**PatBlt**(
int x,
int y,
int nWidth,
int nHeight,
DWORD dwRop) —

параметр dwRop определяет режим комбинирования текущей кисти и содержимого области, задаваемой параметрами x, y, nWidth и nHeight. Возможны следующие режимы:

PATCOPY	Заполнение области при помощи текущей кисти
PATINVERT	Комбинирование текущего заполнения с кистью при помощи оператора XOR
DSTINVERT	Инвертирование текущего заполнения
BLACKNESS	Заданная область закрашивается сплошным черным цветом
WHITENESS	Заданная область закрашивается сплошным белым цветом
PATPAINT	Комбинирование инвертированного битового массива с текущей кистью при помощи оператора OR и комбинирование результата этой операции с текущим заполнением при помощи оператора OR

Не все устройства поддерживают выполнение функции PatBlt. Для определения поддержки необходимо проверить результат вызова функции GetDeviceCaps с параметром RASTERCAPS (флаг RC_BITBLT).

Функция BitBlt копирует битовый массив из одного контекста устройства в другой:

BOOL CDC::**BitBlt**(
int x,
int y,
int nWidth,
int nHeight,
CDC* pSrcDC,
int xSrc,
int ySrc,
DWORD dwRop) —

параметр dwRop определяет растровую операцию, используемую при копировании битового массива из контекста устройства, заданного параметром pSrcDC в область, задаваемую параметрами x, y, nWidth и nHeight. Параметры xSrc и ySrc задают верхний левый угол области контекста устройства источника, который и определяет копируемый битовый массив. Для операций, не требующих битового массива в качестве источника, параметр pSrcDC должен быть равен NULL.

Можно использовать следующие растровые операции:

BLACKNESS	То же, что и для функции PatBlt
DSTINVERT	То же, что и для функции PatBlt
MERGECOPY	Комбинирование текущего заполнения с текущей кистью при помощи оператора AND
MERGEPAINT	Комбинирование инвертированного битового массива с текущим заполнением при помощи оператора OR
NOTSRCCOPY	Копирование инвертированного битового массива
NOTSRCERASE	Инвертирование результата комбинирования текущего заполнения с битовым массивом при помощи оператора OR
PATCOPY	То же, что и для функции PatBlt
PATINVERT	То же, что и для функции PatBlt
PATPAINT	То же, что и для функции PatBlt
SRCAND	Комбинирование текущего заполнения и битового массива при помощи операции AND
SRCCOPY	Заполнение заданной области битовым массивом
SRCERASE	Комбинирование битового массива и инвертированного текущего заполнения при помощи операции AND
SRCINVERT	Комбинирование битового массива и текущего заполнения при помощи операции XOR
SRCPAINT	Комбинирование битового массива и текущего заполнения при помощи операции OR
WHITENESS	То же, что и для функции PatBlt

Данная функция существенно быстрее выполняется для окон, имеющих стили CS_BYTEALIGNWINDOW или CS_BYTEALIGNCLIENT. Если размеры принимающей области больше битового массива источника, GDI вызывает функцию StretchBlt, т. е. происходит масштабирование.

Не все устройства поддерживают выполнение функции BitBlt. Для определения поддержки необходимо проверить результат вызова функции GetDeviceCaps с параметром RASTERCAPS (флаг RC_BITBLT).

Функция StretchBlt копирует битовый массив из одного контекста в другой, изменяя размеры.

```
BOOL CDC::StretchBlt(
    int x,
    int y,
    int nWidth,
    int nHeight,
    CDC* pSrcDC,
    int xSrc,
    int ySrc,
    int nSrcWidth,
    int nSrcHeight,
    DWORD dwRop) —
```

копирует битовый массив из контекста устройства, заданного параметром pSrcDC, в область, задаваемую параметрами x, y, nWidth и nHeight. Параметры xSrc и ySrc определяют верхний левый угол области контекста устройства источника, а параметры nSrcWidth и nSrcHeight — размеры битового массива. Если параметры nWidth и nSrcWidth имеют разные знаки, то исходный битовый массив отображается зеркально по оси X. Если параметры nHeight и nSrcHeight имеют разные знаки, то исходный битовый массив отображается зеркально по оси Y.

При уменьшении или увеличении образа используется режим преобразования, заданный функцией CDC::StretchBltMode.

Параметр dwRop принимает такие же значения, как и для функции BitBlt, хотя операции, не использующие битовые массивы, для этой функции не имеют смысла — масштабировать будет нечего.

Функция осуществляет масштабирование битового массива в памяти и только потом выполняет копирование (с учетом заданного растрового режима) в контекст устройства.

Не все устройства поддерживают выполнение функции StretchBlt. Для определения поддержки необходимо проверить результат вызова функции GetDeviceCaps с параметром RASTERCAPS (флаг RC_STRETCHBLT).

Следующая функция обеспечивает закрашивание произвольных областей с использованием текущей кисти:

BOOL CDC::**ExtFloodFill**(
int x,
int y,
COLORREF crColor,
UINT nFillType) —

возвращает TRUE при успешном завершении и FALSE в случае возникновения ошибки. Закрашивание невозможно, если пиксель, координаты которого заданы параметрами x и y, определяющий область закрашивания, имеет цвет, совпадающий с crColor, при режиме закрашивания FLOODFILLBORDER, или имеет цвет, не совпадающий с crColor, при режиме закрашивания FLOODFILLSURFACE, или если пиксель находится вне области отсечения.

Параметр crColor определяет либо цвет границы области, либо цвет самой области, что зависит от значения режима определения области закрашивания, задаваемого параметром nFillType. Всего есть два режима определения области закрашивания:

FLOODFILLBORDER	Область закрашивания определяется границей, цвет которой задается параметром crColor; этот режим обеспечивается также вызовом функции CDC::FloodFill
FLOODFILLSURFACE	Область закрашивания определяется цветом, заданным параметром crColor; этот режим полезен для закрашивания областей, граничащих с областями разных цветов

Эта функция поддерживается только контекстами устройств в памяти и растровыми устройствами.

Прокрутка

Эта группа содержит только одну функцию — ScrollDC, которая обеспечивает прокрутку прямоугольных областей, как по вертикали, так и по горизонтали.

Управление выводом текста

В этом разделе мы познакомим вас с возможностями вывода текста, получения информации о выбранных шрифтах и т. п. Однако как и в случае вывода графических объектов, знакомство будет неполным, т. к. в этом разделе мы практически не упоминаем о самих объектах рисования — шрифтах, о которых будет рассказано в следующей главе. Пока же познакомимся с функциями собственно вывода.

Вывод текста

Функции этой группы предназначены для организации вывода текстовых строк. Группа включает функции, от функций простейшего вывода и настройки режимов выравнивания до функций получения ряда характеристик, таких как наименование шрифта (GetTextFace), длина заданной строки в логических единицах устройства вывода (GetTextExtent и GetOutputTextExtent) и установка режима выравнивания (GetTextAlign, SetTextAlign и SetTextJustification). Из всех функций вывода текста остановимся только на основных и самых мощных.

Функция ExtTextOut позволяет вывести текстовую строку в заданную прямоугольную область с использованием текущего шрифта и управляет режимами отображения фона. (Для вывода текста с заданной позиции используются функции TextOut и TabbedTextOut.) Функция представлена двумя версиями:

```
virtual BOOL CDC::ExtTextOut(
    int x,
    int y,
    UINT nOptions,
    LPCRECT lpRect,
    LPCTSTR lpszString,
    UINT nCount,
    LPINT lpDxWidths)
```

и

```
BOOL CDC::ExtTextOut(
    int x,
    int y,
    UINT nOptions,
    LPCRECT lpRect,
    const CString & str,
    LPINT lpDxWidths)
```

Основное отличие этих функций состоит в механизме задания строки символов. В первой версии функции строка задается параметром lpszString, а ее длина задается параметром nCount; если последний равен −1, то длина

строки вычисляется. Во второй версии строка задается ссылкой на объект класса CString, задаваемый параметром str.

Параметр lpDxWidths указывает на массив чисел, определяющих расстояние между символами; если значение параметра равно NULL, то величина промежутков задается значением по умолчанию.

Параметры x и y задают координаты (в логических единицах) левого верхнего угла области вывода первого символа строки. Параметр lpRect задает прямоугольную область, в которую должен выводиться текст, и может иметь значение NULL.

Параметр nOptions может принимать одно, оба или не принимать ни одного из следующих значений, смысл которых понятен из описания:

ETO_CLIPPED	Вывод текста осуществляется только в область, заданную параметром lpRect, а параметры x и y игнорируются
ETO_OPAQUE	Перед выводом текста прямоугольная область, заданная параметром lpRect, либо область, требуемая для вывода текста, заполняется текущим цветом фона

По умолчанию текущая позиция вывода после вызова функции не изменяется. Однако, если задан флаг TA_UPDATECP при вызове функции CDC::SetTextAlign, параметры x и y будут проигнорированы, вывод будет осуществляться с текущей позиции, а по завершении вывода текущая позиция будет установлена либо в конец выведенной строки, либо установлена в соответствии с последним элементом массива (параметр lpDxWidths), в зависимости от того, какое значение больше.

Функция DrawText выводит текст с различными видами форматирования:

```
virtual int CDC::DrawText(
    LPCTSTR lpszString,
    int nCount,
    LPRECT lpRect,
    UINT nFormat)
```

и

```
int CDC::DrawText(
    const CString & str,
    LPRECT lpRect,
    UINT nFormat) —
```

обе реализации в случае успеха возвращают высоту выведенного текста. Строка, как и в функции ExtTextOut, задается параметрами lpszString совместно с nCount или str. Параметр lpRect указывает на структуру RECT или объект класса CRect, которые содержат координаты (в логических единицах) прямоугольной области, в которую собственно и выводится форматируемый текст.

Параметр nFormat определяет способ форматирования заданного текста и может быть комбинацией следующих флагов:

DT_BOTTOM	Устанавливает выравнивание по нижней границе заданной прямоугольной области; должен комбинироваться с DT_SINGLELINE
DT_CALCRECT	Определение ширины и высоты прямоугольной области вывода; для текста в несколько строк используется ширина прямоугольной области, заданной параметром lpRect, и высота настраивается так, чтобы поместился весь текста; для текста в одну строку, напротив, ширина настраивается таким образом, чтобы точно соответствовать ширине выводимого текста; в любом случае функция возвращает высоту области, необходимую для полного вывода текста, однако собственно вывод *не осуществляется*; не может использоваться совместно с DT_TABSTOP
DT_CENTER	Устанавливает горизонтальное выравнивание по центру заданной области
DT_EXPANDTABS	При выводе производится замена символов табуляции на пробелы; каждая табуляция имеет размер в восемь символов
DT_EXTERNALLEADING	При определении высоты строки учитывается размер между строками; не может использоваться совместно с DT_TABSTOP
DT_LEFT	Устанавливает выравнивание по левому краю заданной прямоугольной области
DT_NOCLIP	Осуществляет вывод без учета ширины задаваемой прямоугольной области; не может использоваться совместно с DT_TABSTOP
DT_NOPREFIX	При выводе амперсанд (&) не интерпретируется как указание подчеркивания следующего за ним символа, а двойной амперсанд (&&) — как указание на вывод только одного; не может использоваться совместно с DT_TABSTOP
DT_RIGHT	Устанавливает выравнивание по правому краю заданной прямоугольной области
DT_SINGLELINE	Указание на вывод одной строки текста, при этом символы перевода строки и возврата каретки не разбивают строку
DT_TABSTOP	Устанавливает размер табуляции: в старшем байте параметра nFormat указано число символов на табуляцию (по умолчанию размер табуляции равен восьми символам)
DT_TOP	Устанавливает выравнивание по верхней границе заданной прямоугольной области; должен комбинироваться с DT_SINGLELINE
DT_VCENTER	Устанавливает вертикальное выравнивание по центру заданной области; должен комбинироваться с DT_SINGLELINE
DT_WORDBREAK	Устанавливает режим автоматического переноса строки между словами, если следующее слово выходит за рамки заданной прямоугольной области; символы переноса строки интерпретируются как обычно

Функция DrawText для вывода текста использует текущий шрифт, а также текущие цвета символов и фона. Если флаг DT_NOCLIP не задан, текст не

выходит за заданные границы. При этом в случае, если текущий шрифт слишком велик для заданной прямоугольной области, функция DrawText не делает попыток его уменьшить или заменить.

По умолчанию текущая позиция вывода после вызова функции не изменяется. Однако, если задан флаг TA_UPDATECP при вызове функции CDC::SetTextAlign, левый верхний угол области вывода будет проигнорирован — вывод будет осуществляться с текущей позиции, а по завершении вывода текущая позиция будет установлена в конец выведенного текста. При задании флага TA_UPDATECP флаг DT_WORDBREAK игнорируется.

К этой же группе относятся функции GetOutputTextMetrics и GetTextMetrics, которые позволяют получить основную информацию о текущем шрифте контекста устройства графического вывода (m_hDC) и контекста, используемого для получения параметров вывода (m_hAttribDC), соответственно:

BOOL CDC::**GetOutputTextMetrics**(
LPTEXTMETRIC lpMetrics)

и

BOOL CDC::**GetTextMetrics**(
LPTEXTMETRIC lpMetrics) —

возвращают TRUE в случае успешного завершения и FALSE в противном случае. В качестве единственного параметра используется указатель на очень важную структуру TEXTMETRIC, которая и будет содержать, собственно, запрашиваемую информацию. Все размеры задаются в логических единицах, т. е. соответствуют текущей системе координат, устанавливаемой функцией SetMapMode.

```
typedef struct tagTEXTMETRIC {
    LONG tmHeight;
    LONG tmAscent;
    LONG tmDescent;
    LONG tmInternalLeading;
    LONG tmExternalLeading;
    LONG tmAveCharWidth;
    LONG tmMaxCharWidth;
    LONG tmWeight;
    LONG tmOverhang;
    LONG tmDigitizedAspectX;
    LONG tmDigitizedAspectY;
    BCHAR tmFirstChar;
    BCHAR tmLastChar;
    BCHAR tmDefaultChar;
    BCHAR tmBreakChar;
    BYTE tmItalic;
    BYTE tmUnderlined;
```

```
    BYTE tmStrickOut;
    BYTE tmPitchAndFamily;
    BYTE tmCharSet;
} TEXTMETRIC;
```

Остановимся подробнее на элементах структуры, часть которых проиллюстрирована на рис. 17.6:

Рис. 17.6. Определение основных параметров шрифта с помощью элементов структуры TEXTMETRIC

tmHeight	Высота символов (tmAscent + tmDescent)
tmAscent	Высота области над базовой линией
tmDescent	Размер области под базовой линией
tmInternalLeading	Размер дополнительной области над символом, которая может быть использована для знаков ударения или диакритических символов[1]
tmExternalLeading	Размер области между строками текста; при выводе эта область никак не заполняется ни в режиме OPAQUE (непрозрачный), ни в режиме TRANSPARENT (прозрачный)

[1] И в русском языке есть символы, использующие эту область. Например, буква "Й", которая выше "И" как раз на величину tmInternalLeading.

tmAveCharWidth	Средняя ширина символов; обычно определяется шириной для символа "x"; это значение не учитывает ширину символов в режимах **bold** (полужирный) и *italic* (курсив)
tmMaxCharWidth	Максимальная ширина символов шрифта
tmWeight	"Вес" или, что еще хуже, "жирность" символов шрифта
tmOverhang	Значение, которое может быть добавлено для определения ширины строки для некоторых синтетических шрифтов (полужирных или курсивных)
tmDigitizedAspectX, tmDigitizedAspectY	Отношение tmDigitizedAspectX к tmDigitizedAspectY определяет масштаб устройства, для которого разрабатывался шрифт
tmFirstChar	Первый определенный в шрифте символ
tmLastChar	Последний определенный в шрифте символ
tmDefaultChar	Символ, используемый для замены неопределенных символов
tmBreakChar	Символ, используемый для разделения слов при выравнивании
tmItalic	Ненулевое значение параметра указывает, что начертание текущего шрифта — курсивное
tmUnderlined	Ненулевое значение параметра указывает, что текст, использующий текущий шрифт, будет подчеркнут
tmStrickOut	Ненулевое значение параметра указывает, что текст, использующий текущий шрифт, будет перечеркнут
tmPitchAndFamily	Содержит флаги стиля шрифта. Старшие четыре бита определяют семейство шрифтов: FF_DECORATIVE, FF_DONTCARE, FF_MODERN, FF_ROMAN, FF_SCRIPT, FF_SWISS. Младшие четыре бита определяют тип шрифта: TMPF_FIXED_PITCH — Определяет пропорциональность шрифта: если бит содержит 1, шрифт — пропорциональный, если 0 — все символы шрифта имеют одинаковую ширину (моноширинный шрифт) TMPF_VECTOR — Определяет, является ли шрифт векторным TMPF_TRUETYPE — Если этот бит равен 1, шрифт является шрифтом TrueType TMPF_DEVICE — Если этот бит равен 1, шрифт является шрифтом устройства

tmCharSet	Определяет таблицу символов шрифта; Windows определяет следующие кодировки: ANSI_CHARSET, DEFAULT_CHARSET, SYMBOL_CHARSET, SHIFTJIS_CHARSET, GB2312_CHARSET, HANGEUL_CHARSET, CHINESEBIG5_CHARSET, OEM_CHARSET, JOHAB_CHARSET, HEBREW_CHARSET, ARABIC_CHARSET, GREEK_CHARSET, TURKISH_CHARSET, THAI_CHARSET, EASTEUROPE_CHARSET, RUSSIAN_CHARSET, MAC_CHARSET, BALTIC_CHARSET

Информация о шрифтах

Функции этой группы (GetFontData, GetKerningPairs, GetOutlineTextMetrics, GetGlyphOutline, GetCharABCWidth, GetCharWidth и GetOutputCharWidth) позволяют получить полную информацию о шрифтах и на ее основе выбирать подходящий шрифт и точно определять параметры вывода, такие как размеры, область вывода и т. п.

Дополнительные функции

В этом разделе указаны группы функций, которые не занимаются непосредственно выводом или настройкой, однако связаны с контекстом устройства.

Интерфейс низкого уровня с устройствами

Класс CDC имеет в своем составе функции, которые могут напрямую взаимодействовать не только с принтерами, но и с графическими устройствами (видео-картами). Такие функции (Escape и DrawEscape) дают возможность программе вызывать выполнение действий, которые не реализованы средствами GDI. Естественно, наличие таких функций отчасти противоречит идеологии аппаратной независимости, но, тем не менее, они есть.

Управление процессом печати документов

Функции этой группы предоставляют возможность взаимодействия с системным диспетчером печати (StartDoc, StartPage, EndPage, EndDoc и т. д.).

Метафайлы

Функции этой группы предназначены для отображения содержимого метафайла (функция PlayMetaFile) и добавления комментария к метафайлу (функция AddMetaFileComment).

Глава 18

Классы графических объектов

Как и для класса CDC, рассмотрим, что именно предоставляет библиотека классов MFC для работы с графическими объектами, иллюстрируя материал небольшими фрагментами кодов.

Класс CGdiObject

Класс CGdiObject является базовым для всех классов, представляющих и обеспечивающих интерфейс с графическими объектами Windows. Напомним, что таких объектов шесть — карандаш, кисть, битовый массив, палитра, шрифт и регион. Несмотря на то что класс CGdiObject не является абстрактным, нет смысла создавать объекты этого класса. Значение этого класса в том, что он имеет набор функций, общих для всех производных классов.

Для получения представления о возможностях, предоставляемых классом, познакомимся с его компонентами.

Для хранения дескриптора одного из шести типов объектов Windows, связанного с объектом класса, используется член класса m_hObject :

HANDLE CGdiObject::**m_hObject**

Несмотря на то что m_hObject является общедоступным членом, класс CGdiObject имеет в своем составе функцию, возвращающую копию дескриптора:

HGDIOBJ CGdiObject::**GetSafeHandle**( )

Достоинства использования этой функции в том, что она возвращает значение NULL в случае, если сам объект CGdiObject не существует, и указатель this равен NULL. Любая иная проверка, подразумевающая доступ к членам класса, в таком случае приведет к ошибке доступа к памяти.

Примечание

Напомним, что и в Windows 95/98, и в Windows NT младшие адреса виртуального адресного пространства зарезервированы для работы с нулевыми и близкими к нулю указателями на ранней стадии отладки, т. к. такие действия ошибочны.

```
CGdiObject *pObject = NULL;
if (pObject->m_hObject != NULL)  // В этом месте обращение
                                 // по несуществующему адресу -
                                 // ошибка доступа к памяти
  ...
if (pObject->GetSafeHandle() != NULL) // Ошибки нет, т. к.
                                      // фактически доступа
                                      // к данным-членам класса нет
  ...
```

Аналогично классу CDC, класс CGdiObject имеет в своем составе статическую функцию, задача которой состоит в том, чтобы получить указатель на объект одного из классов графических объектов, созданный на основе заданного дескриптора:

static CGdiObject* PASCAL CGdiObject::**FromHandle**(
HGDIOBJ hObject) —

скорее всего, указатель будет указывать на временный объект. Действия над временными графическими объектами аналогичны действиям над временными объектами классов контекстов устройств (см. предыдущую главу).

Созданный ранее графический объект Windows можно явно удалить, т. е. освободить соответствующие системные ресурсы, для чего необходимо воспользоваться функцией DeleteObject:

BOOL CGdiObject::**DeleteObject**( ) —

деструктор класса в любом случае удалит графический объект Windows как раз при помощи этой функции. Однако следует понимать, что этот объект не может быть текущим для контекста устройства.

Фактически вызов функции DeleteObject может понадобиться только в одном случае — если один и тот же объект, созданный на основе класса CGdiObject, необходим для обеспечения интерфейса с различными графическими объектами Windows. Поясним сказанное на примере.

Функции OnPaint, классов CNeptuneFrame и CTitanFrame рисуют квадрат, причем для горизонтальных и вертикальных его сторон используются различные по толщине и цвету карандаши.

В первом случае для каждого типа сторон создаются различные объекты класса CPen — pen_1 и pen_2.

```
void CNeptuneFrame::OnPaint()
{
  CPaintDC dc(this);
  CPen pen_1(PS_SOLID, 1, RGB(0, 0, 0));
  CPen *pPen = dc.SelectObject(&pen_1);
  dc.MoveTo(100, 100);
  dc.LineTo(200, 100);
  dc.MoveTo(100, 200);
  dc.LineTo(200, 200);
  CPen pen_2(PS_SOLID, 2, RGB(192, 192, 192));
  // Заменяем первый карандаш вторым
  dc.SelectObject(&pen_2);
  // Рисуем при помощи второго карандаша,
  // но первый еще занимает ресурсы
  dc.MoveTo(100, 100);
  dc.LineTo(100, 200);
  dc.MoveTo(200, 100);
  dc.LineTo(200, 200);
  // Естественно, заменяем наш карандаш
  // тем, что был установлен ранее
  dc.SelectObject(pPen);
}
```

Во втором случае можно обойтись только одним объектом класса CPen для каждого типа сторон.

```
void CTitanFrame::OnPaint()
{
  CPen *pPen, pen;
  pen.CreatePen(PS_SOLID, 1, RGB(0, 0, 0));
  pPen = dc.SelectObject(&pen);
  dc.MoveTo(100, 100);
  dc.LineTo(200, 100);
  dc.MoveTo(100, 200);
  dc.LineTo(200, 200);
  dc.SelectObject(pPen);
  pen.DeleteObject();
  // Теперь программа не занимает ресурсы для хранения карандаша
  pen.CreatePen(PS_SOLID, 2, RGB(192, 192, 192));
  dc.SelectObject(&pen);
  dc.MoveTo(100, 100);
```

```
  dc.LineTo(100, 200);
  dc.MoveTo(200, 100);
  dc.LineTo(200, 200);
  // Естественно, заменяем наш карандаш
  // тем, что был установлен ранее
  dc.SelectObject(pPen);
}
```

Обратите внимание на то, что перед вызовом (неявным) функции DeleteObject мы восстановили в контексте устройства ранее установленный карандаш. Связано это с тем, что по понятным причинам нельзя удалять графический объект в тот момент, когда он является текущим для контекста устройства. Простое повторное создание нового объекта библиотекой запрещено. Функция CGdiObject::Attach, осуществляющая связывание объекта Windows с объектом соответствующего графического класса, не отвечает за удаление ранее созданных графических объектов Windows. Вместо этого она сначала проверяет наличие таких объектов и, при обнаружении, генерирует ошибку. Вот исходный код (и соответствующий комментарий), осуществляющий эти действия:

```
BOOL CGdiObject::Attach(HGDIOBJ hObject)
{
  ASSERT(m_hObject == NULL);  // only attach once,
                              // detach on destroy
  ...
}
```

Обратите также внимание, что явно функция DeleteObject не вызывается — она используется в деструкторе, который автоматически вызывается по выходу из области видимости, в данном случае по завершении функции OnPaint.

Здесь надо быть очень внимательными и помнить, что после выхода из области видимости объект класса уже не существует, а вот графический объект может быть фактически не удален. Чтобы это не случилось, следует обязательно заменять созданные графические объекты в контексте устройства, иначе соответствующий объект Windows удален не будет даже после вызова функции DeleteObject, что приведет к постепенному исчерпанию ресурсов и краху системы.

Класс CGdiObject имеет в своем составе функцию, которая создает объекты, связанные с предопределенными графическими объектами Windows:

BOOL CGdiObject::**CreateStockObject**(int nIndex) —

параметр nIndex принимает значения, аналогичные тем, что используются функцией CDC::SelectStockObject, описанной в предыдущей главе.

Функция GetObject позволяет получить информацию о графическом объекте, доступ к которому может быть не реализован функциями-членами соответствующих классов библиотеки MFC:

int CGdiObject::**GetObject**(
int nCount,
LPVOID lpObject) —

параметр lpObject является указателем на блок данных, длина которого задается параметром nCount. Функция возвращает количество байт информации, фактически переданной приложению, и 0 в случае возникновения ошибки. Для каждого графического класса, кроме CRgn, предусмотрен доступ к перечисленным ниже структурам:

Класс	Тип данных
CPen	LOGPEN
CBrush	LOGBRUSH
CFont	LOGFONT
CBitmap	BITMAP
CPalette	int — количество элементов палитры

Например, для битовых массивов использование этой функции может быть единственным способом получить информацию о размерах, т. к. соответствующая функция CBitmap::GetBitmapDimension возвращает размеры только в том случае, если они были установлены функцией Cbitmap ::SetBitmapDimension.

В заключение этого раздела необходимо отметить, что у всех классов, инкапсулирующих графические объекты Windows, есть функции-члены, которые являются перегруженными операторами, приводящими соответствующий объект класса к дескриптору объекта Windows, упрощая использование функций Win32 API:

CGdiObject::**operator HGDIOBJ**( )

CPen::**operator HPEN**( )

CBrush::**operator HBRUSH**( )

CFont::**operator HFONT**( )

CBitmap::**operator HBITMAP**( )

CPalette::**operator HPALETTE**( )

CRgn::**operator HRGN**( )

Далее мы достаточно подробно остановимся на пяти из шести классов, инкапсулирующих возможности графических объектов Windows. Класс CPalette не рассматривается из-за его достаточно редкого использования, в основном в специальных областях обработки изображений.

Карандаш (класс CPen)

Класс CPen инкапсулирует графический объект Windows "карандаш". Карандаши используются для рисования различных линий и контуров объектов. Эти объекты имеют наиболее богатые возможности их создания — в Windows имеется целый набор различных предопределенных стилей, свойств и типов, которые открывают возможность простого и эффективного использования карандашей и, соответственно, рисования самых разнообразных линий.

Как и все другие классы графических объектов, класс CPen позволяет создавать объекты класса, не связывая их при этом с объектами Windows, или наоборот — при создании объекта класса непосредственно создавать и соответствующий графический объект.

Что касается последней возможности, то она дублируется наличием функций создания карандаша Windows уже после создания объекта класса. О возможных способах диагностики результатов создания мы говорили в главе 17. Так как собственно создание объекта Windows заключается в формировании соответствующих данных, которые передаются системе функциями-членами класса CPen, мы не будем дублировать описание функций создания, а подробно остановимся только на конструкторах.

CPen::**CPen**( ) —

конструктор без параметров, позволяет создавать объекты класса CPen, никак не связанные с объектами Windows, и прежде чем его использовать, необходимо создать такой объект.

```
CPen::CPen(
    int nPenStyle,
    int nWidth,
    COLORREF crColor)
```

и

```
CPen::CPen(
    int nPenStyle,
    int nWidth,
    const LOGBRUSH* pLogBrush,
    int nStyleCount = 0,
    const DWORD* lpStyle = NULL) —
```

конструкторы, позволяющие при создании объекта класса CPen создавать "карандаш" Windows.

Существование двух версий конструкторов, создающих объект Windows, объясняется различием создаваемых "карандашей" и возможными значениями, прежде всего, параметра nPenStyle. Рассмотрим отдельно для каждой версии конструктора возможные значения параметра nPenStyle.

С помощью конструктора первой версии можно создавать карандаши с предопределенными в Windows стилями для рисования следующих типов линий:

PS_SOLID	Сплошная линия
PS_DASH	Длинные штрихи, толщина которых равна 1
PS_DOT	Короткие штрихи, толщина которых равна 1
PS_DASHDOT	Штрих-пунктир, толщина которого равна 1
PS_DASHDOTDOT	Двойной штрих-пунктир, толщина которого равна 1
PS_NULL	Прозрачная линия
PS_INSIDEFRAME	Сплошная линия для рисования по внутренней границе замкнутой области, например, при использовании функций класса CDC — Rectangle, RoundRect, Ellipse, Chord и т. д.; при использовании "карандаша" данного типа для рисования линий, например, при помощи функции LineTo, результат будет тем же, что и для стиля PS_SOLID; еще одна область применения этого стиля — создание "карандашей" толщиной более одного пикселя и имеющих цвет, недоступный для текущего устройства; в этом случае линии будут состоять из пикселей доступных цветов, совокупность которых наиболее близко будет соответствовать заданному цвету

Для наглядности продемонстрируем линии различных стилей. Обратите внимание, что при выводе линий одной длины фактически некоторые штриховые линии оканчиваются раньше сплошных (рис. 18.1), что вполне естественно — эти линии завершились в месте пустого пространства между штрихами. Такой эффект может негативно сказаться при решении конкретных задач.

```
void CLineStyleFrame::OnPaint()
{
  CPaintDC dc(this);
  struct { int m_nStyle; LPTSTR m_csStyle; } lnStyle[] = {
    PS_SOLID,       _T("PS_SOLID"),
    PS_DASH,        _T("PS_DASH"),
    PS_DOT,         _T("PS_DOT"),
    PS_DASHDOT,     _T("PS_DASHDOT"),
    PS_DASHDOTDOT,  _T("PS_DASHDOTDOT"),
    PS_NULL,        _T("PS_NULL"),
    PS_INSIDEFRAME, _T("PS_INSIDEFRAME")
  };
  //
  // Чтобы при любом размере шрифта тексты не пересекались,
```

```
  // получим и учтем характеристики текущего шрифта
  //
  TEXTMETRIC tm;
  dc.GetTextMetrics(&tm);
  // Устанавливаем режим "прозрачного" вывода
  dc.SetBkMode(TRANSPARENT);
  for (int i = 0; i < sizeof(lnStyle) / sizeof(lnStyle[0]); i++)
  {
    // Будем рисовать "карандашами" разными по стилю,
    // но имеющими единичную толщину и черный цвет
    CPen pen(lnStyle[i].m_nStyle, 1, RGB(0, 0, 0));
    CPen *pPen = dc.SelectObject(&pen);
    // Учитываем высоту символов и расстояние между строками
    int nY = 10+i*(tm.tmHeight+tm.tmExternalLeading);
    dc.TextOut(10, nY, lnStyle[i].m_csStyle);
    dc.MoveTo(200, nY+(tm.tmHeight+tm.tmExternalLeading) / 2);
    dc.LineTo(400, nY+(tm.tmHeight+tm.tmExternalLeading) / 2);
    dc.SelectObject(pPen);
  }
}
```

Рис. 18.1. Результат работы функции CLineStyleFrame::OnPaint

Использование второй версии конструктора позволяет комбинировать в параметре nPenStyle стиль, тип, режим рисования концов линии и режим соединения линий. Каждый вид параметров может быть представлен только одним значением, а заданные значения комбинируются при помощи оператора OR (|).

"Карандаши" Windows могут принадлежать одному из двух типов — PS_GEOMETRIC и PS_COSMETIC, которые отличаются использованием кистей для рисования линий и требованиями к заданию ширины. Кроме того, от выбранного типа зависит интерпретация параметров задания размеров штрихов для пользовательских стилей.

Вторая версия конструктора поддерживает два дополнительных стиля — PS_ALTERNATE (может использоваться только совместно с типом PS_COSMETIC) и PS_USERSTYLE, который требует задания массива длин штрихов для рисования линий.

Для "карандашей" также можно задать один из трех режимов рисования концов (это имеет смысл при рисовании широких линий), для первых двух режимов фактическая длина линии может быть больше заданной:

PS_ENDCAP_ROUND	Концы линий закруглены
PS_ENDCAP_SQUARE	Концы линий имеют прямоугольную форму
PS_ENDCAP_FLAT	То же, что и для PS_ENCAP_SQUARE, но концы линии не выходят за заданные размеры.

Для соединения линий можно выбирать один из следующих режимов:

PS_JOIN_BEVEL	Соединение обрезано
PS_JOIN_MITER	Соединение определяется продолжением контура линий, если только расстояние между концами линий не превышает заданную функцией CDC::SetMiterLimit величину, в противном случае соединение обрезается, как и в случае задания режима PS_JOIN_BEVEL
PS_JOIN_ROUND	Соединение закруглено

Одно из требований по использованию режимов рисования концов линий и соединений заключается в том, что рисовать линии "карандашами", созданными с флагами PS_ENDCAP_SQUARE, PS_ENDCAP_FLAT, PS_JOIN_BEVEL и PS_JOIN_MITER, можно только на основе механизма контуров, что и продемонстрировано ниже в примере. Обратите внимание, что по оси каждой линии, имеющей тип PS_GEOMETRIC, для наглядности проведены сплошные тонкие линии с теми же координатами (рис. 18.2).

```
void CCapJoinFrame::OnPaint()
{
  CPaintDC dc(this);
  int nEnd[] =
    { PS_ENDCAP_ROUND, PS_ENDCAP_SQUARE, PS_ENDCAP_FLAT };
  int nJoin[] =
    { PS_JOIN_BEVEL, PS_JOIN_MITER, PS_JOIN_ROUND };
  CString csEnd[] = { _T("PS_ENDCAP_ROUND"),
    _T("PS_ENDCAP_SQUARE"), _T("PS_ENDCAP_FLAT") };
  CString csJoin[] = { _T("PS_JOIN_BEVEL"),
    _T("PS_JOIN_MITER"), _T("PS_JOIN_ROUND") };
  //
  // Создаем новый шрифт и заменяем им текущий,
  // т. к. он слишком велик для вывода имен режимов
```

```
//
CFont font;
font.CreateFont(12, 0, 0, 0, FW_BOLD, 0, 0, 0, DEFAULT_CHARSET,
  OUT_DEFAULT_PRECIS, CLIP_DEFAULT_PRECIS, PROOF_QUALITY,
  VARIABLE_PITCH | FF_DONTCARE, "Arial");
CFont *pFont = dc.SelectObject(&font);
// Устанавливаем режим "прозрачного" вывода
dc.SetBkMode(TRANSPARENT);
// Перебираем все возможные комбинации (их 9)
for (int i = 0; i < 3; i++)
{
  dc.TextOut(100+i*120, 10, csEnd[i]);
  dc.TextOut(10, 90+i*80, csJoin[i]);
  for (int j = 0; j < 3; j++)
  {
    // Для создания карандашей понадобится структура LOGBRUSH,
    // которая определена в Win32 API для описания кистей
    LOGBRUSH lb;
    lb.lbStyle = BS_SOLID;
    lb.lbColor = RGB(128, 128, 128);
    lb.lbHatch = 0;
    CPen pen(
      PS_SOLID | PS_GEOMETRIC | nEnd[i] | nJoin[j],
      20, &lb);
    CPen *pPen = dc.SelectObject(&pen);
    // Чтобы увидеть отличие режимов, придется создавать
    // контуры и только затем их обводить
    dc.BeginPath();
    dc.MoveTo(100+i*120,  50+j*80);
    dc.LineTo(140+i*120, 150+j*80);
    dc.LineTo(180+i*120,  50+j*80);
    dc.EndPath();
    dc.StrokePath();
    // Для рисования базовых линий используем
    // карандаш, заданный по умолчанию
    dc.SelectObject(pPen);
    dc.MoveTo(100+i*120,  50+j*80);
    dc.LineTo(140+i*120, 150+j*80);
    dc.LineTo(180+i*120,  50+j*80);
  }
```

```
  }
  // Шрифт, так же как и другие графические объекты,
  // надо восстановить в контексте
  dc.SelectObject(pFont);
}
```

Рис. 18.2. Результат работы функции CCapJoinFrame::OnPaint

Создавая "карандаш", нужно также задать ширину при помощи параметра nWidth. При задании в первой версии конструктора ширины, равной 0, гарантируется вывод линии шириной в один пиксель независимо от режима вывода (mapping mode). При использовании второй версии конструктора для "карандашей" типа PS_GEOMETRIC ширина задается в логических единицах, а для "карандашей" типа PS_COSMETIC ширина должна быть равна 1.

Для задания цвета карандаша используется параметр crColor.

Параметр pLogBrush является указателем на структуру LOGBRUSH для создания кисти, с помощью которой будет производиться рисование линий. Для карандашей типа PS_COSMETIC элемент структуры LOGBRUSH::lbStyle должен содержать значение BS_SOLID.

Параметр lpStyle указывает на массив, содержащий nStyleCount чисел, определяющих попарно длину штриха и расстояние между штрихами для "карандашей", имеющих стиль PS_USERSTYLE.

Как упоминалось в начале раздела, создать "карандашей" можно с помощью функций CreatePen, CreatePenIndirect или CreateStockObject. Функция CreatePenIndirect создает только обычные "карандаши", как и конструктор первой версии, описанный выше.

Кисть (класс CBrush)

Класс CBrush инкапсулирует графический объект Windows "кисть", который служит, прежде всего, основой при закрашивании замкнутых фигур. Кисти могут быть однотонные, со штриховкой или созданные на основе шаблона.

Как и все другие классы графических объектов, класс CBrush имеет в своем составе конструктор без параметров:

CBrush::**CBrush**( ) —

создает объект класса, но для его использования требует вызова одной из "создающих" функций — CreateSolidBrush, CreateHatchBrush, CreateBrushIndirect, CreatePatternBrush, CreateDIBPatternBrush, CreateSysColorBrush. Первые четыре функции полностью дублируют возможности конструкторов. Пятая функция создает кисть на основе битового массива, но, в отличие от четвертой, этот массив хранит изображение в независимом от устройств виде. Последняя функция создает кисть со штриховкой на основе системного цвета.

Кроме конструктора по умолчанию, класс CBrush имеет в своем составе конструкторы, позволяющие создавать кисти одного цвета (однотонные):

CBrush::**CBrush**(COLORREF crColor)

кисти со штриховкой:

```
CBrush::CBrush(
    int nIndex,
    COLORREF crColor)
```

и кисти на основе битового массива, используемого в качестве шаблона:

CBrush::**CBrush**(CBitmap* pBitmap)

Параметр crColor определяет цвет закрашивания для кистей одного цвета, а для кистей со штриховкой этот параметр определяет цвет только штриховки.

Чтобы создать кисть со штриховкой, необходимо установить параметр nIndex в одно из следующих значений:

HS_BDIAGONAL	45-градусная штриховка слева направо и сверху вниз
HS_CROSS	Горизонтальная и вертикальная штриховка крест-накрест
HS_DIAGCROSS	45-градусная штриховка крест-накрест
HS_FDIAGONAL	45-градусная штриховка слева направо и снизу вверх
HS_HORIZONTAL	Горизонтальная штриховка
HS_VERTICAL	Вертикальная штриховка

Для создания кисти на основе битового массива необходимо сначала создать или загрузить битовый массив, а затем, передав указатель на него, вызвать последний из перечисленных конструкторов. При этом надо иметь в виду, что битовый массив должен иметь размеры не меньше чем 8×8 пикселей, а

сама кисть будет создана на основе самой левой верхней квадратной области битового массива размером также 8×8 пикселей.

Шрифт (класс CFont)

Класс CFont инкапсулирует графический объект Windows "шрифт". Для создания объекта этого класса имеется конструктор без параметров, однако чтобы создать собственно шрифт, необходимо вызвать одну из следующих функций: CreateFont, CreateFontIndirect, CreatePointFont или CreatePointFontIndirect. Для использования созданного шрифта необходимо установить (выбрать) его в конкретный контекст устройства, а по завершении использования заменить ранее установленным шрифтом — все как и для других графических объектов.

Многие классы, инкапсулирующие графические объекты Windows, имеют в своем составе функции создания объектов при помощи структур. Однако мы описывали, в основном, те функции, которые принимают в качестве параметров те же значения, что и поля таких структур. В случае со шрифтами мы поступим иначе, прежде всего потому, что структура LOGFONT, используемая для создания шрифта, достаточно часто используется не только вместе с классом CFont, а также, например, и при выборе шрифта с помощью блока диалога, созданного при помощи класса CFontDialog (см. главу 15).

Итак, для создания шрифта используется функция CreateFontInderect:

BOOL CFont::**CreateFontIndirect**(
const LOGFONT* lpLogFont) —

параметр lpLogFont указывает на структуру LOGFONT, которая, в свою очередь, определяет шрифт, необходимый приложению.

Остановимся более подробно на составе структуры и описании ее полей:

```
typedef struct tagLOGFONT {
    LONG lfHeight;
    LONG lfWidth;
    LONG lfEscapement;
    LONG lfOrientation;
    LONG lfWeight;
    BYTE lfItalic;
    BYTE lfUnderline;
    BYTE lfStrikeOut;
    BYTE lfCharSet;
    BYTE lfOutPrecision;
    BYTE lfClipPrecision;
```

```
    BYTE lfQuality;
    BYTE lfPitchAndFamily;
    TCHAR lfFaceName[LF_FACESIZE];
} LOGFONT;
```

Поле lfHeight определяет высоту (в логических единицах) *знакомест* или *символов шрифта*. Высота последних определяется как разность между полной высотой знакоместа и размером области для отображения специальных знаков, например, ударения.

Примечание

Более подробно термины, описывающие размеры символов, были описаны в предыдущей главе.

Это поле структуры LOGFONT может интерпретироваться, в зависимости от знака, следующим образом:

> 0	Значение преобразуется в единицы устройства вывода и используется для выбора шрифта из имеющихся, соответствующего по высоте знакоместа
0	Для нахождения шрифта используется значение высоты по умолчанию
< 0	Значение преобразуется в единицы устройства вывода и используется для выбора шрифта из имеющихся, соответствующего по высоте символа

При поиске подходящего шрифта выбирается шрифт с наиболее подходящей высотой знакоместа или символа (высота должна быть меньше или равна запрашиваемой).

Для режима отображения MM_TEXT высота может быть определена через размер шрифта, задаваемого в пунктах (points), следующим образом:

```
lfHeight = -MulDiv(PointSize, dc.GetDeviceCaps(LOGPIXELSY), 72);
```

Поле lfWidth определяет среднюю ширину символов в логических единицах. Нулевое значение поля lfWidth обеспечивает выбор шрифта из имеющихся, наиболее подходящего к конкретному устройству по возможностям масштабирования.

Поле lfEscapement определяет угол (в десятых долях градуса) между базовой линией выводимого текста и горизонталью.

Поле lfOrientation определяет угол (в десятых долях градуса) между базовой линией каждого выводимого символа и горизонталью.

Примечание

К сожалению, в рамках Windows 95/98 значения обоих полей lfEscapement и lfOrientation должны быть равны, т. е. можно выводить под наклоном только текст всей строки, когда базовые линии символов лежат на базовой линии строки. В Windows NT при установке графического режима GM_ADVANCED, устанавливаемого функцией Win32 API ::SetGraphicsMode, возможности по выводу текста расширяются.

Поле lfWeight определяет толщину линий начертания символов и может принимать значения от 0 до 1000. При нулевом значении используется значение по умолчанию. Некоторым значениям даны символьные наименования:

FW_DONTCARE	0
FW_THIN	100
FW_EXTRALIGHT	200
FW_ULTRALIGHT	200
FW_LIGHT	300
FW_NORMAL	400
FW_REGULAR	400
FW_MEDIUM	500
FW_SEMIBOLD	600
FW_DEMIBOLD	600
FW_BOLD	700
FW_EXTRABOLD	800
FW_ULTRABOLD	800
FW_HEAVY	900
FW_BLACK	900

Поле lfItalic определяет курсивное начертание шрифта.

Поле lfUnderline определяет подчеркнутое начертание шрифта.

Поле lfStrikeOut определяет перечеркнутое начертание шрифта.

Поле lfCharSet определяет таблицу кодировки шрифта. Определены следующие таблицы: ANSI_CHARSET, DEFAULT_CHARSET, SYMBOL_CHARSET, SHIFTJIS_CHARSET, GB2312_CHARSET, HANGEUL_CHARSET, CHINESEBIG5_CHARSET, OEM_CHARSET.

В рамках Windows 95/98 дополнительно определены следующие таблицы: JOHAB_CHARSET, HEBREW_CHARSET, ARABIC_CHARSET, GREEK_CHARSET, TURKISH_CHARSET, THAI_CHARSET, EASTEUROPE_CHARSET, RUSSIAN_CHARSET, MAC_CHARSET, BALTIC_CHARSET.

Примечание

Значение OEM_CHARSET определяет таблицу кодировки конкретной версии операционной системы. Значение DEFAULT_CHARSET для идентификации шрифта требует задания имени (поле lfFaceName) и размера, однако отсутствие запрашиваемого шрифта может привести к непредсказуемым результатам, т. к. он будет подменен шрифтом, подходящим по другим параметрам, но неизвестной заранее кодировки.

Особенно важно задание правильной таблицы кодировки при указании имени шрифта в явном виде, т. к. именно кодировка *в первую очередь определяет набор установленных шрифтов*, среди которых будет произведен поиск по имени (рис. 18.3):

SYMBOL_CHARSET и "Times New Roman"	RUSSIAN_CHARSET и "Wingdings"
?yЩ±?, Ще?е??И ?eцце ??е?Ц?б??	Текст, который можно прочитать

Рис. 18.3. Пример доминирования заданной таблицы кодировки над именем шрифта

Поле lfOutPrecision определяет механизм выбора шрифта, т. е. насколько шрифт должен соответствовать заданным параметрам, прежде всего типу шрифта (растровый, системный или шрифт TrueType), а также ширине, высоте, ориентации и т. д. Это поле может принимать одно из следующих значений:

OUT_CHARACTER_PRECIS	Не используется
OUT_DEFAULT_PRECIS	Определяет механизм выбора, принятый по умолчанию
OUT_DEVICE_PRECIS	Указывает на то, чтобы в случае, когда в системе установлены несколько шрифтов с одним и тем же названием, был выбран системный шрифт
OUT_OUTLINE_PRECIS	Только для Windows NT: выбор осуществляется из набора векторных или шрифт TrueType
OUT_RASTER_PRECIS	Указывает на то, чтобы в случае, когда в системе установлены несколько шрифтов с одним и тем же названием, был выбран растровый шрифт
OUT_STRING_PRECIS	Используется для получения списка растровых шрифтов
OUT_STROKE_PRECIS	Используется для получения списка векторных и шрифтов TrueType
OUT_TT_ONLY_PRECIS	Задает выбор только шрифта TrueType, и если ни один такой шрифт не установлен, используется механизм выбора, принятый по умолчанию
OUT_TT_PRECIS	Указывает на то, чтобы в случае, когда в системе установлены несколько шрифтов с одним и тем же названием, был выбран шрифт TrueType

Поле lfClipPrecision определяет механизм отсечения символов шрифта в случае, если символы частично оказываются вне области вывода. Это поле может принимать одно или несколько из следующих значений:

CLIP_DEFAULT_PRECIS	Определяет использование механизма, принятого по умолчанию
CLIP_CHARACTER_PRECIS	Не используется
CLIP_STROKE_PRECIS	Используется только для получения списка растровых, векторных или шрифт TrueType
CLIP_MASK	Не используется
CLIP_EMBEDDED	Необходимо указывать этот флаг при использовании внедренных шрифтов
CLIP_LH_ANGLES	При использовании этого флага отсчет направления угла поворота (по часовой стрелки или против) зависит от выбранной системы координат; если флаг не используется, то поворот текста, выводимого с использованием системных шрифтов, производится против часовой стрелки, однако для всех остальных шрифтов направление поворота определяется системой координат
CLIP_TT_ALWAYS	Не используется

Поле lfQuality определяет желаемое качество вывода, которое зависит от выбранного шрифта. Определены следующие три значения качества вывода:

DEFAULT_QUALITY	Качество вывода текста не критично
DRAFT_QUALITY	Качество вывода текста менее важно, чем для PROOF_QUALITY; для растровых шрифтов разрешено масштабирование; при необходимости возможно создание синтетических начертаний шрифтов — полужирного, курсивного, подчеркнутого и перечеркнутого
PROOF_QUALITY	Качество вывода текста более важно, чем соответствие заданным атрибутам; для растровых шрифтов масштабирование запрещено, выбирается шрифт наиболее подходящий по размеру; при необходимости можно создавать синтетические начертания шрифтов — полужирное, курсивное, подчеркнутое и перечеркнутое

Поле lfPitchAndFamily определяет тип и семейство шрифтов. Это поле позволяет выбрать шрифт, наиболее похожий на заданный по имени в случае, если последний не установлен в системе. Для задания типа и семейства шрифта используется операция OR (|) между определениями типа и семейства соответственно.

Тип шрифта определяется двумя младшими битами поля и может принимать одно из следующих значений:

DEFAULT_PITCH	Тип, принятый по умолчанию
FIXED_PITCH	Непропорциональный шрифт, т. е. все символы шрифта имеют одинаковый размер
VARIABLE_PITCH	Пропорциональный шрифт

Семейство определяется старшими четырьмя битами поля и может принимать одно из следующих значений:

FF_DECORATIVE	Декоративный шрифт, например, Old English
FF_DONTCARE	Семейство шрифта не имеет значения
FF_MODERN	Непропорциональный шрифт с засечками или без них, например CourierNew®
FF_ROMAN	Пропорциональный шрифт с засечками, например MS® Serif
FF_SCRIPT	Шрифт, схожий по начертанию с рукописным, например Script или Cursive
FF_SWISS	Пропорциональный шрифт без засечек, например MS® Sans Serif

Поле lfFaceName содержит имя шрифта. Длина имени не может быть более 32 символов, включая завершающий нуль-символ. Если имя не указано или шрифт с таким именем не найден, поиск подходящего шрифта будет проводиться с использованием других заданных параметров. Однако в ряде случаев имя не является определяющим (см. комментарии к полю lfCharSet).

Создать шрифт можно также с помощью функции CreateFont, которая принимает в качестве параметров те же значения, что и поля структуры LOGFONT:

```
BOOL CFont::CreateFont(
    int nHeight, int nWidth, int nEscapement,
    int nOrientation, int nWeight, BYTE bItalic,
```

BYTE bUnderline, BYTE cStrikeOut,
BYTE nCharSet, BYTE nOutPrecision,
BYTE nClipPrecision, BYTE nQuality,
BYTE nPitchAndFamily,
LPCTSTR lpszFacename)

В ряде случаев при выборе шрифта нет необходимости столь подробно указывать его ожидаемые параметры. В то же время использование большинства текстовых редакторов приучает к тому, что размер шрифта характеризуется вполне точным, если хотите, абсолютным значением — размер задается в пунктах. А пункты, как вы помните, — вовсе не экранные точки, т. е. пиксели, и их размер известен — 1/72 дюйма. Ранее мы приводили формулу для вычисления размера в пикселях на основе размера в пунктах. Класс CFont имеет в своем составе функцию, которая, с одной стороны, создает шрифт на основе только его имени, а с другой — устанавливает желаемый размер как раз в привычных всем единицах:

BOOL CFont::**CreatePointFont**(
int nPointSize,
LPCTSTR lpszFaceName,
CDC* pDC = NULL) —

параметр nPointSize определяет величину шрифта в десятых долях пункта, например, при выборе шрифта с размером символов в 8 пунктов необходимо установить значение этого параметра в 80.

Имя шрифта задается параметром lpszFaceName, нулевое значение которого приводит к выбору системно-независимого шрифта.

Последний параметр pDC нужен для правильного преобразования размера шрифта в логические единицы, которые определяются именно для контекста устройства. Нулевое значение параметра приводит к выбору в качестве контекста устройства контекст экрана и, соответственно, его логических единиц.

Аналог функции CreateFontIndirect — функция CreatePointFontIndirect — также в качестве параметра принимает указатель на структуру LOGFONT, однако единицами измерения поля lfHeight этой структуры являются точки.

В качестве примера приведем функцию, которая выводит строку "Visual C++" под разными углами (рис. 18.4):

```
void CCreateFontFrame::OnPaint()
{
  CPaintDC dc(this);
  CRect rect;
  GetClientRect(rect);
  CFont font;
  LOGFONT lf = {
    32, 0, 0, 0, FW_NORMAL, 0, 0, 0, DEFAULT_CHARSET,
    OUT_TT_ONLY_PRECIS, CLIP_DEFAULT_PRECIS, PROOF_QUALITY,
```

```
    VARIABLE_PITCH | FF_DONTCARE, NULL
  };
  // Устанавливаем режим "прозрачного" вывода
  dc.SetBkMode(TRANSPARENT);
  for (int angle = 0; angle < 3600; angle += 450)
  {
    //
    // Задаем угол наклона всей строки
    //
    lf.lfEscapement = angle;
    lf.lfOrientation = angle;
    font.CreateFontIndirect(&lf);
    // Назначаем текущим ранее подготовленный шрифт
    CFont *pFont = dc.SelectObject(&font);
    dc.TextOut(rect.Width() / 2, rect.Height() / 2,
      _T("    Visual C++"));
    dc.SelectObject(pFont);
    font.DeleteObject();
  }
}
```

Рис. 18.4. Результат работы функции CCreateFontFrame::OnPaint

Битовый массив (класс CBitmap)

Класс CBitmap инкапсулирует графический объект Windows "битовый массив". Для работы с объектом класса необходимо создать сам объект, затем создать собственно битовый массив (CreateBitmap, CreateBitmapInderect, CreateCompatibleBitmap или CreateDiscardableBitmap), в том числе путем загрузки соответствующего ресурса (LoadBitmap, LoadOEMBitmap или LoadMappedBitmap), или установить связь с ранее созданным (загруженным) битовым массивом (Attach). Кроме этого, класс предоставляет набор функций, позволяющих менять и узнавать содержимое (SetBitmapBits и GetBitmapBits) и размеры (SetBitmapDimension и GetBitmapDimension) массива.

Сначала рассмотрим функции загрузки битового массива, созданного в качестве ресурса, а затем перейдем к функциям создания массивов непосредственно в памяти:

BOOL CBitmap::**LoadBitmap**(
LPCTSTR lpszResourceName)

или

BOOL CBitmap::**LoadBitmap**(
UINT nIDResource) —

позволяют загрузить битовые массивы, заданные текстовой строкой (параметр lpszResourceName), т. е. по имени, или числовым идентификатором (параметр nIDResource). Обе эти функции позволяют загружать битовые массивы из ресурсов исполняемого модуля текущего приложения.

Примечание

В общем случае, загрузка ресурсов осуществляется при явном указании дескриптора программного модуля. В приведенном ниже примере мы использовали дескриптор приложения.

Для загрузки системных битовых массивов служит функция LoadOEMBitmap:

BOOL CBitmap::**LoadOEMBitmap**(
UINT nIDBitmap) —

в качестве идентификатора ресурса может принимать одно из следующих значений, определенных в файле WINUSER.H:

OBM_LFARROW	OBM_LFARROWI	OBM_LFARROWD	OBM_OLD_LFARROW
OBM_RGARROW	OBM_RGARROWI	OBM_RGARROWD	OBM_OLD_RGARROW
OBM_DNARROW	OBM_DNARROWI	OBM_DNARROWD	OBM_OLD_DNARROW

OBM_UPARROW	OBM_UPARROWI	OBM_UPARROWD	OBM_OLD_UPARROW
OBM_COMBO			
OBM_MNARROW			
OBM_RESTORED	OBM_RESTORE		OBM_OLD_RESTORE
OBM_ZOOMD	OBM_ZOOM		OBM_OLD_ZOOM
OBM_REDUCED	OBM_REDUCE		OBM_OLD_REDUCE
OBM_CLOSE	OBM_OLD_CLOSE		
OBM_BTNCORNERS			
OBM_CHECKBOXES			
OBM_CHECK			
OBM_BTSIZE	OBM_SIZE		

Идентификаторы, начинающиеся с OBM_OLD, связаны с пустыми битовыми массивами. Для того чтобы получить доступ к вышеперечисленным идентификаторам, в файл STDAFX.H необходимо включить следующую строку:

```
#define OEMRESOURCE
```

В приведенном ниже примере отображаются все системные битовые массивы кроме "старых" (рис. 18.5). Следует иметь в виду, что размеры всех массивов определяются текущими установками и их размеры невозможно получить при помощи функции GetBitmapDimension, т. к. последняя возвращает размеры, только если ранее они были установлены при помощи функции SetBitmapDimension.

```
struct { int m_nBitmap; LPTSTR m_csBitmap; } btBitmap[] = {
  OBM_LFARROWI,   _T("OBM_LFARROWI"),
  OBM_LFARROWD,   _T("OBM_LFARROWD"),
  OBM_LFARROW,    _T("OBM_LFARROW"),
  OBM_RGARROWI,   _T("OBM_RGARROWI"),
  OBM_RGARROWD,   _T("OBM_RGARROWD"),
  OBM_RGARROW,    _T("OBM_RGARROW"),
  OBM_DNARROWI,   _T("OBM_DNARROWI"),
  OBM_DNARROWD,   _T("OBM_DNARROWD"),
  OBM_DNARROW,    _T("OBM_DNARROW"),
  OBM_UPARROWI,   _T("OBM_UPARROWI"),
  OBM_UPARROWD,   _T("OBM_UPARROWD"),
  OBM_UPARROW,    _T("OBM_UPARROW"),
  OBM_COMBO,      _T("OBM_COMBO"),
  OBM_MNARROW,    _T("OBM_MNARROW"),
  OBM_RESTORED,   _T("OBM_RESTORED"),
  OBM_RESTORE,    _T("OBM_RESTORE"),
```

```
  OBM_ZOOMD,       _T("OBM_ZOOMD"),
  OBM_ZOOM,        _T("OBM_ZOOM"),
  OBM_REDUCED,     _T("OBM_REDUCED"),
  OBM_REDUCE,      _T("OBM_REDUCE"),
  OBM_BTNCORNERS, _T("OBM_BTNCORNERS"),
  OBM_CHECK,       _T("OBM_CHECK"),
  OBM_BTSIZE,      _T("OBM_BTSIZE"),
  OBM_SIZE,        _T("OBM_SIZE"),
  OBM_CHECKBOXES, _T("OBM_CHECKBOXES"),
  OBM_CLOSE,       _T("OBM_CLOSE") };
...
void COEMBitmapFrame::OnPaint()
{
  CPaintDC dc(this);
  TEXTMETRIC tm;
  dc.GetTextMetrics(&tm);
  // Устанавливаем режим "прозрачного" вывода
  dc.SetBkMode(TRANSPARENT);
  int nY = 10;
  int nX = 10;
  CBitmap bitmap;
  for (int i = 0; i < sizeof(btBitmap) / sizeof(btBitmap[0]); i++)
  {
    // Загружаем системный битовый массив
    bitmap.LoadOEMBitmap(btBitmap[i].m_nBitmap);
    //
    // Получаем размеры, заполняя структуру
    // BITMAP, которая определена в Win32 API
    // для описания битовых массивов
    //
    BITMAP bmp;
    // Первый способ - используем функцию GetObject
    bitmap.GetObject(sizeof(BITMAP), &bmp);
    // второй способ - используем функцию GetBitmap
    // bitmap.GetBitmap(&bmp);
    CSize sz(bmp.bmWidth, bmp.bmHeight);
    dc.DrawState(CPoint(nX, nY), sz,
      bitmap, DST_BITMAP);
    dc.TextOut(nX+60,
      nY-tm.tmInternalLeading, btBitmap[i].m_csBitmap);
```

```
    // Сдвиг вниз на 5 единиц (в данном случае
    // пикселей) больше, чем максимальная высота
    // битового массива или соответствующего текста
    nY += max(sz.cy, tm.tmHeight) + 5;
    bitmap.DeleteObject();
    // Выводим изображения и текст в две колонки
    if (i == sizeof(btBitmap) / sizeof(btBitmap[0]) / 2)
    {
      nX += 200;
      nY = 10;
    }
  }
}
```

Рис. 18.5. Результат работы функции COEMBitmapFrame::OnPaint

Обратите внимание, что именно эти значки используются в Windows для рисования многих элементов управления. Причем, при смене параметров функционирования графической системы меняются и эти элементы управления (их размеры и цвет), так что приложения, пользующиеся системными битовыми массивами, могут так же легко приспосабливаться к изменившимся условиям!

Далее мы рассмотрим два примера, задача которых состоит в создании битовых массивов прямо в памяти, путем "задания" картинки в массивах данных. Но перед этим рассмотрим функции-члены класса CBitmap, которые используются в примерах.

BOOL CBitmap::**CreateCompatibleBitmap**(
CDC* pDC,
int nWidth,
int nHeight) —

предназначена для создания битовых массивов, совместимых с конкретным устройством вывода. Windows поддерживает два вида растровых изображений: независимых от устройства (Device Independent Bitmap) и, соответственно, зависимых. Данная функция создает битовые массивы последнего типа, но т. к. изображение готовится к выводу на конкретном устройстве — это вполне нормально. В качестве параметров функция принимает указатель на объект класса CDC (параметр pDC), для которого и создается изображение, параметры nWidth и nHeight определяют ширину и высоту соответственно. После выполнения этой функции битовый массив заполнен случайными значениями цветов для каждой точки. Такие битовые массивы могут быть либо заполнены необходимыми значениями для каждой точки, либо они могут являться основой контекстов устройств, расположенных в памяти. Для такого контекста выполнимы все функции рисования.

Другая функция создает битовый массив для вполне конкретного устройства:

BOOL CBitmap::**CreateBitmap**(
int nWidth,
int nHeight,
UINT nPlanes,
UINT nBitcount,
const void* lpBits) —

параметры: nWidth и nHeight, как и для предыдущей функции, определяют размеры изображения; *nPlanes* и *nBitCount* определяют параметры хранения информации о цвете: количество слоев и количество битов для хранения одного цвета; lpBits является указателем на массив, в котором хранятся значения цветов для каждой точки. К сожалению, формат хранения такой информации строго зависит от устройства вывода.

В функции CCreateBmpOne::OnPaint мы реализовали два способа создания битового массива:

- Созданный битовый массив, совместимый с контекстом устройства окна, заполняется ранее подготовленными значениями цветов (функция SetBitmapBits)
- Создание и инициализация битового массива производятся одновременно

Недостаток этих подходов в том, что в обоих случаях предполагается, что мы имеем дело с устройством, требующим для хранения цвета одного пикселя 16 бит, причем 5 бит отводится под красную составляющую, 6 бит — под зеленую и 5 бит — под синюю. Созданный таким образом битовый массив не будет верно отображаться на устройствах с другой организацией (рис. 18.6 и 18.7).

```
void CCreateBmpOneFrame::OnPaint()
{
  CPaintDC dc(this);
  int CX = 400, CY = 100;
  // Первое допущение: надеемся, что количество бит,
```

```
  // необходимых для данных типа WORD, — 16
  WORD *lpvBits = new WORD[CX * CY];
  for (int i = 0; i < CY;?i++)
  {
    for (int j = 0; j < CX; j++)
    {
      WORD color = 31-abs(i*31/(CY/2)-31);
      // Второе допущение: надеемся, что устройство вывода
      // требует для хранения цвета 16 бит, причем
      // для красного - старшие 5
      // для зеленого - средние 6
      // для синего   - младшие 5
      lpvBits[i*CX+j] =
        (WORD)((color << 11) + (color*2 << 5) + color);
    }
  }
  CBitmap bitmap;
  // Первый вариант создания битового массива
  bitmap.CreateCompatibleBitmap(&dc, CX, CY);
  bitmap.SetBitmapBits(sizeof(WORD) * CX * CY, lpvBits);
  // Второй вариант создания битового массива
  // bitmap.CreateBitmap(CX, CY, 1, 16, lpvBits);
  // Надеемся - что-нибудь будет выведено
  dc.DrawState(CPoint(10, 10),
    CSize(CX, CY), &bitmap, DST_BITMAP);
  delete lpvBits;
}
```

Рис. 18.6. Результат работы функции CCreateBmpOneFrame::OnPaint — система требует 16 бит на цвет

В функции CCreateBmpTwo::OnPaint реализован второй способ создания битового массива. В примере используется функция Win32 API CreateDIBitmap, которая принимает в качестве информации о цвете точки

RGB-тройки, что существенно упрощает программирование и делает код независимым от конкретных настроек видеосистемы (рис. 18.8 и 18.9).

Рис. 18.7. Результат работы функции CCreateBmpOneFrame::OnPaint — система требует 8 бит на цвет

```
void CCreateBmpTwo::OnPaint()
{
  CPaintDC dc(this);
  int CX = 400, CY = 100;
  DWORD *lpbInit = new DWORD[CX * CY];
  for (int i = 0; i < CY; i++)
  {
    for (int j = 0; j < CX; j++)
    {
      BYTE color = (BYTE)(255-abs(i*255/(CY/2)-255));
      lpbInit[i*CX+j] = RGB(color, color, color);
    }
  }
  BITMAPINFO bmi = {
    sizeof(BITMAPINFOHEADER), CX, CY, 1, 32, BI_RGB, 0,
    0, 0, 0, 0
  };
  HBITMAP hBitmap = ::CreateDIBitmap(
    dc.GetSafeHdc(), &bmi.bmiHeader, CBM_INIT,
    lpbInit, &bmi, DIB_RGB_COLORS);
  CBitmap bitmap;
  bitmap.Attach(hBitmap);
  dc.DrawState(CPoint(10, 10), CSize(CX, CY),
    &bitmap, DST_BITMAP);
  bitmap.DeleteObject();
  delete lpbInit;
}
```

Рис. 18.8. Результат работы функции CCreateBmpTwoFrame::OnPaint — система требует 16 бит на цвет

Рис. 18.9. Результат работы функции CCreateBmpTwoFrame::OnPaint — система требует 8 бит на цвет

Регион (класс CRgn)

Класс CRgn инкапсулирует объект Windows "регион", который может представлять собой совокупность прямоугольных и эллиптических областей. Основное назначение регионов — определение области отсечения в пределах окна. Класс CRgn имеет в своем составе набор функций для создания регионов, схожих по своим параметрам с функциями рисования прямоугольников, эллипсов и многоугольников, а также функции, позволяющие комбинировать регионы.

Создание регионов

Для создания региона при помощи класса CRgn необходимо использовать конструктор без параметров, который создает объект класса CRgn, но не создает при этом собственно объект Windows "регион". Для создания последнего можно воспользоваться функциями, описанными ниже.

Создание прямоугольных регионов

```
BOOL CRgn::CreateRectRgn(
    int x1,
    int y1,
```

int x2,
int y2)

и

BOOL CRgn::**CreateRectRgnIndirect**(
LPCRECT lpRect) —

первая функция получает в качестве параметров координаты левого верхнего и правого нижнего углов прямоугольной области, вторая — указатель на структуру RECT, в которой также указываются координаты прямоугольной области.

Создание эллиптических регионов

BOOL CRgn::**CreateEllipticRgn**(
int x1,
int y1,
int x2,
int y2)

и

BOOL CRgn::**CreateEllipticRgnIndirect**(
LPCRECT lpRect) —

обе функции получают информацию аналогично функциям предыдущей группы, о прямоугольной области, в которую вписан эллипс.

Создание сложных регионов на базе многоугольников

BOOL CRgn::**CreatePolygonRgn**(
LPPOINT lpPoints,
int nCount,
int nPolyFillMode) —

принимает в качестве информации о прямоугольнике массив координат (lpPoints) и количество его вершин (nCount).

BOOL CRgn::**CreatePolyPolygonRgn**(
LPPOINT lpPoints,
LPINT lpPolyCounts,
int nCount,
int nPolyFillMode) —

принимает в качестве информации о прямоугольниках массив координат их вершин (lpPoints) и массив, содержащий количество вершин для каждого из них (lpPolyCounts). Параметр nCount содержит количество многоугольников.

Параметр nPolyFillMode в этих функциях определяет режим закраски ALTERNATE или WINDING (см. главу 17).

Создание регионов с закругленными углами

BOOL CRgn::**CreateRoundRectRgn**(
int x1,
int y1,
int x2,
int y2,
int x3,
int y3) —

первые четыре параметра определяют прямоугольную область, а x3 и y3 — ширину и высоту эллипса, на основе которого строятся закругленные углы.

Комбинирование регионов

Более подробно остановимся на функции, которая на основе комбинации двух регионов изменяет третий. Именно *изменяет*, т. к. объект класса, для которого вызывается эта функция и который собственно и будет хранить результат комбинации двух других регионов, обязательно должен существовать — так работает Windows, так работает и библиотека классов MFC.

Примечание

Объект класса CRgn, для которого вызывается функция, может совпадать с одним из тех, указатели на которые передаются в качестве параметров.

int CRgn::**CombineRgn**(
CRgn* pRgn1,
CRgn* pRgn2,
int nCombineMode) —

возвращает одно из следующих значений, показывающих, какого типа регион получился в результате комбинирования:

COMPLEXREGION	Регион состоит из набора более простых областей
ERROR	Произошла ошибка
NULLREGION	Регион пуст
SIMPLEREGION	Регион представляет собой одну простую область — прямоугольник или эллипс

Параметры pRgn1 и pRgn2 являются указателями на объекты класса CRgn, которые должны быть связаны с регионами Windows, именно они и будут комбинироваться. Параметр nCombineMode определяет режим комбинирования. Таких режимов несколько, мы уже упоминали их в предыдущей главе, но в несколько другом контексте, поэтому перечислим их еще раз:

RGN_AND	Регион будет пересечением обоих заданных, т. е. определяется областью, которая относится и к первому и ко второму региону одновременно
RGN_COPY	Регион будет точной копией первого региона (pRgn1)

RGN_DIFF	Регион будет включать в себя только те области первого региона (pRgn1), которые не пересекаются со вторым (pRgn2)
RGN_OR	Регион будет объединением обоих заданных, т. е. включает все области, относящиеся и к первому и ко второму региону
RGN_XOR	Регион будет включать области первого (pRgn1) и второго регионов (pRgn2) кроме тех, которые принадлежат и первому и второму одновременно

Несмотря на то, что Windows требует наличия объектов для всех трех регионов для осуществления комбинирования, возможно использование в качестве одного или даже обоих параметров указателя на объект класса CRgn, для которого и вызывается функция CombineRgn:

```
rgn.CombineRgn(&rgn, &rgnPrev);
```

Кроме вышеперечисленных функций-членов, упомянем еще несколько. Класс CRgn имеет в своем составе функцию CopyRgn для копирования, функцию CreateFromPath для преобразования текущего контура (path) в регион и функцию EqualRgn для сравнения двух регионов.

В качестве примера использования регионов приведем функцию, в которой в результате комбинирования трех регионов производится закрашивание части замкнутых областей одним вызовом функции рисования (рис. 18.10).

```
void CRegionFrame::OnPaint()
{
  CPaintDC dc(this);
  CRect rect;
  GetClientRect(rect);
  // Определяем треугольную область с одной вершиной
  // в центре клиентской части окна и двумя другими,
  // совпадающими с левым верхним и левым нижним углами
  POINT pt_1[4] = { 0, 0,
                    rect.right / 2, rect.bottom / 2,
                    0, rect.bottom,
                    0, 0 };
  CRgn rgn_1;
  rgn_1.CreatePolygonRgn(pt_1, 4, WINDING);
  // Определяем треугольную область с одной вершиной
  // в центре клиентской части окна и двумя другими,
  // совпадающими с правым верхним и правым нижним углами
  POINT pt_2[4] = { rect.right, 0,
                    rect.right / 2, rect.bottom / 2,
                    rect.right, rect.bottom,
                    rect.right, 0 };
```

```
  CRgn rgn_2;
  rgn_2.CreatePolygonRgn(pt_2, 4, WINDING);
  // Определяем прямоугольную область, лежащую точно в
  // центре клиентской области окна и немного меньше ее по размеру
  CRgn rgn;
  rgn.CreateRectRgn(40, 40, rect.right-40, rect.bottom-40);
  // Определяем регион как комбинацию трех заданных
  rgn.CombineRgn(&rgn, &rgn_1, RGN_XOR);
  rgn.CombineRgn(&rgn, &rgn_2, RGN_XOR);
  //
  // Определяем параметры рисования
  //
  // Кисть
  CBrush brush(HS_DIAGCROSS, RGB(0, 0, 0));
  CBrush *pBrush = dc.SelectObject(&brush);
  // Цвет фона
  dc.SetBkColor(RGB(128, 128, 128));
  // Область отсечения
  dc.SelectObject(&rgn);
  // Рисуем прямоугольник на всю клиентскую часть окна
  dc.Rectangle(rect);
  dc.SelectObject(pBrush);
}
```

Рис. 18.10. Результат работы функции CRegionFrame::OnPaint

К сожалению, даже на таком простом примере при выполнении программы видно, как медленно происходит рисование — складывается впечатление, что для каждой точки проверяется возможность ее закрашивания.

PQClock — пример приложения

Этот раздел полностью посвящен исходным текстам приложения, задача которого отображать текущее время в виде аналоговых часов (т. е. часов с циферблатом и со стрелками). Естественно, мы преследовали несколько иную основную цель. А именно — показать практическое использование большей части того, чему были посвящены три последние главы.

Ниже приводится лишь основная часть исходных текстов приложения-примера. Это два файла (заголовочный и файл кода) класса приложения CPQClockApp и два файла (заголовочный и файл кода) класса главного окна приложения CClockFrame. Большое количество комментариев позволит лучше и быстрее понять суть примера, а также используемые классы и алгоритмы.

Класс приложения

```
/************************************************************
Файл PQClock.h
Copyright (c) 1999 Мешков А., Тихомиров Ю.
************************************************************/
// PQClock.h : main header file for the PQClock application
//
#ifndef __AFXWIN_H__
  #error include 'stdafx.h' before including this file for PCH
#endif
#include "resource.h"    // main symbols
/////////////////////////////////////////////////////////////////////////////
// CPQClockApp:
// See PQClock.cpp for the implementation of this class
//
class CPQClockApp : public CWinApp
{
public:
  CPQClockApp();
// Overrides
  // ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CPQClockApp)
  public:
  virtual BOOL InitInstance();
  virtual BOOL OnIdle(LONG lCount);
  //}}AFX_VIRTUAL
```

```
// Implementation
  //{{AFX_MSG(CPQClockApp)
    // NOTE - the ClassWizard will add and
    // remove member functions here.
    //    DO NOT EDIT what you see in these
    //    blocks of generated code !
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()
};
/////////////////////////////////////////////////////////////////////////////
/*************************************************************
Файл PQClock.cpp
Copyright (c) 1999 Мешков А., Тихомиров Ю.
*************************************************************/
// PQClock.cpp : Defines the class behaviors for the application.
//
#include "stdafx.h"
#include "PQClock.h"
#include "ClockFrm.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
/////////////////////////////////////////////////////////////////////////////
// CPQClockApp
BEGIN_MESSAGE_MAP(CPQClockApp, CWinApp)
  //{{AFX_MSG_MAP(CPQClockApp)
    // NOTE - the ClassWizard will add and
    // remove mapping macros here.
    //    DO NOT EDIT what you see in these
    //    blocks of generated code!
  //}}AFX_MSG
  ON_COMMAND(ID_HELP, CWinApp::OnHelp)
END_MESSAGE_MAP()
/////////////////////////////////////////////////////////////////////////////
// CPQClockApp construction
CPQClockApp::CPQClockApp()
{
  // TODO: add construction code here,
```

```
  // Place all significant initialization in InitInstance
}
/////////////////////////////////////////////////////////////////////////////
// The one and only CPQClockApp object
CPQClockApp theApp;
/////////////////////////////////////////////////////////////////////////////
// CPQClockApp initialization
BOOL CPQClockApp::InitInstance()
{
  // Standard initialization
  // If you are not using these features and
  // wish to reduce the size
  // of your final executable,
  // you should remove from the following
  // the specific initialization routines you do not need.
#ifdef _AFXDLL
  Enable3dControls();    // Call this when using MFC in a shared DLL
#else
  Enable3dControlsStatic();  // Call this when linking to MFC
                             // statically
#endif
  CClockFrame *pWnd = new CClockFrame;
  m_pMainWnd = pWnd;
  int nSize = min(GetSystemMetrics(SM_CXMAXIMIZED),
    GetSystemMetrics(SM_CYMAXIMIZED)) * 3 / 5;
  pWnd->Create(NULL, "Часы 'PRAGMA Quartz'",
    WS_OVERLAPPEDWINDOW, CRect(0, 0, nSize, nSize));
  pWnd->CenterWindow();
  pWnd->ShowWindow(m_nCmdShow);
  pWnd->UpdateWindow();
  return TRUE;
}
BOOL CPQClockApp::OnIdle(LONG lCount)
{
  CWinApp::OnIdle(lCount);
  // Перерисовываем окно
  ((CClockFrame *)m_pMainWnd)->Redraw();
  // Чтобы не заканчивать цикл вызова
  // функции OnIdle, возвращаем TRUE
  return TRUE;
}
```

Обратите внимание, что именно в функции OnIdle вызывается процедура перерисовки часов — при этом не задействуется таймер и не добавляется код в функцию WinMain, как это пришлось бы сделать в рамках Win32 API.

Вот как выглядит окно приложения, функции собственно отображения которого приведены далее (рис. 18.11):

Рис. 18.11. Окно приложения PQClock при начальных размерах окна

Класс окна

```
/************************************************************
Файл ClockFrm.h
Copyright (c) 1999 Мешков А., Тихомиров Ю.
************************************************************/
// ClockFrm.h : header file
//
/////////////////////////////////////////////////////////////////////////////
// CClockFrame frame
class CClockFrame : public CFrameWnd
{
public:
  CClockFrame();
// Attributes
public:
// Operations
```

```
public:
  void Redraw();
private:
  void RotatePoint(POINT pt[], int nNum, double nAngle);
  void SetMode(CDC *pDC);
  void DrawClock(CDC *pDC);
  void DrawLabel(CDC *pDC);
  void DrawArrows(CDC *pDC, CTime &tm);
// Overrides
  // ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CClockFrame)
  //}}AFX_VIRTUAL
// Implementation
protected:
  virtual ~CClockFrame();
  // Generated message map functions
  //{{AFX_MSG(CClockFrame)
  afx_msg void OnSize(UINT nType, int cx, int cy);
  afx_msg void OnPaint();
  afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()
private:
  int m_cxClient;
  int m_cyClient;
  CFont m_Font_1;
  CFont m_Font_2;
  CFont m_Font_3;
  CBrush m_wndBrush;
  CPen m_wndPen;
  CBrush m_arrBrush;
  CPen m_arrPen;
  CBrush m_clkBrush;
  CPen m_clkPen;
  CBrush m_rndBrush;
  CPen m_rndPen;
  CBrush m_crnBrush;
  CPen m_crnPen;
};
```

```
/////////////////////////////////////////////////////////////////
/*************************************************************
Файл ClockFrm.cpp
Copyright (c) 1999 Мешков А., Тихомиров Ю.
*************************************************************/
// ClockFrm.cpp : implementation file
//
#include "stdafx.h"
#include "PQClock.h"
#include "ClockFrm.h"
#include "math.h"
// Для возможности быстрого изменения цветов
// определим их в одном месте -
// в начале файла
#define COLOR_BCKGR   RGB(224,224,224)
#define COLOR_WHITE   RGB(255,255,255)
#define COLOR_BLACK   RGB(  0,  0,  0)
#define COLOR_ARROW   RGB(164,164,  0)
#define COLOR_CLOCK   RGB(164,164,  0)
#define COLOR_ROUND   RGB( 92, 92,  0)
#define COLOR_CRNER   RGB( 92, 92,  0)
#define TWOPI         (2 * 3.14159)
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
/////////////////////////////////////////////////////////////////
// CClockFrame
CClockFrame::CClockFrame()
{
}
CClockFrame::~CClockFrame()
{
}
//
// Функция вычисления текущих координат точек
// в зависимости от угла поворота
//
```

```
void CClockFrame::RotatePoint(POINT pt[], int nNum, double nAngle)
{
  POINT ptTemp;
  // Поворот осуществляется вокруг начала координат
  for (int i = 0; i < nNum; i++)
  {
    ptTemp.x = (int)(pt[i].x * cos (TWOPI * nAngle / 360) +
                     pt[i].y * sin (TWOPI * nAngle / 360));
    ptTemp.y = (int)(pt[i].y * cos (TWOPI * nAngle / 360) -
                     pt[i].x * sin (TWOPI * nAngle / 360));
    pt[i] = ptTemp;
  }
}
//
// Функция формирования системы координат
//
void CClockFrame::SetMode(CDC *pDC)
{
  // Устанавливаем масштаб, начало и направления
  // координатных осей
  pDC->SetMapMode(MM_ISOTROPIC);
  pDC->SetWindowExt(1000, 1000);
  pDC->SetViewportExt(m_cxClient / 2, -m_cyClient / 2);
  pDC->SetViewportOrg(m_cxClient / 2,  m_cyClient / 2);
  // В результате:
  //   начало координат в середине клиентской области окна
  //   ось "x" направлена влево
  //   ось "y" направлена вверх
  //   размер окна 2000x2000 логических единиц
  //     X: [-1000; +1000]
  //     Y: [-1000; +1000]
}
//
// Функция рисования циферблата часов
//
void CClockFrame::DrawClock(CDC *pDC)
{
  CPen *pPen;
  CBrush *pBrush;
```

```
int nAngle;
//
// Рисуем отметки для каждого часа,
// причем для 12, 3, 6 и 9 часов
// отметки — прямоугольники,
// для остальных — круги
//
pBrush = pDC->SelectObject(&m_clkBrush);
for (nAngle = 0; nAngle < 360; nAngle += 30)
{
  POINT pt[2] = { 0, 900 };
  RotatePoint(pt, 1, nAngle);
  int x = (nAngle % 180) ? 70 : 250;
  int y = ((nAngle+90) % 180) ? 70 : 250;
  pt[0].x -= x / 2;
  pt[0].y -= y / 2;
  pt[1].x  = pt[0].x + x;
  pt[1].y  = pt[0].y + y;
  if (nAngle % 90)
    pDC->Ellipse(pt[0].x, pt[0].y, pt[1].x, pt[1].y);
  else
  {
    pDC->RoundRect(pt[0].x, pt[0].y, pt[1].x, pt[1].y,
      min(x, y) / 2, min(x, y) / 2);
  }
}
//
// Рисуем дуги между часовыми отметками
// так, чтобы их диаметр был немного
// больше окружности, на которой расположены
// часовые отметки
//
pPen = pDC->SelectObject(&m_rndPen);
POINT p?Prev;
for (nAngle = -30; nAngle < 360; nAngle += 30)
{
  POINT pt[2] = { -45, 950, 45, 950 };
  RotatePoint(pt, 2, nAngle);
  if (nAngle != -30)
```

```
    {
      pDC->Arc(-980, -980, 980, 980,
        pt[0].x, pt[0].y, ptPrev.x, ptPrev.y);
    }
    ptPrev = pt[1];
  }
  //
  // Рисуем "углы" часов, причем
  // четырьмя различными способами
  //
  CRect round(-1100, -1100, 1100, 1100);
  CRgn rgn;
  CRect rect;
  pDC->SelectObject(&m_crnPen);
  pDC->SelectObject(&m_crnBrush);

  //
  // I способ:
  // Рисуем левый верхний угол
  //
  rect = CRect(-950, 950, -550, 550);
  //
  // Формируем область отсечения
  // (для этого нужны координаты в единицах
  // устройства вывода)
  pDC->LPtoDP(&rect);
  rgn.CreateRectRgnIndirect(&rect);
  pDC->SelectObject(&rgn);
  // Рисуем замкнутую дугу (круг), но выводится
  // только ее часть в соответствии с заданной
  // областью отсечения
  pDC->Arc(&round, CPoint(0, 0), CPoint(0, 0));
  // Добавляем две линии для формирования
  // замкнутого контура
  pDC->MoveTo(-550, 950);
  pDC->LineTo(-950, 950);
  pDC->LineTo(-950, 550);
  // Разрешаем для рисования всю клиентскую часть окна
  pDC->SelectClipRgn(NULL);
```

```
  // Заполняем замкнутую область
  // от внутренней точки до границы
  pDC->ExtFloodFill(-900, 900, COLOR_CRNER, FLOODFILLBORDER);
  //
  // II способ:
  // Рисуем правый верхний угол
  //
  // Рисуем дугу и две линии для формирования
  // замкнутого контура
  pDC->Arc(&round, CPoint(950, 550), CPoint(550, 950));
  pDC->MoveTo(550, 950);
  pDC->LineTo(950, 950);
  pDC->LineTo(950, 550);
  // Заполняем замкнутую область, заданную цветом,
  // с внутренней точки
  pDC->ExtFloodFill(900, 900, COLOR_BCKGR, FLOODFILLSURFACE);
  //
  // III способ:
  // Рисуем правый нижний угол
  //
// Создаем два региона
  //
  // Первый - прямоугольный, в него
  // вписана окружность
  rect = CRect(550, -950, 950, -550);
  CRgn rgn1;
  rgn1.CreateRectRgnIndirect(&rect);
  // второй - эллиптический, он как раз
  // равен окружности
  CRect rect2 = round;
  CRgn rgn2;
  rgn2.CreateEllipticRgnIndirect(&rect2);
  // На основе созданных регионов создаем
  // комбинированный с нужной нам формой
  rgn.CombineRgn(&rgn1, &rgn2, RGN_DIFF);
  // Закрашиваем его
  pDC->PaintRgn(&rgn);
  // В явном виде объекты можно не удалять -
  // деструктор сделает все за нас!
```

```
  // rgn.DeleteObject();
  // rgn1.DeleteObject();
  // rgn2.DeleteObject();
  //
  // IV способ:
  // Рисуем левый нижний угол
  //
  // Начинаем формирование контура
  //
  pDC->BeginPath();
  // Массив точек для рисования
  // кривой Безье
  POINT pt[] = {
    -950, -550,
    -950, -670,
    -670, -950,
    -550, -950
  };
  // "Рисуем" кривую Безье, т. к. другие
  // функции рисования гладких кривых
  // не фиксируются в контуре
  pDC->PolyBezier(pt, 4);
  // Добавляем две линии для формирования
  // замкнутого контура
  pDC->MoveTo(-950, -550);
  pDC->LineTo(-950, -950);
  pDC->LineTo(-550, -950);
  pDC->EndPath();
  // Закрашиваем пространство, ограниченное
  // сформированным контуром
  pDC->FillPath();
  pDC->SelectObject(pPen);
  pDC->SelectObject(pBrush);
}
//
// Функция рисования строк текста на циферблате
//
void CClockFrame::DrawLabel(CDC *pDC)
{
```

```
  pDC->SetTextAlign(TA_CENTER);
  // Первые две строки рисуем так, чтобы
  // промежутки заполнялись цветом фона
  COLORREF color = pDC->SetBkColor(COLOR_BCKGR);
  CFont *pFont = pDC->SelectObject(&m_Font_1);
  pDC->TextOut(0,  600, _T("PRAGMA"), 6);
  TEXTMETRIC tm;
  pDC->GetTextMetrics(&tm);
  pDC->SelectObject(&m_Font_2);
  pDC->TextOut(0,  600-(tm.tmHeight+tm.tmExternalLeading),
    _T("QUARTZ"), 6);
  //
  // Последнюю строку рисуем так, чтобы
  // промежутки между контурами символов
  // оставались без изменения
  //
  // Цвет фона не оказывает влияния на эти области
  pDC->SetBkColor(color);
  pDC->SetBkMode(TRANSPARENT);
  pDC->SelectObject(&m_Font_3);
  pDC->TextOut(0, -500, _T("NO WATER RESIST"));
  pDC->SelectObject(pFont);
}
//
// Функция вывода стрелок установленными вне
// нее текущими "карандашом" и "кистью"
//
void CClockFrame::DrawArrows(CDC *pDC, CTime &tm)
{
  // Массив точек, которые определяют контуры стрелок
  static POINT pt[3][11] =
  {
      0,  -50, -50,    0, -5,   40, -5,  100, -30,  120,
      0,  600,  30,
    120,    5, 100,    5, 40,   50,  0,    0, -50,
      0,  -40, -40,    0, -5,   30, -5,   90, -20,  110,
      0,  800,
     20,  110,   5,   90,  5,   30, 40,    0,   0,  -40,
      0, -100,  -5, -100, -5, -100, -5, -100,  -5, -100,
```

```
      0,  850,
      5, -100,   5, -100,  5, -100,  5, -100,   0, -100
  };
  double nAngle[3];
  // Вычисление угла поворота для каждой из трех стрелок
  nAngle[0] = (tm.GetHour()   * 30) % 360 + tm.GetMinute() / 2;
  nAngle[1] = (tm.GetMinute() * 60 + tm.GetSecond()) / 10.0;
  nAngle[2] = (tm.GetSecond() *  6);
  POINT ptTemp[3][11];
  memcpy(ptTemp, pt, sizeof(pt));
  for (int i = 0; i < 3; i++)
  {
    // Вычисление текущих координат всех точек
    // контура стрелки
    RotatePoint(ptTemp[i], 11, nAngle[i]);
    // Рисование стрелки
    pDC->Polygon(ptTemp[i], 11);
  }
}
//
// Функция перерисовки часов, может вызываться
// практически в любой функции программы
//
void CClockFrame::Redraw()
{
  static CTime m_tmPrev;
  if (IsIconic())
    return;
  CClientDC dc(this);
  CTime tmCurrent = CTime::GetCurrentTime();
  if (tmCurrent != m_tmPrev)
  {
    // Устанавливаем нашу систему координат
    SetMode(&dc);
    // Рисуем циферблат
    DrawClock(&dc);
    // Закрашиваем цветом фона ранее выведенные стрелки
    CPen *pPen = dc.SelectObject(&m_wndPen);
    CBrush *pBrush = dc.SelectObject(&m_wndBrush);
```

```
    DrawArrows(&dc, m_tmPrev);
    // Выводим текст надписей
    DrawLabel(&dc);
    // Выводим стрелки по их новому месту
    dc.SelectObject(&m_arrPen);
    dc.SelectObject(&m_arrBrush);
    DrawArrows(&dc, tmCurrent);
    dc.SelectObject(pBrush);
    dc.SelectObject(pPen);
  }
  m_tmPrev = tmCurrent;
}
BEGIN_MESSAGE_MAP(CClockFrame, CFrameWnd)
  //{{AFX_MSG_MAP(CClockFrame)
  ON_WM_SIZE()
  ON_WM_PAINT()
  ON_WM_CREATE()
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()
/////////////////////////////////////////////////////////////////////////////
// CClockFrame message handlers
void CClockFrame::OnSize(UINT nType, int cx, int cy)
{
  CFrameWnd::OnSize(nType, cx, cy);
  // Запоминаем размер клиентской области
  // окна, чтобы каждый раз при перерисовке
  // не запрашивать размер заново
  m_cxClient = cx;
  m_cyClient = cy;
}
void CClockFrame::OnPaint()
{
  CPaintDC dc(this);
  // Закрашиваем клиентскую область
  CBrush brush(COLOR_BCKGR);
  dc.FillRect(
    CRect(0, 0, m_cxClient, m_cyClient), &brush);
  // Циферблат, текст и стрелки рисовать не будем,
  // этим займется функция CPQClock::OnIdle
```

```
}
int CClockFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
  if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
    return -1;
  //
  // Для экономии времени вместо того, чтобы при
  // рисовании создавать "шрифты", "карандаши" и "кисти",
  // сделаем это один раз.
  // При этом надо помнить, что захваченные таким
  // образом ресурсы будут освобождены при удалении
  // объекта класса CClockFrame - объекты соответствующих
  // классов являются членами класса CClockFrame
  //
  m_Font_1.CreateFont(130, 0, 0, 0, FW_BOLD,
    0, 0, 0, DEFAULT_CHARSET,
    OUT_DEFAULT_PRECIS, CLIP_DEFAULT_PRECIS, PROOF_QUALITY,
    VARIABLE_PITCH | FF_DONTCARE, "Arial");
  m_Font_2.CreateFont(100, 0, 0, 0, FW_BOLD,
    0, 0, 0, DEFAULT_CHARSET,
    OUT_DEFAULT_PRECIS, CLIP_DEFAULT_PRECIS, PROOF_QUALITY,
    VARIABLE_PITCH | FF_SWISS, "Arial");
  m_Font_3.CreateFont(70, 0, 0, 0, FW_BOLD,
    0, 0, 0, DEFAULT_CHARSET,
    OUT_DEFAULT_PRECIS, CLIP_DEFAULT_PRECIS, PROOF_QUALITY,
    VARIABLE_PITCH | FF_SWISS, "Arial");
  m_wndBrush.CreateSolidBrush(COLOR_BCKGR);
  m_wndPen.CreatePen(PS_SOLID, 1, COLOR_BCKGR);
  m_arrBrush.CreateHatchBrush(HS_DIAGCROSS, COLOR_ARROW);
  m_arrPen.CreatePen(PS_SOLID, 1, COLOR_BLACK);
  m_clkBrush.CreateSolidBrush(COLOR_CLOCK);
  m_clkPen.CreatePen(PS_SOLID, 1, COLOR_BLACK);
  m_rndBrush.CreateSolidBrush(COLOR_ROUND);
  //
  // Задаем прямоугольные концы для линий,
  // для которых мы создаем "карандаш", в нашем примере
  // не влияет на рисование, т. к. это свойство
  // проявляется лишь тогда, когда рисование
  // осуществляется при помощи создания контуров,
```

```
  // а данный "карандаш" будет использоваться для
  // рисования дуг, которые функцией CDC::StrokePath
  // не отображаются
  //
  LOGBRUSH lb;
  m_rndBrush.GetLogBrush(&lb);
  m_rndPen.CreatePen(PS_SOLID | PS_GEOMETRIC |
    PS_ENDCAP_SQUARE, 20, &lb);
  m_crnBrush.CreateSolidBrush(COLOR_CRNER);
  m_crnPen.CreatePen(PS_SOLID, 1, COLOR_CRNER);

  return 0;
}
```

Используемая методика рисования в режиме MM_ISOTROPIC позволяет не добавлять ни строчки кода для настройки или перенастройки параметров (прежде всего координат) при изменении размеров окна. Сравните следующий рисунок (рис. 18.12), с тем, что был приведен ранее (рис. 18.11) — размеры другие, а содержимое окна то же.

Примечание

Мы надеемся, что разное время на часах не введет вас в заблуждение.

Рис. 18.12. Окно приложения PQClock после изменения размеров окна

АРХИТЕКТУРА "ДОКУМЕНТ/ПРЕДСТАВЛЕНИЕ"

Глава 19

Основные положения

Прежде чем переходить к рассмотрению темы, вынесенной в заголовок части, сделаем обзор тех основных возможностей, которые мы уже можем осознанно использовать.

Каждое приложение, написанное для Windows, имеет одно главное окно, которое состоит из двух компонентов — фрейма и клиентской (рабочей) области. Оно может быть как стандартным окном Windows (например, при построении SDI- и MDI-приложений), так и обычным блоком диалога. В любом случае, внутри главного окна приложение предоставляет пользователю всю необходимую для работы информацию, под которой понимаются не только данные, т. е. текст, рисунки и т. п., но и элементы интерфейса — меню, полосы прокрутки, панели инструментов, строка состояния и т. д. Каждый из этих элементов (по крайней мере, для приложений, созданных с использованием библиотеки классов MFC) является объектом, в состав которого входят функции, определяющие его поведение, — взаимодействие как с другими объектами, так и с данными, с которыми он работает. Любое действие пользователя во время выполнения приложения порождает сообщение, которое содержит в себе информацию о том, *что* конкретно сделал пользователь. Все объекты приложения обладают способностью реагировать на те или иные сообщения. Во многих случаях эта способность заложена в классах библиотеки MFC, а для других реализуются специальные обработчики. Повторим — все перечисленные свойства характерны (с той или иной степенью полноты) для каждого Windows-приложения.

Сосредоточимся на тех приложениях, которые в качестве своего главного окна имеют стандартное окно Windows. Основным для таких окон является объект, созданный на базе класса CFrameWnd (для SDI-приложений) или CMDIFrameWnd (для MDI-приложений). То, что делалось до сих пор, заключалось в создании некоторых необходимых оконных объектов и *самостоятельной* организации взаимодействия между ними. Мы полностью (более того — сознательно) игнорировали такое мощное средство, заложенное

разработчиками в библиотеку классов MFC, как архитектура "документ/представление" (document/view).

В основе этой архитектуры лежат три глобальных понятия — *фрейм*, *документ* и *представление*. И если с фреймом мы уже достаточно близко познакомились (некоторые его свойства, связанные с архитектурой "документ/представление", будут представлены чуть ниже), то два других необходимо рассмотреть подробно, что мы и сделаем в следующей главе. Здесь же просто определим их место в архитектуре "документ/представление".

Что же включают в себя эти понятия? В самом общем случае, под *документом* фирма Microsoft понимает те данные, с которыми работает приложение. Это может быть все, что угодно, — "простой" текст, картинка и т. п. Отображение этих данных на экране осуществляется в так называемом *фрейме* документа. Разработчики библиотеки MFC создали для фреймов специальные классы окон — *представления*, которые отображают данные документа и управляют взаимодействием пользователя с ними. Другими словами, способ хранения данных в памяти или на диске никоим образом не влияет на их внешнее представление пользователю.

Примечание

К сожалению, сама фирма Microsoft сплошь и рядом нарушает это положение, что наглядно видно на поставляемых примерах. Да и класс CEditView спроектирован отнюдь не на этом принципе. Мы, тем не менее, будем придерживаться этого основного положения.

Роль фрейма в этой архитектуре заключается в создании соответствующих фреймов документов, содержащих специальные окна — представления. Кроме того, фрейм осуществляет координацию взаимодействия между документами различных типов и их представлениями, направляя им команды и получая от них извещения. Маршруты, по которым осуществляется передача сообщений, подробно описаны нами в пятой части книги.

Архитектура "документ/представление" разделяет для себя данные — документы, и их изображения. Каждый документ представлен в виде объекта, который обеспечивает пространство для хранения данных в памяти и отвечает за такие операции, как запись и чтение документа с диска. Очевидно, что большая часть таких операций выполняется самим приложением, а не классами библиотеки MFC. Кроме того, для каждого используемого представления также создается отдельный объект, который представляет собой экранное окно, отвечающее за взаимодействие с пользователем и соответствующим объектом документа, печать и т. д. Совместную работу перечисленных объектов координирует фрейм, который, в большинстве случаев, является главным окном приложения (рис. 19.1).

Когда запускается приложение под Windows, пользователь взаимодействует с документами посредством их изображений во фреймах. Фреймы докумен-

тов имеют два основных компонента: собственно фрейм и его содержимое. Для них библиотека MFC использует два различных класса. Класс фрейма управляет непосредственно фреймом, а класс представления — его содержимым. При этом окно представления является дочерним по отношению к фрейму, т. е. размещается в его рабочей области. Сам фрейм документа может быть окном SDI-приложения или дочерним окном MDI-приложения.

Рис. 19.1. Взаимосвязь между фреймом, документом и представлением

Работа с основными компонентами фрейма документа осуществляется также по-разному. За взаимодействие пользователя непосредственно с фреймом (изменение размеров, перемещение и т. д.) полностью отвечает сама система Windows. Управление же содержимым фрейма, т. е. представлениями, ложится на плечи разработчика.

Для координации создания и взаимодействия трех основных объектов рассматриваемой архитектуры в библиотеке классов MFC используются специальные классы шаблонов документов, которые создаются и управляются объектом "приложение".

Таким образом, архитектура "документ/представление" охватывает следующие основные классы:

- CWinApp — класс для создания единственного объекта — приложения
- CFrameWnd — класс для создания главного окна однодокументного приложения и базовый для классов CMDIFrameWnd и CMDIChildWnd, которые отвечают за работу многодокументного приложения
- CDocTemplate — базовый абстрактный класс для создания шаблонов документов; при работе с однодокументным приложением используется производный от него класс CSingleDocTemplate, а для многодокументных — класс CMultiDocTemplate
- CDocument — класс для создания собственно документа

- CView — базовый класс, который совместно со своими производными классами — CCtrlView, CEditView, CListView, CTreeView и CScrollView — отвечает за отображение данных документа и за взаимодействие с пользователем; для этих же целей можно использовать класс CSplitterWnd

Примечание

В представленный список не включены классы, предназначенные для работы с базами данных, OLE и Internet.

Нам осталось рассмотреть вопросы их создания и взаимодействия.

Место объекта-приложения в архитектуре "документ/представление"

Класс CWinApp был достаточно подробно рассмотрен в главе 7, и нет смысла повторяться. Однако там мы не касались его свойств, связанных с архитектурой "документ/представление", отложив обсуждение этого вопроса до более удобного момента. Теперь пришло время заполнить этот пробел.

Но прежде чем обсуждать роль класса CWinApp в рассматриваемой архитектуре, необходимо ввести понятие *шаблона документа*.

Приложение может поддерживать произвольное число типов документов. И единственное, что для этого нужно — создать и "зарегистрировать" во время инициализации объекта-приложения (при выполнении функции InitInstance) необходимое число шаблонов документов. Причем для каждого типа документа, с которым предполагается работа, используется свой шаблон. Например, если приложение поддерживает документы двух разных типов — табличные и текстовые, то оно должно создать для них два разных шаблона.

В библиотеке классов MFC для работы с шаблонами документов реализованы специальный класс — CDocTemplate и два производных от него — CSingleDocTemplate и CMultiDocTemplate. Рассмотрим возможности, которые в них заложены.

Класс CDocTemplate

Это абстрактный класс, в котором реализованы основные функциональные возможности для работы с шаблонами документов — организация и управление взаимодействием между классами трех типов:

- Классом документа, образованного из CDocument
- Классом представления, который изображает данные документа; можно создать этот класс на базе CView, CScrollView, CFormView, CEditView и

т. д. (CEditView можно использовать непосредственно для текстовых документов)

- Классом фрейма, который содержит представление; для SDI-приложений он создается на основе CFrameWnd, а для MDI-приложений — на базе CMDIChildWnd (базовые классы можно использовать и непосредственно)

Рассмотрим основные компоненты этого класса. Начнем, как обычно, с конструктора, который фиксирует типы документа, фрейма и представления, доступные для работы с данным шаблоном:

CDocTemplate::**CDocTemplate**(
UINT nIDResource,
CRuntimeClass *pDocClass,
CRuntimeClass *pFrameClass,
CRuntimeClass *pViewClass) —

создает объект класса. В качестве аргументов используются: nIDResource — идентификатор ресурсов (меню, акселераторы и строка описания типа шаблона документа), используемых с этим типом документов; pDocClass — указатель на объект CRuntimeClass, отвечающий за данные документа; pFrameClass — указатель на объект CRuntimeClass, характеризующий фрейм документа; pViewClass — указатель на объект CRuntimeClass, отвечающий за представление документа. Эти объекты должны быть уже определены к моменту создания объекта "шаблон документа". Память под этот объект следует распределять динамически, и полученный указатель передать в качестве параметра в функцию CWinApp::AddDocTemplate.

virtual void CDocTemplate::**AddDocument**(CDocument *pDoc)

и

virtual void CDocTemplate::**RemoveDocument**(CDocument *pDoc) —

добавляет (или, соответственно, удаляет) документ, на который указывает pDoc, в (из) список документов, ассоциированных с этим шаблоном. В производных классах CSingleDocTemplate и CMultiDocTemplate эти функции переопределены.

virtual void CDocTemplate::**LoadTemplate**( ) —

загружает ресурсы для данного шаблона документа. Обычно вызывается библиотекой классов во время создания объекта и практически не требует прямого вызова. Исключение составляет случай, когда объект класса создается на глобальном уровне. В этом случае во время выполнения функции CWinApp::AddDocTemplate необходим ее явный вызов. В качестве ресурсов, используемых документами этого типа, могут использоваться меню, акселераторы и единственный строковый ресурс.

Рассмотрим структуру строки описания типа более подробно.

```
/ Строка описания типа шаблона документа
STRINGTABLE PRELOAD DISCARDABLE
BEGIN
 IDR_NOTETYPE "\nNote\nNote\nNote Files (*.tnd)\n.tnd
               \nNote.Document\nNote Document"
END
```

Как видите, он состоит из семи подстрок (первая в данном примере — пустая), разделенных символами "\n", содержащих дополнительную информацию о типе документа:

- Заголовок главного окна SDI-приложения (Main frame caption). Не используется для MDI-приложений
- Имя файла, присваиваемое новому документу по умолчанию (Doc type name) — Note. Если эта подстрока отсутствует, то используется имя Untitled или его аналог для локализованных версий Windows (БезИмени — для русской версии)
- Имя типа документа (File new name (OLE short name)) — Note. Если в приложении определено больше одного шаблона, то при создании нового документа на экране отображается блок диалога для выбора типа создаваемого документа
- Описание типа документа и его фильтра (Filter name) — Note Files (*.tnd). Эта строка отображается в списке типов документов стандартных блоков диалога **Open** (открыть) и **Save As** (сохранить как)
- Расширение, используемое для документов этого типа (File extension) — .tnd
- Идентификатор типа документа, хранящийся в реестре Windows (File type ID), — Note.Document
- Имя типа документа, хранящееся в реестре Windows (File new name (OLE long name)), — Note Document. Это же имя используется в качестве длинного имени объекта OLE, если в приложение включена соответствующая поддержка

Примечание

В скобках указаны названия соответствующих полей панели **Advanced Options** (Дополнительные параметры) мастера создания каркаса приложений AppWizard.

Если какая-либо из этих подстрок не включается в строку, то вместо нее обязательно используются разделяющие символы "\n". Заключительные символы "\n" в строке являются необязательными.

virtual BOOL CDocTemplate::**GetDocString**(
CString& rString,
enum DocStringIndex index) —

позволяет получить определенную подстроку, описывающую тип документа, которая хранится в шаблоне и получена из соответствующей строки файла ресурсов приложения. Найденная подстрока записывается в rString. Параметр index определяет индекс подстроки, информацию о котором необходимо получить. Он может принимать одно из следующих значений:

CDocTemplate::**windowTitle**	Имя, отображаемое в заголовке главного окна приложения; представлено только в шаблоне документа для SDI-приложений

CDocTemplate**::docName**	Основа имени документа; имя нового документа этого типа складывается из основы, к которой добавляется число; если значение не определено, то по умолчанию используется Untitled (для русских версий Windows – БезИмени)
CDocTemplate::**fileNewName**	Имя типа этого документа; если значение не определено, то этот тип документа недоступен при выполнении команды **File\New** (файл\создать)
CDocTemplate::**filterNewName**	Описание типа документа и нейтрального символа для фильтра, сопоставляемого с документами этого типа; эта строка изображается в списке типов файлов в блоке диалога **File Open**; если значение не определено, то этот тип документа недоступен при выполнении команды **File\Open** (файл\открыть)
CDocTemplate::**filterExt**	Расширение документов этого типа; если значение не определено, то этот тип документа недоступен при выполнении команды **File\Open**
CDocTemplate::**regFileTypeId**	Идентификатор типа документа, хранящийся в реестре Windows; эта строка только для внутреннего использования; если не определена, то тип документа не может быть зарегистрирован в **File Manager** (диспечер файлов)
CDocTemplate::**regFileTypeName**	Имя типа документа, хранящийся в реестре Windows; эта строка может быть отображена в блоках диалога приложения, вызываемых для доступа к реестру Windows

virtual BOOL CDocTemplate::**SaveAllModified**( ) —

позволяет сохранить все модифицированные документы, которые ассоциированы с этим шаблоном.

virtual void CDocTemplate::**CloseAllDocuments**(BOOL bEndSession) —

позволяет закрыть все открытые документы. Обычно используется как часть кода, обрабатывающего команду **Exit** (Выход). Реализация по умолчанию вызывает функцию CDocument::DeleteContents, которая удаляет данные из документов и закрывает фреймы всех представлений, присоединенных к документу. Параметр bEndSession определяет, нужно ли завершать сеанс работы (TRUE) или нет (FALSE). Если перед закрытием документа необходимо выполнить какие-либо специфические действия, то функцию следует переопределить.

Кроме вышеперечисленных, в классе объявлены еще четыре "чисто виртуальные" функции: GetFirstDocPosition, GetNextDoc, OpenFileName и SetDefaultTitle. Наличие этих функций превращает CDocTemplate в абстрактный класс и, следовательно, его нельзя использовать непосредственно. Обычно приложения используют один из двух производных от него классов, предоставляемых библиотекой MFC: CSingleDocTemplate — для SDI- и CMultiDocTemplate — для MDI-приложений.

Если по каким-либо причинам может потребоваться интерфейс, который фундаментально отличается от SDI и MDI, то можно образовать свой класс непосредственно из CDocTemplate.

Для полноты картины приведем прототипы еще нескольких функций этого класса, которые в явном виде используются достаточно редко, но в некоторых случаях могут быть полезны.

Если шаблон поддерживает несколько типов файлов, то, используя функцию MatchDocType для каждого шаблона, можно определить, какие шаблоны документов подходят для данного файла, и выбрать тот, который нужен:

virtual Confidence CDocTemplate::**MatchDocType**(
LPCTSTR lpszPathName,
CDocument *&rpDocMatch) —

служит для определения типа шаблона документа, используемого для открытия файла. Параметры: lpszPathName — полное имя файла; rpDocMatch — указатель на уже открытый документ, если файл, определенный в lpszPathName, уже открыт. Функция возвращает значение, определенное в перечислении Confidence:

```
enum Confidence
{
  noAttempt,
  maybeAttemptForeign,
  maybeAttemptNative,
  yesbeAttemptForeign,
  yesbeAttemptNative,
  yesAlreadyOpen
};
```

Если файл, определенный в параметре lpszPathName, уже открыт, то функция возвращает CDocTemplate::yesAlreadyOpen и копирует объект CDocument файла в объект rpDocMatch. Если файл еще не открыт, но расширение имени в lpszPathName совпадает со значением, определенным в CDocTemplate::filterExt, то функция возвращает CDocTemplate::yesAttemptNative и устанавливает rpDocMatch в NULL. В остальных случаях функция возвращает значение CDocTemplate::yesAttemptForeign. Другие члены перечисления можно использовать при переопределении этой функции.

virtual CDocument* CDocTemplate::**CreateNewDocument**( ) —

создает новый документ, тип которого определяется текущим шаблоном. Функция возвращает указатель на вновь созданный документ или NULL в случае ошибки.

virtual CFrameWnd* CDocTemplate::**CreateNewFrame**(
CDocument *pDoc,
CFrameWnd *pOther) —

создает новый фрейм (обычно невидимый) с присоединенными к нему документом и представлением. Параметр pDoc указывает на документ, на который ссылается новый фрейм. Если он равен NULL, то библиотека выводит соответствующее информационное сообщение. При создании нового фрейма можно использовать модель, которая задается параметром pOther. Эта функция позволяет создать новый фрейм вне стандартной реализации обработчиков команд ID_FILE_NEW и ID_FILE_OPEN. Она возвращает указатель на вновь созданный фрейм или NULL, если произошла какая-либо ошибка.

virtual void CDocTemplate::**InitialUpdateFrame**(
CFrameWnd *pFrame,
CDocument *pDoc,
BOOL bMakeVisible = TRUE) —

применяется совместно с функцией CreateNewFrame для создания нового фрейма вне стандартной реализации команд ID_FILE_NEW и ID_FILE_OPEN и служит для получения представлением фрейма (параметр pFrame) вызова функции OnInitialUpdate. Параметр pDoc указывает на документ, ассоциированный с фреймом, или содержит NULL. Параметр bMakeVisible показывает, должен ли фрейм стать видимым и активным.

Как мы уже сказали, CDocTemplate — абстрактный класс, и на его основе объекты шаблонов не создаются. Его основная роль заключается в обеспечении шаблонов основными свойствами, а для создания конкретных объектов в библиотеке классов MFC реализованы два класса — CSingleDocTemplate и CMultiDocTemplate, которые специально адаптированы для работы соответственно в SDI- и MDI-приложениях.

Класс CSingleDocTemplate

Данный класс определяет шаблон документа для однодокументного интерфейса. В SDI-приложениях главный фрейм является одновременно и фреймом документа, т. е. в конкретный момент времени может быть открыт только один документ. Обычно SDI-приложения поддерживают один тип документа (хотя это и не является обязательным условием), т. е. они содержат только один объект класса CSingleDocTemplate.

Объекты этого класса обрабатываются внутри библиотеки MFC. В своей защищенной части класс содержит член, в котором хранится указатель на документ, присоединенный к шаблону:

```
class CSingleDocTemplate : public CDocTemplate
{
...
protected:
  CDocument *m_pOnlyDoc;
};
```

Ниже представлены основные функции класса.

CSingleDocTemplate::**CSingleDocTemplate**(
UINT nIDResource,
CRuntimeClass *pDocClass,
CRuntimeClass *pFrameClass,
CRuntimeClass *pViewClass) —

создает объект соответствующего класса. Назначение параметров см. в описании конструктора класса CDocTemplate.

Помимо конструктора, в классе реализованы все четыре "чисто виртуальные" функции класса CDocTemplate.

virtual POSITION CDocTemplate::**GetFirstDocPosition**( ) —

возвращает позицию первого документа из списка документов, ассоциированных с данным шаблоном, или NULL, если список пуст. Она переопределена в обоих производных классах — CSingleDocTemplate и CMultiDocTemplate. Если возникло желание создать свой класс на базе CDocTemplate, то эту функцию следует в нем переопределить.

virtual CDocument* CDocTemplate::**GeNextDoc**(POSITION &rPos) —

возвращает указатель на объект-документ из общего списка документов, ассоциированных с шаблоном, который хранится непосредственно за документом, заданным его позицией в списке (параметр rPos). Позиция полученного документа записывается в rPos. Если возвращен указатель на последний элемент списка, то в параметр rPos функция записывает NULL. Для получения указателя на первый документ списка необходимо вызвать эту функцию до GetFirstDocPosition. При использовании в качестве параметра rPos недопустимого значения библиотека MFC генерирует исключение.

virtual CDocument* CDocTemplate::**OpenDocumentFile**(
LPCTSTR lpszPathName,
BOOL bMakeVisible = TRUE) —

позволяет открыть файл, заданный полным именем lpszPathName, который содержит необходимый документ. Если значение этого параметра равно NULL, то вызывается функция CreateNewDocument, которая создает новый файл, содержащий документ типа, ассоциированного с этим шаблоном. При успешном завершении функция возвращает указатель на документ, в противном случае NULL.

virtual void CDocTemplate::**SetDefaultTitle**(CDocument *pDoc) —

загружает заголовок документа, используемый по умолчанию и определенный в строке ресурса, и выводит его в полосу заголовка фрейма документа.

Кроме перечисленных "чисто виртуальных" функций, в классе переопределены также функции базового класса AddDocument и RemoveDocument.

Класс CMultiDocTemplate

Этот класс определяет шаблон документа для многодокументного интерфейса. MDI-приложения используют главный фрейм как рабочее место, в котором пользователь может открывать произвольное число окон документа, отображающих данные того или иного документа. Они могут поддерживать больше одного шаблона документов, и документы различных типов могут быть открыты одновременно.

Приложение использует шаблон(ы) документа, когда пользователь создает новый документ. Если приложение поддерживает больше одного типа документов, то библиотека получает имена поддерживаемых типов из шаблона и отображает их в списке блока диалога **New**, реализованного на базе класса CNewTypeDlg. Как только пользователь выбрал тип документа, приложение

создает объекты "документ", "фрейм" и "представление" и сопоставляет их друг другу.

В классе определены два общедоступных члена, отвечающих за совместное использование меню и таблицы акселераторов:

```
HMENU  CMultiDocTemplate::m_hMenuShared;
HACCEL CMultiDocTemplate::m_hAccelTable;
```

Поскольку в многодокументных приложениях одновременно могут быть открыты несколько документов, в классе предусмотрен защищенный член:

```
CPtrList m_docList;
```

хранящий список открытых документов этого типа. Доступ к элементам списка осуществляется посредством функций GetFirstDocPosition и GetNextDoc.

Число непоименованных открытых документов хранится в специальном защищенном члене класса (нумерация начинается с нуля):

```
UINT m_nUntitledCount;
```

Значение, хранящееся в этой переменной, используется для генерации имени по умолчанию, и автоматически *увеличивается* при создании новых документов.

В MDI-приложениях определен, прежде всего, конструктор класса:

CMultiDocTemplate::**CMultiDocTemplate**(
UINT nIDResource,
CRuntimeClass *pDocClass,
CRuntimeClass *pFrameClass,
CRuntimeClass *pViewClass) —

создает объект класса.

Помимо конструктора, в классе также реализованы все четыре "чистые" функции базового класса: GetFirstDocPosition, GetNextDoc, OpenFileName и SetDefaultTitle, описание которых приведено выше. Кроме того, "наполнены новым содержанием" еще три функции: LoadTemplate, AddDocument и RemoveDocument.

В заключение обсуждения вопросов создания и использования шаблонов документов рассмотрим, каким образом с документом ассоциируется некоторый значок. Начнем с того, что написано в документации. "Значок, зарегистрированный для каждого шаблона документа, базируется на его позиции в списке шаблонов приложения. Порядок расположения шаблонов определяется порядком его занесения в список, т. е. порядком вызова функций AddDocTemplate. При этом библиотека MFC присваивает первый ресурс значка самому приложению, следующий — первому документу, и т. д.". А теперь попытаемся разобраться, что же все-таки здесь имеется в виду,

ведь при работе значок определяется практически его числовым идентификатором, а отнюдь не "позицией в списке". Рассмотрим фрагмент файла resource.h приложения NoteDraw:

```
...
#define IDR_MAINFRAME 128 // Идентификатор главного окна
#define IDR_NOTETYPE  129 // Идентификатор окна текстового документа
#define IDR_DRAWTYPE  130 // Идентификатор окна графического
                          // документа
#define IDI_NOTEICON  129 // Идентификатор значка окна
                          // текстового документа
#define IDI_DRAWICON  130 // идентификатор значка окна
                          // графического документа
...
```

Прежде всего следует обратить внимание на то, что числа, характеризующие идентификаторы ресурсов документов и значков, принадлежат одному ряду. Это условие является обязательным, т. к. в противном случае на экране вы увидите значок по умолчанию вместо ожидаемого. И, наконец, числовые значения идентификаторов ресурсов документов и значков должны совпадать, иначе можно в окне текстового документа увидеть значок графического и наоборот. Во избежание подобных недоразумений рекомендуем для значков использовать тот же идентификатор, что и для ресурсов документов: IDR_NOTETYPE и IDR_DRAWTYPE.

Возвращаясь к классу приложения CWinApp, рассмотрим кратко те его функции, которые отвечают за взаимодействие с другими объектами, входящими в архитектуру "документ/представление".

void CWinApp::**AddDocTemplate**(CDocTemplate *pTemplate) —

добавляет шаблон документа pTemplate в список доступных шаблонов, которые поддерживает приложение. Все необходимые добавления в шаблон документа следует производить в функции InitInstance до вызова функции RegisterShellFileTypes, т. к. в противном случае типы документов не будут представлены в реестре.

В качестве примера рассмотрим фрагмент кода приложения NoteDraw, отвечающий за создание двух шаблонов документов, один из которых позволяет создавать документы для работы с текстом, а второй — с графическими объектами (полный текст приложения имеется на сопроводительной дискете):

```
...
CMultiDocTemplate* pDocTemplate;
  // Создаем объект шаблона документа для работы с текстом
  pDocTemplate = new CMultiDocTemplate(
              IDR_NOTETYPE,
              RUNTIME_CLASS(CNoteDoc),
```

```
            RUNTIME_CLASS(CNoteFrame),
            RUNTIME_CLASS(CNoteView));
  // Добавляем шаблон в список
  AddDocTemplate(pDocTemplate);
  // Создаем объект шаблона документа для работы с графикой
  pDocTemplate = new CMultiDocTemplate(
            IDR_DRAWTYPE,
            RUNTIME_CLASS(CDrawDoc),
            RUNTIME_CLASS(CDrawFrame),
            RUNTIME_CLASS(CDrawView));
  // Добавляем шаблон в список
  AddDocTemplate(pDocTemplate);
...
STRINGTABLE PRELOAD DISCARDABLE
BEGIN
  IDR_MAINFRAME "NoteDraw"
  IDR_NOTETYPE  "\nNote\nNote\nNote Files
                 (*.tnd)\n.tnd\nNote.Document\nNote Document"
  IDR_DRAWTYPE  "\nDraw\nDraw\nDraw Files
                 (*.dnd)\n.dnd\nDraw.Document\nDraw Document"
END
...
```

POSITION CWinApp::**GetFirstDocTemplatePosition**( ) —

позволяет получить позицию первого шаблона документа приложения. Если список пуст, то возвращается NULL.

CDocTemplate* CWinApp::**GetNextDocTemplate**(POSITION &pos) —

позволяет получить шаблон документа, определяемый его позицией (параметр pos) в списке шаблонов приложения. После возврата из функции в pos содержится значение позиции следующего шаблона документа. Если полученный шаблон был последним в списке, то pos устанавливается в NULL.

Использование этих двух функций позволяет просматривать список доступных шаблонов документов приложения. При этом порядок их расположения в списке определяется порядком их записи туда посредством вызова функции AddDocTemplate.

virtual CDocument* CWinApp::**OpenDocumentFile**(LPCTSTR lpszFileName) —

открывает файл документа, задаваемый именем lpszFileName, и возвращает указатель на соответствующий объект-документ. Если к моменту вызова функции файл уже был открыт, то активизируется фрейм, содержащий этот документ. Если приложение поддерживает несколько шаблонов документов, то для поиска соответствующего шаблона библиотека MFC использует расширение файла. В случае успешного выполнения на основе шаблона создается фрейм и представление соответствующего документа.

void CWinApp::**CloseAllDocuments**(BOOL bEndSession) —

закрывает все открытые документы до выхода из приложения. Если при этом параметр bEndSession равен TRUE, то завершается и сеанс работы с Windows. Функцию следует вызывать после функции HideApplication.

virtual BOOL CWinApp::**SaveAllModified**( ) —

вызывается библиотекой MFC для того, чтобы сохранить все документы, когда закрывается главное окно приложения или обрабатывается сообщение WM_QUERYENDSESSION. Реализация этой функции по умолчанию вызывает функцию CDocument::SaveModified для сохранения всех изменений в документах, открытых внутри приложения.

afx_msg void CWinApp::**OnFileNew**( )

и

afx_msg void CWinApp::**OnFileOpen**( ) —

обработчики команд ID_FILE_NEW и ID_FILE_OPEN. Если обработка команды возлагается на библиотеку классов MFC, в карту сообщений класса приложения, если вы хотите самостоятельно обрабатывать эту команду, необходимо добавить операторы:

```
ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
```

или

```
ON_COMMAND(ID_FILE_NEW, OnNameCommandFileNew)
ON_COMMAND(ID_FILE_OPEN, OnNameCommandFileOpen)
```

Программист должен реализовать обработчик OnNameCommandFileNew (имя, естественно, произвольное).

В свете архитектуры "документ/представление" основным назначением объекта-приложения является то, что он содержит полный список шаблонов документов, используя для этого общедоступный член:

CDocManager* CWinApp::**m_pDocManager**

Фактически список хранится в защищенном члене класса библиотеки MFC — CDocManager:

CPtrList CDocManager::**m_templateList**

и доступ к нему осуществляется при помощи рассмотренных функций объекта-приложения GetFirstDocTemplatePosition и GetNextDocTemplate, которые после предварительной обработки передают управление соответствующей функции класса CDocManager. В качестве примера рассмотрим три фрагмента кода из исходных текстов реализации функций классов CWinApp и CDocManager, первый из которых отвечает за добавление документа в шаблон, а остальные обрабатывают команду ID_FILE_NEW:

```
// Первый фрагмент: добавление шаблона в список
// Создан объект класса CDocManager?
if (m_pDocManager == NULL)
```

```
{
  // Если нет, то создаем его
  m_pDocManager = new CDocManager;
}
// И в обоих случаях передаем ему управление
// для добавления шаблона документа в общий список
m_pDocManager->AddDocTemplate(pTemplate);
...
// Второй фрагмент: обработка команды ID_FILE_NEW
// в классе CWinApp
// Создан объект класса CDocManager?
if (m_pDocManager != NULL)
{
  // Если да, то передаем ему управление для выбора шаблона
  m_pDocManager->OnFileNew();
}
...
// Третий фрагмент: обработка команды ID_FILE_NEW
// в классе CDocManager
...
// Проверяем, имеется ли в списке какой-либо шаблон
if (m_templateList.IsEmpty())
{
  // Если список пуст, то вывод сообщения об ошибке и возврат
  ...
  return;
}
// Получаем указатель на первый в списке шаблон документов
CDocTemplate* pTemplate=(CDocTemplate*)m_templateList.GetHead();
// В списке больше одного шаблона?
if (m_templateList.GetCount() > 1)
{
  // Создаем и выводим на экран блок диалога
  // со списком доступных шаблонов
  CNewTypeDlg dlg(&m_templateList);
  int nID = dlg.DoModal();
  if (nID == IDOK)
  {
    // Фиксируем выбранный шаблон
```

```
    pTemplate = dlg.m_pSelectedTemplate;
  }
  else
  {
    return;
  }
}
// Открываем новый пустой документ на базе выбранного шаблона
pTemplate->OpenDocumentFile(NULL);
...
```

Наибольший интерес здесь представляет тот факт, что приложение может одновременно поддерживать несколько шаблонов (или типов) документов (рис. 19.2).

Рис. 19.2. Несколько документов двух типов, используемых одним приложением

В этом случае для выбора необходимого шаблона при обработке команды ID_FILE_NEW создается и выводится на экран специальный блок диалога (рис. 19.3) класса CNewTypeDlg, реализованного в библиотеке MFC (файл DOCMGR.CPP):

```
class CNewTypeDlg : public CDialog
{
protected:
  CPtrList*     m_pList;             // Список шаблонов документов
public:
  CDocTemplate* m_pSelectedTemplate; // Выбранный шаблон
public:
  //{{AFX_DATA(CNewTypeDlg)
```

```
  enum { IDD = AFX_IDD_NEWTYPEDLG };
  //}}AFX_DATA
  CNewTypeDlg(CPtrList* pList) : CDialog(CNewTypeDlg::IDD)
  {
    m_pList = pList;
    m_pSelectedTemplate = NULL;
  }
...
};
```

Он позволяет пользователю просматривать список имеющихся у приложения шаблонов документов и выбирать один из них.

Примечание

В принципе, вместо этого класса можно использовать и созданный самостоятельно, но мы не видим серьезных причин для этого.

Рис. 19.3. Блок диалога для выбора типа вновь создаваемого документа

Рис. 19.4. Процесс создания шаблона документа

Таким образом, если внимательно посмотреть на перечисленные функции и члены класса CWinApp при работе в рамках архитектуры "документ/представление", роль объекта-приложения заключается в создании, хранении и организации доступа к списку доступных для использования шаблонов документов. Фактически его роль сводится лишь к передаче управления "диспетчеру документов", представленному классом CDocManager, который, собственно, и выполняет всю необходимую для этого работу. Схематично процесс создания шаблона документа изображен на рис. 19.4.

Роль фреймов в архитектуре "документ/представление"

Фрейм документа, который мы будем здесь рассматривать, имеет две основные составляющие — собственно фрейм и его содержимое, представляющее на экране данные документа. Эти два компонента представлены и управляются различными группами классов библиотеки MFC: классы фреймов (CFrameWnd и CMDIChildWnd) и классы представлений (CView, CScrollView и другие). Процесс создания главного и дочерних фреймов подробно обсуждался ранее (см. главы 9, 10) и нет смысла повторяться. Вместо этого рассмотрим несколько фрагментов кода из приложения NoteDraw:

```
// Описание класса (файл Childfrm.h)
class CNoteFrame : public CMDIChildWnd
{
// Для возможности динамического создания объекта
// класса CNoteFrame вызывается макрос
  DECLARE_DYNCREATE(CNoteFrame)
protected:
  CNoteFrame();
...
};
и
// Реализация класса (файл Childfrm.cpp)
// Для возможности динамического создания объекта
// класса CNoteFrame вызывается макрос
IMPLEMENT_DYNCREATE(CNoteFrame, CMDIChildWnd)
```

Интерес здесь представляют два макроса, подробно описанные в главе 8, — DECLARE_DYNCREATE и IMPLEMENT_DYNCREATE — и защищенный (protected) конструктор по умолчанию. Дело в том, что все вышеперечисленные составляющие архитектуры "документ/представление" (документ, фрейм и представление) создаются шаблоном документов *динамически*, во

время выполнения программы. Кроме того, эти макросы охватывают также все возможности двух других групп макросов — DECLARE_DYNAMIC, IMPLEMENT_ DYNAMIC, DECLARE_SERIAL и IMPLEMENT_ SERIAL, что гарантирует также поддержку получения информации времени выполнения и сериализации. Забыв включить эти макросы соответственно в объявление и реализацию некоторого класса, вы тем самым исключаете для него возможность корректной работы механизмов RUNTIME_CLASS и сериализации.

Рассмотрим функции класса CFrameWnd, которые связаны с архитектурой "документ/представление".

void BOOL CFrameWnd::**OnCreateClient**(
LPCREATESTRUCT lpcs,
CCreateContext *pContext) —

вызывается только библиотекой классов MFC из обработчика OnCreate и не должна вызываться непосредственно. В качестве параметров в функцию передаются: lpcs — указатель на структуру CREATESTRUCT и pContext — указатель на структуру CCreateContext:

```
struct CCreateContext{
  CRuntimeClass* m_pNewViewClass;
  CDocument*     m_pCurrentDoc;
  CDocTemplate*  m_pNewDocTemplate;
  CView*         m_pLastView;
  CFrameWnd*     m_pCurrentFrame;
CCreateContext();
};
```

Структура CCreateContext создается шаблоном документа (при создании документа и ассоциированных с ним компонентов). Перечислим поля этой структуры (все поля могут содержать NULL):

m_pNewViewClass	Указатель на структуру CRuntimeClass вновь создаваемого представления
m_pCurrentDoc	Указатель на уже существующий документ, с которым ассоциируется новое представление
m_pNewDocTemplate	Указатель на шаблон документа, с которым ассоциирован создаваемый фрейм
m_pLastView	Указатель на начальное представление, к которому добавляется создаваемое, например, при работе с разделенным окном или при создании дополнительного представления документа
m_pCurrentFrame	Указатель на текущий фрейм, к которому добавляется создаваемый, например, при создании дополнительного фрейма документа

Библиотекой MFC эта структура используется в следующих функциях: CWnd::Create, CFrameWnd::CreateView, CFrameWnd::LoadFrame, CFrameWnd::OnCreateClient, CFrameWnd::Create, CSplitterWnd::Create и CSplitterWnd::CreateView.

virtual void CFrameWnd::**ActivateFrame**(int nCmdShow = −1) —

активизирует фрейм, т. е. делает его видимым и доступным для пользователя. Параметр nCmdShow принимает такие же значения, как и посылаемые в функцию CWnd::ShowWindow. По умолчанию функция делает фрейм видимым и переводит его на самый верх Z-порядка, и при необходимости то же самое выполняет для главного окна приложения. При переопределении после проведения необходимых действий, например, разворачивания дочернего окна, необходимо вызвать соответствующую функцию базового класса с явным указанием параметра nCmdShow, как, например, сделано в приложении NoteDraw:

```
void CNoteFrame::ActivateFrame(int nCmdShow)
{
// Разворачиваем фрейм документа
  CMDIChildWnd::ActivateFrame(SW_MAXIMIZE);
}
```

void CFrameWnd::**InitialUpdateFrame**(
CDocument *pDoc,
BOOL bMakeVisible) —

вызывается после создания нового фрейма и инициирует вызов обработчика OnInitialUpdate для всех своих представлений. При этом активизируется представление, у которого идентификатор дочернего окна установлен в AFX_IDW_PANE_FIRST. Если значение параметра bMakeVisible равно TRUE, то фрейм делается видимым; в противном случае — нет. Параметр pDoc определяет документ, ассоциированный с данным фреймом.

virtual CFrameWnd* CFrameWnd::**GetActiveFrame**( ) —

возвращает указатель на активное дочернее окно MDI-приложения. При работе с SDI-приложениями функция возвращает указатель на текущий объект — this.

void CFrameWnd::**SetActiveView**(
CView *pViewNew,
BOOL bNotify = TRUE) —

переводит представление pView в активное состояние. Эта функция автоматически вызывается библиотекой MFC, когда пользователь сменил фокус представления внутри фрейма. Ее можно вызвать специально для программной смены фокуса представления.

```
// Устанавливаем фокус на область (1, 0)
SetActiveView((CView*)m_spltWnd.GetPane(1, 0));
```

Параметр bNotify определяет, будет ли само представление извещено об активизации. Если его значение равно TRUE, то для активизируемого представления вызывается функция OnActivateView.

CView* CFrameWnd::**GetActiveView**( ) —

возвращает указатель на объект текущего активного представления. Если такого нет, то возвращается NULL.

CWnd* CFrameWnd::**CreateView**(
CCreateContext *pContext,
UINT nID = AFX_IDW_PANE_FIRST) —

создает представление внутри фрейма, которое не является производным от CView, и возвращает указатель на него. После выполнения функции необходимо самостоятельно установить активность и видимость вновь созданного представления. Параметр pContext определяет тип представления и документа, а nID — идентификатор представления.

virtual CDocument* CFrameWnd::**GetActiveDocument**( ) —

возвращает указатель на документ, ассоциированный с текущим активным представлением. Если такого нет, что, в принципе, возможно, то функция возвращает значение NULL.

В качестве примера (фрагмент кода из приложения NoteDraw) рассмотрим, как можно переопределить функцию OnCreateClient для создания статического разделенного окна с двумя горизонтальными областями, в каждой из которых создается свое представление (рис. 19.5).

Примечание

Более подробно все перечисленные действия будут описаны в следующей главе при обсуждении собственно представлений.

```
BOOL CNoteFrame::OnCreateClient(LPCREATESTRUCT lpcs,
                                CCreateContext* pContext)
{
  // Создаем разделенное окно Windows с двумя строками и одним столбцом
  if(!m_spltWnd.CreateStatic(this, 2, 1))  {
    ...
  }
  // Создаем представление в первой строке (горизонтальной области)
  if(!m_spltWnd.CreateView(0, 0, pContext->m_pNewViewClass,
      CSize(lpcs->cx, lpcs->cy*2/3), pContext))  {
    ...
  }
  // Создаем представление во второй строке (горизонтальной области)
  if(!m_spltWnd.CreateView(1, 0, RUNTIME_CLASS(CTextView),
      CSize(0, 0), pContext))  {
    ...
  }
  // Устанавливаем фокус на область (1, 0)
```

```
  SetActiveView((CView*)m_spltWnd.GetPane(1, 0));
  return TRUE;
}
```

Рис. 19.5. Результат выполнения переопределенной функции OnCreateClient

Характер рассмотренных функций достаточно наглядно показывает роль фрейма в архитектуре "документ/представление". Прежде всего, фрейм хранит путь к текущему активному представлению. В случае, когда он содержит более одного представления, например, при работе с разделенным окном, то в качестве активного принимается самое последнее используемое представление.

Примечание

Активность представления не зависит от активного окна Windows или текущего фокуса ввода.

Последняя из описанных выше функций показывает, что фрейм имеет также доступ к документу, ассоциированному с текущим активным представлением. Кроме того, фрейм занимается координацией взаимодействия между различными представлениями, ассоциированными с определенным документом, направляя им команды и получая от них извещения. Маршруты, по которым осуществляется обмен сообщениями, подробно описаны нами в главе 8.

И последнее. Мы уже говорили, что с каждым документом ассоциируются определенные ресурсы меню и акселераторов. Так вот, задачу разделения этих ресурсов среди документов, созданных на основе одного шаблона, и замены ресурсов при активизации документа другого типа также с успехом решают главный фрейм и фрейм документа.

В заключение коротко перечислим те шаги, которые необходимо проделать для организации работы в рамках архитектуры "документ/представление":

- Создать объект-приложение
- Создать объекты-документы (см. главу 20)
- Создать необходимое число представлений каждого документа (см. главу 20)
- В функции InitInstance объекта-приложения создать объекты-шаблоны документов и добавить их к его списку
- Создать необходимые ресурсы для каждого из них
- Переопределить функцию CFrameWnd::OnCreateClient, в которой присоединить к фрейму документа необходимые представления
- Создать и вывести на экран главное окно приложения с присоединенными к нему фреймами документа

Для того чтобы закончить с основными положениями архитектуры "документ/представление", нам осталось только рассмотреть вопрос чтения/записи данных документов. Реализовано это на основе метода сериализации.

Сериализация

Преобразование в последовательную форму и обратно, т. е. *сериализация* — метод, который позволяет сохранять и восстанавливать объекты классов, созданных на базе классов CObject. Этот метод можно разделить функционально на две составляющие.

С одной стороны, наличие определенной виртуальной функции (Serialize) позволяет унифицировать процесс сохранения/восстановления объектов. Впрочем, это обычно для программ, написанных с использованием практически любого объектно-ориентированного языка. Для реализации этого процесса следует только переопределить виртуальную функцию, вызов которой по указателю на объект одного из базовых классов приведет к вызову нужной функции. Библиотека классов MFC идет несколько дальше, обеспечивая открытие потока, прежде всего файла, самостоятельно. Связывая этот поток с классом CArchive, библиотека обеспечивает тем самым удобный способ определения типа операции (чтение/запись) и целый набор функций для всех простейших типов данных (например, чисел) и библиотечных классов.

С другой стороны, сериализация поддерживает механизм динамического создания объектов заранее неизвестного типа. Например, приложение должно сохранять и восстанавливать некоторое количество объектов различного типа. Естественно, что для вызова соответствующего конструктора при восстановлении объекта необходимо точно знать тип создаваемого объ-

екта. Механизм сериализации делает это "за программиста", сохраняя необходимую информацию самостоятельно. Единственное, что требуется — все классы сохраняемых и восстанавливаемых объектов должны базироваться на классе CObject и предоставлять системе соответствующую информацию, используя макросы DECLARE_SERIAL и IMPLEMENT_SERIAL при объявлении и в реализации соответственно.

Поясним все вышесказанное на примере. Класс CGraphicsView имеет список указателей на объекты класса CDrawObject и производные от него. Класс CDrawObject основан на CObject и является абстрактным, а классы CLine, CRectangle и CEllipse являются производными от CDrawObject и поддерживают динамическое создание на основе сохраненной информации. Класс CDrawObject не может поддерживать динамическое создание в полном объеме, т. к. является абстрактным.

Рассмотрите объявление классов и реализацию их функций, обратив внимание на объявление конструкторов.

```
class CDrawObject : public CObject
{
protected: // Вызывается только во время сериализации
  CDrawObject();
public:
  CDrawObject(BOOL Selected);
  virtual ~CDrawObject();
  ...
  virtual void Draw(CDC *pDC) = 0;
  virtual void Serialize(CArchive& ar);
private:
  BOOL m_Selected;
};
class CLine : public CDrawObject
{
protected: // Вызывается только во время сериализации
  CLine();
  DECLARE_SERIAL(CLine)
public:
  CLine(CPoint beg, CPoint end);
  ~CLine();
  ...
  virtual void Draw(CDC *pDC);
  virtual void Serialize(CArchive& ar);
protected:
```

```
  CPoint m_beg;
  CPoint m_end;
};
class CRectangle : public CLine
{
protected: // Вызывается только во время сериализации
  CRectangle();
  DECLARE_SERIAL(CRectangle)
public:
  CRectangle(CPoint beg, CPoint end);
  virtual ~CRectangle();
  ...
  virtual void Draw(CDC *pDC);
};
class CEllipse : public CLine
{
protected: // Вызывается только во время сериализации
  CEllipse();
  DECLARE_SERIAL(CEllipse)
public:
  CEllipse(CPoint beg, CPoint end);
  virtual ~CEllipse();
  ...
  virtual void Draw(CDC *pDC);
};
class CGraphicsView : public CScrollView
{
protected: // Вызывается только во время сериализации
  CGraphicsView();
  DECLARE_DYNCREATE(CGraphicsView)
  ...
public:
  virtual void Serialize(CArchive& ar);
  ...
private:
  CTypedPtrList <CPtrList, CDrawObject *> m_List;
};
CDrawObject::CDrawObject()
{
```

```
  m_Selected = FALSE;
}
CDrawObject::CDrawObject(BOOL Selected)
{
  m_Selected = Selected;
}
CDrawObject::~CDrawObject(){}
void CDrawObject::Serialize(CArchive& ar)
{
  if (ar.IsStoring())  {
    ar << m_Selected;
  }  else  {
    ar >> m_Selected;
  }
}
IMPLEMENT_SERIAL(CLine,CObject,1)
CLine::CLine()
{
}
CLine::CLine(CPoint beg, CPoint end) : CDrawObject(FALSE)
{
  m_beg = beg;
  m_end = end;
}
CLine::~CLine(){}
void CLine::Serialize(CArchive& ar)
{
  CDrawObject::Serialize(ar);
  if (ar.IsStoring())  {
    ar <<  m_beg;
    ar <<  m_end;
  }  else  {
    ar >>  m_beg;
    ar >>  m_end;
  }
}
void CLine::Draw (CDC* pDC)
{
  ...
```

```
}
IMPLEMENT_SERIAL(CRectangle,CLine,1)
CRectangle::CRectangle(){}
CRectangle::CRectangle(CPoint beg, CPoint end) : CLine(beg, end){}
CRectangle::~CRectangle(){}
void CRectangle::Draw(CDC* pDC)
{
  ...
}
IMPLEMENT_SERIAL(CEllipse,CLine,1)
CEllipse::CEllipse(){}
CEllipse::CEllipse(CPoint beg, CPoint end) : CLine(beg, end){}
CEllipse::~CEllipse(){}
void CEllipse::Draw(CDC* pDC)
{
  ...
}
void CGraphicsView::Serialize(CArchive& ar)
{
  if (ar.IsStoring())  {
    ar << m_List.GetCount();
    POSITION pos = m_List.GetHeadPosition();
    while (pos != NULL)
      ar << m_List.GetNext(pos);
  }  else  {
    int nCount;
    ar >> nCount;
    while (nCount--)    {
      CObject *pObject = NULL;
      ar >> pObject;
      m_List.AddTail(pObject);
    }
  }
}
```

Итак, каждый класс имеет конструкторы двух типов:

- Конструкторы без параметров (реализация конструкторов по умолчанию), которые объявлены как защищенные
- Общедоступные конструкторы, которые можно использовать для создания объектов по мере необходимости.

Защищенные конструкторы без параметров как раз и вызываются при восстановлении объектов в момент выполнения оператора чтения из архива.

```
ar >> pObject;
```

Собственно чтение в этот момент активизирует не только соответствующие конструкторы, но и функции Serialize уже созданных объектов. Например, вызов конструкторов и функций для восстановления объекта класса CRectangle происходит в следующем порядке:

CObject::CObject( )
CDrawObject::CDrawObject( )
CLine::CLine( )
CRectangle::CRectangle( )
CLine::Serialize( )

Класс CRectangle не переопределяет функцию Serialize и, следовательно, используется реализация функции базового класса.

Далее представлено содержимое файла, в котором располагаются три объекта "прямоугольник" и два объекта "линия" в следующем порядке: "прямоугольник", "линия", "прямоугольник", "прямоугольник" и "линия". Одинарной чертой подчеркнуто описание типа объекта, двойной — ссылка на него в виде однобайтового числа и символа 'A'. Информация о типе объекта содержит только имя соответствующего класса, по которому и происходит вызов нужного конструктора во время исполнения программы, причем только один раз, далее тип определяется номером, указанным в ссылке. Не правда ли, удобно?

```
0000: 05 00 00 00 FF FF 01 00 0A 00 43 52 65 63 74 61 ¦  _...___...CRecta
0010: 6E 67 6C 65 26 00 00 00 19 00 00 00 58 00 00 00 ¦  ngle&..._...X...
0020: 4B 00 00 00 01 00 00 00 00 00 00 00 01 00 00 00 ¦  K..._........_...
0030: 00 00 00 00 00 00 00 00 26 00 00 00 19 00 00 00 ¦  ........&..._...
0040: 58 00 00 00 4B 00 00 00 FF FF 01 00 05 00 43 4C ¦  X...K..._____._.CL
0050: 69 6E 65 26 00 00 00 19 00 00 00 58 00 00 00 4B ¦  ine&..._...X...K
0060: 00 00 00 01 00 00 00 00 00 00 00 01 00 00 00 00 ¦  ..._........_....
0070: 00 00 00 00 00 00 00 26 00 00 00 19 00 00 00 58 ¦  .......&..._...X
0080: 00 00 00 4B 00 00 00 01 80 26 00 00 00 19 00 00 ¦  ...K..._A&..._..
0090: 00 58 00 00 00 4B 00 00 00 01 00 00 00 00 00 00 ¦  .X...K..._......
00A0: 00 01 00 00 00 00 00 00 00 00 00 00 00 26 00 00 ¦  ._..........&..
00B0: 00 19 00 00 00 58 00 00 00 4B 00 00 00 01 80 26 ¦  ._...X...K..._A&
00C0: 00 00 00 19 00 00 00 58 00 00 00 4B 00 00 00 01 ¦  ..._...X...K..._
00D0: 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 00 ¦  ......._........
00E0: 00 00 00 26 00 00 00 19 00 00 00 58 00 00 00 4B ¦  ...&..._...X...K
00F0: 00 00 00 03 80 26 00 00 00 19 00 00 00 58 00 00 ¦  ..._A&..._...X..
0100: 00 4B 00 00 00 01 00 00 00 00 00 00 00 01 00 00 ¦  .K..._......._..
0110: 00 00 00 00 00 00 00 00 00 26 00 00 00 19 00 00 ¦  .........&..._..
0120: 00 58 00 00 00 4B 00 00 00                     ¦  .X...K...
```

Класс CArchive

CArchive

Этот класс позволяет сохранять совокупную схему объектов в непрерывной бинарной форме — обычно в файле на диске. Архивный объект можно представлять как разновидность бинарного потока, и, подобно потоку ввода/вывода, он ассоциируется с файлом и дает возможность для буферизованного чтения/записи данных в/из хранилища. Но в отличие от потока ввода/вывода, обрабатывающего последовательность символов ASCII, архив обрабатывает данные бинарного объекта в эффективном безызбыточном формате.

До создания объекта CArchive необходимо создать объект класса CFile (для одного файла может использоваться только один активный архив). При этом необходимо также обеспечить, чтобы состояние загрузки/сохранения архива было совместимо с режимом открытия файла.

При конструировании объекта CArchive он присоединяется к объекту класса CFile или производного от него, в момент открытия файла. Кроме того, необходимо определить, будет ли архив использоваться для загрузки или сохранения. Объекты этого класса предназначены для работы не только с примитивными типами, но и с объектами классов, производных от CObject, подготовленными для сериализации, т. е. обычно имеющими функцию Serialize и использующими макросы DECLARE_SERIAL и IMPLEMENT_SERIAL.

Как обычно, рассмотрим основные члены этого класса.

CDocument CArchive::**m_pDocument** —

по умолчанию устанавливается в NULL, но по желанию пользователя может указывать на любой документ. Его основное назначение заключается в передаче дополнительной информации в процессе сериализации. Библиотека MFC устанавливает этот член в значение указателя на обрабатываемый (сериализуемый) документ, когда пользователь вызывает команду ID_FILE_OPEN или ID_FILE_SAVE.

CArchive::**CArchive**(
CFile *pFile,
UINT nMode,
int nBufSize = 4096,
void *lpBuf = NULL) —

конструктор; создает объект класса и определяет, будет ли он использоваться для загрузки или сохранения объектов. Это назначение архива не может быть изменено после того, как он был создан. Более того, нельзя изменять также и состояние файла, пока архив не будет закрыт, т. к. любые операции такого рода портят целостность архива. Параметры: pFile — указатель на объект CFile, который является окончательным приемником или источником сохраняемых данных; nBufSize — целое число, определяющее размер (в байтах) внутреннего буфера файла (по умолчанию имеет размер 4096 байт); lpBuf — необязательный указатель на буфер размера nBufSize, если использовать значение по умолчанию (NULL), то архив сам запросит под него требуемый объем памяти и сам же освободит ее при разрушении объекта; nMode — флаг, который определяет, будет ли

объект загружаться из архива или сохраняться в нем, должен иметь одно из следующих значений:

CArchive::**load**	Загружает данные из архива; требует только чтения CFile
CArchive::**store**	Сохраняет данные в архиве; требует записи CFile
CArchive::**bNoFlushOnDelete**	Блокирует архив от автоматического вызова функции Flush при вызове его деструктора; если этот флаг установлен, то до вызова деструктора необходим явный вызов функции Close, в противном случае данные могут быть испорчены

void CArchive::**Abort**( ) —

закрывает архив без обработки исключения. Деструктор объекта вызывает функцию Close, которая принудительно записывает в файл любые несохраненные данные, ассоциированные с объектом CFile, что может вызвать исключение. Если архив создавался динамически (с использованием оператора new), то после закрытия файла необходимо освободить память.

void CArchive::**Close**( ) —

принудительно записывает в файл любые данные, оставшиеся в буфере, закрывает архив и отсоединяет его от файла. После закрытия архива можно создать другой для того же самого файла или закрыть файл. Выполнение этой функции гарантирует, что все данные переданы из архива в файл, и сам архив больше не доступен. Для сохранения данных на устройстве постоянного хранения (чаще всего на диске) сначала необходимо использовать функцию CFile::Close и только после этого разрушить объект с помощью функции CFile.

В классе реализованы несколько вариантов перегруженных операторов >> и <<, которые, соответственно, загружают указанный объект (или примитивный тип) из архива или сохраняют его в архиве. Они имеют достаточно однотипные формы, некоторые из которых представлены ниже:

friend CArchive& **operator** <<(
CArchive &ar,
Cobject *&pOb)

friend CArchive& **operator** >>(
CArchive &ar,
Cobject *&pOb)

CArchive& **operator** <<(Data_Type& data)

CArchive& **operator** >>(Data_Type& data) —

через Data_Type здесь обозначены примитивные типы данных BYTE, WORD, int, DWORD, float и double.

Для работы с более сложными структурами данных в классе реализованы функции:

UINT CArchive::**Read**(
void *lpBuf, UINT nMax) —

читает из архива в буфер (параметр lpBuf) определенное число (параметр nMax) байт. Функция возвращает действительно прочитанное число байт, которое может быть меньше заданного при достижении конца файла. Ее можно использовать внутри переопределенной функции Serialize для чтения простых структур, содержащихся в объекте.

void CArchive::**Write**(
const void *lpBuf, UINT nMax) —

записывает из буфера (параметр lpBuf) в архив определенное число (параметр nMax) байт. Ее можно использовать внутри переопределенной функции Serialize для записи простых структур, которые содержатся в объекте.

Поскольку строки являются одними из наиболее часто используемых типов данных, то для работы с ними реализованы специальные функции: ReadString и WriteString.

Поскольку архив может быть открыт либо для чтения, либо для записи данных (но не одновременно), то необходимо иметь информацию о его "предназначении", что обеспечивается двумя функциями:

BOOL CArchive::**IsLoading**( ) —

определяет, был ли архив открыт для загрузки данных. Вызывается из функций Serialize классов, поддерживающих сериализацию.

BOOL CArchive::**IsStoring**( ) —

определяет, был ли архив открыт для сохранения данных. Вызывается из функций Serialize классов, поддерживающих сериализацию.

```
void CDrawDoc::Serialize(CArchive& ar)
{
  ...  // Действия, относящиеся и к загрузке и к сохранению архива
  if (ar.IsStoring())  {
    ... // Сохранение данных в архиве
  }  else  {
    ... // Загрузка данных из архива
  }
}
```

Теперь, когда мы разобрались в общей структуре рассматриваемой архитектуры, перейдем к вопросам создания и работы с ее основными составляющими — документами и ассоциированными с ними представлениями.

Глава 20

Документ и его представления

Как мы уже неоднократно отмечали, одним из основных достоинств архитектуры "документ/представление" является то, что в ней данные отделены от их изображения.

Любые изменения данных осуществляются посредством специального класса документа, который обеспечивает необходимое пространство для их хранения в памяти, отвечает за запись и чтение документа с диска и предоставляет интерфейс для доступа и обновления данных. За изображение данных на экране, принтере или любом другом устройстве отвечает представление, которое использует для этого специальный класс CView, или производный от него. Можно сказать, что объект этого класса представляет собой окно, посредством которого, с одной стороны, осуществляется взаимодействие с пользователем, а с другой — организуется доступ к интерфейсной части документа для возможности обновления данных. В достаточно упрощенном виде сказанное можно проиллюстрировать с помощью рис. 20.1.

Рис. 20.1. Взаимоотношения документа и представления

Следует также повторить, что один и тот же документ одновременно может иметь несколько различных представлений.

Таким образом, действуя совместно, документ и его представления:

- Содержат, изображают данные приложения и управляют ими
- Предоставляют интерфейс с данными документа
- Участвуют в чтении и записи файлов
- Предоставляют необходимые возможности для печати данных
- Обрабатывают большинство команд и сообщений приложения

Рассмотрим эти основные компоненты архитектуры "документ/представление" подробно и начнем с документов.

Документы

В самом общем случае можно сказать, что документы содержат и управляют данными приложения. Или, другими словами, документ представляет собой некоторую единицу данных, которую пользователь обычно открывает по команде ID_FILE_OPEN и сохраняет по команде ID_FILE_SAVE.

Рис. 20.2. Последовательность действий при создании документа

Для реализации документа в типичном приложении необходимо проделать следующую последовательность действий (рис. 20.2):

- Для каждого типа документа образовать класс на базе CDocument
- Добавить в него переменные для хранения всех данных документа
- Реализовать функции для чтения и модификации этих данных
- Переопределить функцию CObject::Serialize в новом классе документа для организации чтения/записи данных документа с диска/на диск

Данные документа определяются как переменные специального класса документа, производного от CDocument. Рассмотрим, как это реализовано в приложении NoteDraw, в котором используются документы двух типов. Для текстовых документов мы использовали класс библиотеки MFC CTypedPtrList, который представляет собой связанный список указателей на объекты произвольных типов:

```
class CNoteDoc : public CDocument
{
...
// Данные текстовых документов
public:
// Список указателей на объекты класса CParagraph,
// содержащего параметры форматирования абзаца текста
  CTypedPtrList<CObList, CParagraph*> m_listPar;
...
};
```

В данном случае CTypedPtrList содержит указатели на объекты специального созданного класса CParagraph, отвечающего за хранение и использование таких параметров, как текст, его цвет, шрифт и стиль выравнивания для каждого абзаца текста.

Для графических документов используются другие типы данных:

```
class CDrawDoc : public CDocument
{
...
// Данные графических документов
protected:
// Параметры, используемые всеми нарисованными линиями
  int m_nWidth;
  CPen m_curPen;
  COLORREF m_curColor;
public:
// Массив указателей на объекты класса CRect, содержащего
// координаты начальной и конечной точек рисуемой линии
  CArray<CRect*, CRect*> m_lineArray;
```

```
...
//
CPen  *GetPen() {return &m_curPen;}
...
};
```

Помимо самих данных, в классе документа зачастую определяются специальные функции для установки и извлечения элементов данных, а также для выполнения необходимых операций над ними (в приведенном фрагменте функция GetPen).

К последнему действию, необходимому при создании какого-либо документа приложения — переопределению функции Serialize для реализации процесса сериализации, — мы еще вернемся, а пока рассмотрим "цикл жизни" документа в рамках архитектуры "документ/представление":

- Во время динамического создания вызывается конструктор объекта — "документ"
- Для каждого нового документа вызывается функция OnNewDocument или OnOpenDocument
- Пользователь взаимодействует с документом посредством представлений, ассоциированных с ним
- Для удаления данных документа вызывается функция DeleteContents
- Для удаления объекта "документ" вызывается его деструктор

Перечисленные шаги характерны для MDI-приложений. При работе с SDI-приложениями первый шаг выполняется единственный раз — при первом создании документа, а последний — когда приложение завершает свою работу.

Класс CDocument

Этот класс предоставляет базовые функциональные возможности для классов документов, определенных пользователем. Он поддерживает все стандартные операции, такие как создание документа, его загрузка и сохранение. Библиотека MFC работает с документами, используя интерфейс, определенный в CDocument.

Объекты класса CDocument являются частью стандартного маршрута команд и поэтому получают команды от стандартных компонентов интерфейса пользователя. Документ получает команды раньше активного представления и если сам не обрабатывает некоторую команду, то передает ее на обработку своему шаблону.

Рассмотрим функции этого класса.

CDocument::**CDocument**( ) —

создает объект класса. Непосредственно обработку создания документа выполняет библиотека MFC. Для выполнения каких-либо специфических инициализирующих действий необходимо переопределить функцию OnNewDocument, что особенно важно для однодокументных приложений.

CDocTemplate* CDocument::**GetDocTemplate**( ) —

возвращает указатель на объект-шаблон для документа этого типа. Если документ этого типа не поддерживается каким-либо шаблоном, то возвращается NULL.

const CString& CDocument::**GetPathName**( ) —

возвращает полный путь к файлу документа. Строка будет пустой, если документ еще не был сохранен или с ним не ассоциирован никакой файл на диске.

virtual void CDocument::**SetPathName**(
LPCTSTR lpszPathName,
BOOL bAddToMRU = TRUE) —

устанавливает полный путь к файлу документа на диске (параметр lpszPathName). В зависимости от значения параметра bAddToMRU этот путь добавляется (TRUE) или не добавляется (FALSE) в список последних использованных файлов (MRU).

const CString& CDocument::**GetTitle**( ) —

возвращает название документа. Чаще всего оно образуется из имени соответствующего файла или берется из подстроки "Doc type name" соответствующей строки ресурса описания типа.

virtual void CDocument::**SetTitle**(LPCTSTR lpszTitle) —

устанавливает название документа (параметр lpszTitle). При этом происходит обновление заголовков всех фреймов, которые отображают документ.

BOOL CDocument::**IsModified**( ) —

возвращает TRUE, если документ был изменен после последнего сохранения, в противном случае — FALSE.

void CDocument::**SetModifiedFlag**(BOOL bModifitd = TRUE) —

позволяет пометить документ как измененный (bModified = TRUE), что позволит библиотеке MFC гарантировать вывод окна запроса пользователю на сохранение документа до его закрытия. Использование в качестве значения параметра FALSE позволяет пометить документ как неизмененный, даже если в него были внесены какие-либо изменения, не отменяя их фактически.

void CDocument::**AddView**(CView *pView) —

присоединяет представление pView к документу, добавляя его к общему списку представлений, ассоциированных с этим документом. Помимо этого, функция устанавливает указатель m_pDocument класса CView на этот документ. Вызывается библиотекой MFC при создании нового представления документа, обычно обработчиками команд ID_FILE_NEW, ID_FILE_OPEN, ID_WINDOW_NEW или при разбиении разделяемого окна.

void CDocument::**RemoveView**(CView *pView) —

отсоединяет представление pView от документа и удаляет его от общего списка представлений, ассоциированных с этим документом. Указатель m_pDocument класса CView устанавливается в NULL. Вызывается библиотекой MFC при закрытии фрейма или области разделенного окна.

virtual POSITION CDocument::**GetFirstViewPosition**( ) —

позволяет получить позицию первого представления в общем списке представлений, ассоциированных с документом. Используется для возможности начала поиска требуемого представления.

virtual CView* CDocument::**GetNextView**(POSITION &rPosition) —

возвращает указатель на объект класса CView (или производного от него), определяемый его позицией rPosition в списке представлений документа. После этого устанавливает rPosition в значение для следующего представления в списке. Если полученное представление последнее в списке, то rPosition устанавливается в NULL. Позиция представления в списке полностью определяется порядком его записи в него.

```
...
    // Получаем указатель на документ, с которым
    // ассоциировано текущее представление
    CNoteDoc* pDoc = (CNoteDoc*)GetDocument();
    // Получаем позицию первого представления, хранящегося в
    // списке представлений, ассоциированных с документом
    POSITION pos = pDoc->GetFirstViewPosition();
    // Получаем указатель на первое представление в списке
    pDoc->GetNextView(pos);
    // Получаем указатель на второе представление в списке
    CTextView *pText = (CTextView *)pDoc->GetNextView(pos);

    ...

    // Читаем данные представления
    pText->GetSelectedText(str);
...
```

void CDocument::**UpdateAllViews**(
CView *pSender,
LPARAM lHint = 0L,
CObject *pHint = NULL) —

информирует каждое представление, присоединенное к документу, за исключением изменившего документ, на которое указывает pSender о том, что документ был изменен. Это приводит к вызову функции CView::OnUpdate для каждого представления (за исключением определяемого параметром pSender), передавая им информацию об изменениях в документе. Эта информация может быть представлена каким-либо кодом (параметр lHint) и/или содержаться в объекте (параметр pHint).

```
void CNoteView::OnPageSetup()
{
...
// После того как были изменены параметры страницы,
// необходимо обновить изображение данных на экране
    CNoteDoc* pDoc = (CNoteDoc*)GetDocument();
// Сообщение об обновлении передается всем представлениям,
// ассоциированным с документом, но реально предназначается
// только объекту класса, на который указывает this
    pDoc->UpdateAllViews(NULL, 0, this);
...
}
void CNoteView::OnUpdate(CView*, LPARAM, CObject *pHint)
{
// Если определено представление, требующее перерисовки,
  if(pHint != NULL)
    // и оно является объектом класса CNoteView ...
    if(pHint->IsKindOf(RUNTIME_CLASS(CNoteView)))
    {
      // ... изменяем размеры этого представления
      ...
     }
  ...
}
```

virtual BOOL CDocument::**CanCloseFrame**(CFrameWnd *pFrame) —

вызывается библиотекой MFC до закрытия фрейма документа. Функция проверяет все фреймы, которые отображают документ, и если в нем были какие-либо изменения, выводит на экран окно запроса на сохранение. Параметр pFrame указывает на фрейм представления, присоединенного к документу.

virtual void CDocument::**DeleteContents**( ) —

удаляет данные документа без разрушения самого объекта. Вызывается библиотекой MFC непосредственно перед разрушением документа, а также гарантирует, что документ пустой, если есть необходимость в его многократном использовании. Для выполнения "осмысленных" действий необходимо переопределить эту функцию, т. к. реализация по умолчанию не делает ничего.

```
void CNoteDoc::DeleteContents()
{
// Находим позицию первого абзаца в списке
  POSITION pos = m_listPar.GetHeadPosition();
  while(pos != NULL)
```

```
    {
      // Последовательно проходим весь список
      CParagraph *pPar = m_listPar.GetNext(pos);

      // Освобождаем память, занимаемую текущим абзацем.
      // Если этого не сделать, то для системы память
      // будет потеряна, хотя Windows 95/98 и Windows NT
      // в большинстве случаев "уберут" за вами
      delete pPar;
    }
    // Очищаем список - делаем его "пустым"
    m_listPar.RemoveAll();

  // Вызываем функцию базового класса для корректного завершения
    CDocument::DeleteContents();
}
```

virtual void CDocument::**OnChangedViewList( )** —

вызывается библиотекой MFC после того, как представление было добавлено в документ или удалено из него. В том случае, если текущее представление было последним в списке, удаляется весь документ.

virtual void CDocument::**OnCloseDocument( )** —

вызывается библиотекой MFC, когда документ закрывается. Реализация по умолчанию вызывает функцию DeleteContents для удаления данных документа и закрывает все фреймы представлений, которые к нему присоединены.

virtual BOOL CDocument::**OnNewDocument( )** —

вызывается библиотекой MFC при обработке команды ID_FILE_NEW. Реализация по умолчанию вызывает функцию DeleteContents, которая обеспечивает очистку документа, и соответствующим образом помечает его. При переопределении здесь рекомендуется инициализировать структуру данных нового документа. Для SDI-приложений эта функция вместо создания нового переинициализирует существующий объект документа. Для MDI-приложений библиотека MFC каждый раз создает новый объект документа, и в этой функции производится его инициализация.

```
BOOL CNoteDoc::OnNewDocument()
{
// Всю стандартную обработку по созданию документа
// возлагаем на библиотеку MFC - она делает это прекрасно!
  if (!CDocument::OnNewDocument())
    return FALSE;
  // После успешного создания проводим "настройку"
  // конкретного документа
```

```
    initDoc();

    return TRUE;
  }
```

virtual BOOL CDocument::**OnOpenDocument**(LPCTSTR lpszPathName) —

вызывается библиотекой MFC при обработке команды ID_FILE_OPEN. Реализация по умолчанию открывает файл, имя которого определяется параметром lpszPathName, вызывает функцию DeleteContents, обеспечивающую очистку документа, функцию Serialize для чтения содержимого файла и помечает документ как немодифицированный. Переопределять ее следует, если используется какой-либо специальный механизм работы с архивами и/или файлами, например, распознавание формата документа.

virtual BOOL CDocument::**OnSaveDocument**(LPCTSTR lpszPathName) —

вызывается библиотекой MFC при обработке команд ID_FILE_SAVE или ID_FILE_SAVE_AS. Реализация по умолчанию открывает файл, имя которого определяется параметром lpszPathName, вызывает функцию Serialize для записи в файл данных документа и помечает его как немодифицированный.

Следующая группа функций достаточно редко используется в явном виде, и мы приводим их для полноты картины.

virtual CFile* CDocument::**GetFile**(
LPCTSTR lpszFileName,
UINT nOpenFlags,
CFileException pError) —

позволяет получить указатель на объект CFile по абсолютному или относительному пути lpszPathName к требуемому файлу. Параметр nOpenFlags задает режим доступа и совместного использования, а pError указывает на объект "исключение" при работе с файлом, в который записывается состояние завершения операции.

virtual void CDocument::**ReleaseFile**(
CFile *pFile,
BOOL bAbort) —

освобождает файл pFile, делая его доступным для других приложений. Параметр bAbort определяет функцию, при помощи которой будет освобождаться файл: FALSE — если используется функция CFile::Close, TRUE — при вызове функции CFile::Abort.

virtual void CDocument::**PreCloseFrame**(CFrameWnd *pFrame) —

вызывается библиотекой MFC до разрушения фрейма pFrame, который владеет ассоциированным с ним документом. При переопределении этой функции желательно вызвать также соответствующую функцию базового класса.

Еще раз внимательно посмотрите на рис. 20.2. На нем наглядно показано, что документ может быть создан в двух случаях. Прежде всего, по команде ID_FILE_NEW создается новый, пустой документ. В этом случае для его инициализации достаточно переопределить функцию класса документа OnNewDocument. Во втором случае по команде ID_FILE_OPEN создается новый документ, содержимое которого загружается из файла. При этом для

его инициализации можно переопределить функцию класса документа OnOpenDocument. Этот случай представляет больший интерес, т. к. вплотную подводит нас к достаточно важному понятию не только в рамках архитектуры "документ/представление", но и во всей библиотеке классов MFC. Речь, конечно же, идет о сериализации, или, другими словами, преобразовании в последовательную форму. Мы уже рассматривали этот процесс в предыдущей главе, поэтому напомним лишь ключевые моменты. Основная идея сериализации заключается в том, чтобы обеспечить сохранение и восстановление текущего состояния объектов на устройстве постоянного хранения, например, в файле на диске. Поскольку речь идет о "преобразовании в последовательную форму", то очевидно, что состояние объекта сохраняется в бинарном формате. Следующим важным моментом, на котором мы остановимся, является то, что необходимо включить специальные макросы в объявление:

```
class CParagraph : public CObject
{
...
// Обязательный макрос, обеспечивающий поддержку сериализации
  DECLARE_SERIAL(CParagraph);
...
};
```

и реализацию класса:

```
// Обязательный макрос, обеспечивающий поддержку сериализации
IMPLEMENT_SERIAL(CParagraph, CObject, 2)
```

И, наконец, необходимо переопределить функцию Serialize, которая имеется у объекта любого класса, производного от CObject:

```
void CNoteDoc::Serialize(CArchive& ar)
{
  CNDApp *pApp = (CNDApp*)AfxGetApp();
  if (ar.IsStoring())  {
    // Сохраняем текущий размер бумаги
    ar << pApp->m_sizePaper;
  }  else  {
    // Восстанавливаем текущий размер бумаги
    ar >> pApp->m_sizePaper;
  }
  // Для сохранения и восстановления собственно содержимого
  // документа вызываем функцию Serialize класса CObList,
  // который, в свою очередь, последовательно вызывает
  // аналогичную функцию для каждого абзаца.
  // Последнюю тоже необходимо было переопределить,
```

```
  // чтобы сохранить данные конкретного документа
  m_listPar.Serialize(ar);
}
void CParagraph::Serialize(CArchive& ar)
{
  LOGFONT lf;
  if (ar.IsStoring())  {
    m_font.GetLogFont(&lf);
    ar << m_align;
    ar << m_color;
    ar << lf.lfCharSet;
    ar << lf.lfItalic;
    ar << lf.lfUnderline;
    ar << lf.lfStrikeOut;
    ar << lf.lfWeight;
    ar << lf.lfFaceName;
    ar << (long)(lf.lfHeight/2.3);
    ar << m_words;
  }  else  {
    CString tmp;
    memset(&lf, 0, sizeof(LOGFONT));
    ar >> m_align;
    ar >> m_color;
    ar >> lf.lfCharSet;
    ar >> lf.lfItalic;
    ar >> lf.lfUnderline;
    ar >> lf.lfStrikeOut;
    ar >> lf.lfWeight;
    ar >> tmp;
    strcpy(lf.lfFaceName, tmp);
    ar >> lf.lfHeight;
    lf.lfHeight = (long)(lf.lfHeight*17.55);
    ar >> m_words;
    m_font.CreatePointFontIndirect(&lf);
  }
}
```

Примечание

Последний представленный фрагмент приведен нами для иллюстрации следующего положения. При сохранении параметров выбранного шрифта, которые хра-

нятся в объекте класса CFont, было бы логично ожидать, что при использовании операторов >> и << архива, произошло бы сохранение (и восстановление) объекта m_font, поскольку класс CFont является производным от CObject и, следовательно, поддерживает сериализацию. Однако этого не происходит, т. к. в самом этом классе не переопределена функция Serialize, а аналогичные функции базовых классов, естественно, корректно не работают. Поэтому, прежде чем использовать сериализацию классов, производных от CObject, проверьте, реализована ли в них эта, такая удобная, а во многих случаях и необходимая, функция.

Закончив знакомство с документами, пора переходить к тому, каким образом можно взаимодействовать с хранящимися в них данными. А осуществить это взаимодействие и визуализацию данных документа призваны объекты специальных классов – представления.

Представления

Это специальная группа классов дочерних окон фрейма, которая отвечает за отображение данных документа и за взаимодействие с пользователем. В этой главе мы представим те возможности, доступные для широкого применения, которыми располагает библиотека классов MFC.

Представление ассоциируется с некоторым документом и действует как посредник между ним и пользователем. Другими словами, представление переводит образ документа на экран, принтер или любое другое устройство графического вывода, и интерпретирует действия пользователя как операции над документом. Основным, на что здесь следует обратить внимание, является то, что представление может быть *ассоциировано* (или *присоединено*) только с *одним* документом, т. е. хотя оно и является окном, как, впрочем, почти все видимые объекты Windows, но при этом ориентировано на работу только с определенным набором данных. С другой стороны, в рамках одного фрейма документа можно создавать и использовать сколько угодно представлений для работы с одним и тем же документом.

Достаточно часто возникает желание иметь различные типы представлений одного, данного, типа документа. Что имеется в виду? Например, при работе с текстовым процессором желательно иметь одно представление для текста документа, а другое (контурное) — для изображения области колонтитулов. Эти различные типы представлений могут размещаться как в различных фреймах, так и в различных областях одного и того же фрейма. В обоих случаях достаточно один раз сопоставить представление с документом, но организовать различную обработку данных документа (рис. 20.3).

Таким образом, представление отвечает за изображение и модификацию данных документа, но не имеет никакого отношения к их хранению. (Точнее, не

должно иметь.) Необходимые подробности о различных параметрах этих данных хранятся вместе с самим документом, откуда либо само представление, либо специальные функции класса документа, обеспечивающие для того или иного представления доступ к ним, берут только необходимое.

Рис. 20.3. Использование нескольких представлений для отображения данных одного документа

Несколько слов о взаимодействии с пользователем. Реализация выполнения любых действий такого рода целиком ложится на соответствующий класс представления, а, значит, и на программиста. (Никто не может предугадать всех ваших желаний, даже разработчики прекрасной библиотеки классов MFC.) Это может быть ввод с клавиатуры, действия мышью, выбор элемента меню, нажатие кнопки панели инструментов или работа с полосой прокрутки. Соответствующие команды представление получает от своего фрейма (правда, не напрямую, а опосредованно). Обработку сообщений оно осуществляет аналогично всем другим получателям команд — посредством карты сообщений, что подробно описано нами в первом томе. Если пред-

ставление само не обрабатывает команду, то передает ее дальше "по маршруту", как описано в главе 8.

Перед знакомством с классами представлений (и связанных с ними), приведем последовательность действий по их созданию, выполняемую библиотекой классов MFC (рис. 20.4).

Рис. 20.4. Последовательность создания представления

Теперь, после того как сложилось некоторое общее представление о назначении и возможностях этой группы классов, можно переходить к рассмотрению компонентов этой группы и свойств ее составляющих.

В полном соответствии с общей идеологией построения библиотеки MFC на вершине классов представлений стоит единственный базовый класс — CView, обеспечивающий поддержку печати, общее взаимодействие с документом и некоторые другие возможности, которые будут описаны ниже. От него образованы еще два класса — CCtrlView и CScrollView, каждый из которых унаследовал все свойства базового класса и добавил свои специфические. Класс CCtrlView был добавлен в библиотеку классов MFC начиная с версии 4.0 для того, чтобы можно было использовать в архитектуре "документ/представление" деревья (класс CTreeView), списки (CListView), а также простейший (CEditView) и расширенный (CRichEditView) элементы управления для редактирования текстов. Класс CScrollView добавляет к свойствам базового класса поддержку автоматической прокрутки документов, если данные документа, которые необходимо отобразить, не помещаются в одном окне.

Примечание

Помимо приведенных здесь классов представлений, в библиотеке MFC реализованы также некоторые другие. Производный от CScrollView класс — CFormView

обеспечивает дополнительную поддержку диалоговых ресурсов, таких как кнопки и поля. Наибольшее применение этот класс находит при работе с базами данных, о чем свидетельствуют его производные классы: CDaoRecordView COleDBRecordView и CRecordView. Кроме того, в рассматриваемой версии добавился класс CHtmlView. Более подробно эти классы будут рассмотрены при обсуждении соответствующих вопросов.

Таким образом, в библиотеке классов MFC реализованы двенадцать классов представлений, из которых здесь мы подробно рассмотрим семь. Кроме того, здесь же будет рассмотрен класс CSplitterWnd, который хотя и не входит в группу классов представлений, но достаточно тесно с ними связан.

Класс CView

Этот класс предоставляет базовые функциональные возможности для всех классов представлений, которые есть в библиотеке или определяются пользователем. Рассмотрим основные функции этого класса.

Первым, как всегда, идет конструктор.

CView::**CView( )** —

создает объект класса; вызывается, когда создается новый фрейм или окно разделяется на области. Никакой инициализации представления здесь не производится.

CDocument* CView::**GetDocument( )** —

позволяет получить указатель на объект "документ", присоединенный к этому представлению. Если с ним не ассоциирован никакой документ, то возвращается NULL. Использование этого указателя предоставляет доступ к функциям соответствующего класса документа. Для каждого производного класса создается своя специфичная функция.

virtual void CView::**OnInitialUpdate( )** —

вызывается библиотекой MFC после того, как представление первый раз присоединено к документу, но до его первоначального отображения. Реализация по умолчанию вызывает функцию CView::OnUpdate без какой-либо информации (lHint = 0 и pHint = NULL). Для проведения специальных инициализирующих действий необходимо переопределить эту функцию. Например, так:

```
void CNoteView::OnInitialUpdate()
{
// Пусть библиотека MFC выполнит за нас всю
// необходимую предварительную работу
  CScrollView::OnInitialUpdate();
// Создаем новый (пустой) абзац
  CNoteDoc* pDoc = (CNoteDoc*)GetDocument();
  CParagraph *pPar = pDoc->OnNewPar(ND_LEFT);
// Добавляем его в общий список абзацев
  pDoc->m_listPar.AddTail(pPar);
```

```
// Обновляем все изображения данных документа
  pDoc->UpdateAllViews(NULL, 0, this);
}
```

virtual void CView::**OnUpdate**(
CView *pSender,
LPARAM lHint,
CObject *pHint) —

вызывается из CDocument::UpdateAllViews после того, как документ был модифицирован. Кроме того, она вызывается из функции CView::OnInitialUpdate. Реализация по умолчанию помечает всю рабочую область как недействительную для ее перерисовки при получении следующего сообщения WM_PAINT. Если обновления требует не вся рабочая область, то следует переопределить эту функцию, посылая информацию об изменениях через параметры lHint и pHint. Параметр lHint определяет специальное значение, обычно битовую маску или перечислимый тип, несущее информацию о характере изменений в документе. Параметр pHint определяет указатель на объект, производный от CObject, который хранит информацию об изменениях в документе. При переопределении этой функции можно воспользоваться функцией CObject::IsKindOf для получения типа времени выполнения этого объекта. Если оба эти параметра нулевые, то документ послал общее извещение об обновлении. Представление, получившее такое извещение или не сумевшее декодировать параметры, обновляет всю свою рабочую область. Параметр pSender идентифицирует представление, которое должно быть обновлено в соответствии с изменениями в документе.

Примечание

Обратите внимание! Параметр pSender рассмотренной ранее функции CDocument::UpdateAllViews исключает представление, на которое указывает, из группы представлений, получающих сообщение об обновлении, а параметр pSender функции CView::OnUpdate, наоборот, идентифицирует представление, получившее это сообщение.

```
void CNoteView::OnUpdate(CView*, LPARAM, CObject *pHint)
{
  if(pHint != NULL)  {
    // Нам ли предназначено сообщение?
    if(pHint->IsKindOf(RUNTIME_CLASS(CNoteView)))
    {
      // Здесь можно проводить любые необходимые действия,
      // необходимые для инициализации представления CNoteView
    }
  }

  // Обновляем все, хотя можно и ограничить размеры области
  Invalidate(TRUE);
}
```

В этой функции можно выполнять непосредственно и "перерисовку" данных документа. Однако для этих целей предназначены другие функции. Здесь лучше просто описать (в координатах устройства) прямоугольную область, требующую перерисовки, и передать ее в функцию InvalidateRect, чтобы осуществить рисование при получении следующего сообщения WM_PAINT.

virtual void CView::**OnActivateView**(
BOOL bActivate,
CView *pActivateView,
CView *pDeactiveView) —

вызывается библиотекой MFC, когда представление активизируется (bActivate=TRUE) или деактивизируется. По умолчанию функция устанавливает фокус на активизируемое представление. В качестве параметров она получает также указатели на активизируемое (параметр pActivateView) и деактивизируемое (параметр pDeactiveView) представления. Эти параметры указывают на одно и то же представление, если активизируется главное окно приложения без изменения активного представления, что можно использовать для изменения свойств представления. Но эти параметры отличаются при переключении между различными представлениями одного и того же приложения (или дочерними окнами MDI-приложения). Наиболее часто это связано с "разделяемыми" (splitter) окнами.

virtual void CView::**OnActivateFrame**(
UINT nState,
CFrameWnd *pFrameWnd) —

вызывается в случае, когда активизируется или деактивизируется фрейм (параметр pFrameWnd), ассоциированный с текущим представлением. Характер действия определяется параметром nState, который может принимать одно из значений:

WA_INACTIVE	Фрейм деактивизируется
WA_ACTIVE	Фрейм активизируется любым способом, как от нажатия в нем кнопки мыши, так и от клавиатуры
WA_CLICKACTIVE	Фрейм активизируется путем нажатия кнопки мыши в его границах

virtual BOOL CView::**OnScroll**(
UINT nScrollCode,
UINT nPos,
BOOL bDoScroll) —

вызывается библиотекой классов MFC для определения возможности прокрутки в двух случаях, в зависимости от значения параметра bDoScroll, которое определяет, нужно ли действительно осуществлять прокрутку. В первом случае (bDoScroll = TRUE), когда представление получает сообщение о прокрутке, действительно нужно осуществить прокрутку представления. Во втором случае (bDoScroll = FALSE), когда элемент OLE переносится в область автопрокрутки, осуществлять саму прокрутку представления не нужно. Параметр nScrollCode определяет код прокрутки и состоит из двух частей: младший байт задает код прокрутки по горизонтали, а старший — по вертикали. Он может принимать одно из следующих значений:

SB_BOTTOM	Прокрутка до самого низа
SB_TOP	Прокрутка до самого верха
SB_LINEDOWN	Прокрутка вниз на одну линию (строку)

SB_LINEUP	Прокрутка вверх на одну линию (строку)
SB_PAGEDOWN	Прокрутка вниз на одну страницу
SB_PAGEUP	Прокрутка вверх на одну страницу
SB_THUMBTRACK	Перенос ползунка полосы прокрутки в определенную позицию, указанную в параметре nPos

Функция возвращает TRUE, если прокрутка действительно осуществлена; в противном случае — FALSE.

virtual void CView::**OnDraw**(CDC *pDC) = 0 —

чисто виртуальная функция, которая используется для изображения образа документа. Следует *обязательно переопределить* ее для изображения представления документа, что и сделано в производных классах. Библиотека MFC использует эту функцию как для печати (и предварительного просмотра) документа, так и для отображения его на экране. Это основная и единственная функция для изображения видимого образа документа. От того, каким образом она реализована, зависит, что увидит пользователь на экране или после печати на принтере.

```
void CDrawView::OnDraw(CDC* pDC)
{
  CRect *pLine;
  // Поскольку отобразить нужно данные документа,
  // то получаем указатель на него
  CDrawDoc *pDoc = (CDrawDoc*)GetDocument();
  // Мы могли нарисовать несколько объектов,
  // и все их нужно вывести на экран, поэтому организуем цикл
  for(int i = 0; i< pDoc->m_lineArray.GetSize(); i++)       {
    // Получаем координаты текущей линии
    pLine = pDoc->m_lineArray[i];
    // Выбираем в контекст устройства, которым может быть
    // как экран, так и принтер, текущий "карандаш"
    CPen *pOld = pDC->SelectObject(pDoc->GetPen());
    // Выводим текущую линию на экран
    pDC->MoveTo(pLine->left,  pLine->top);
    pDC->LineTo(pLine->right, pLine->bottom);
    // Восстанавливаем параметры старого "карандаша"
    pDC->SelectObject(pOld);
  }
}
```

Внимательно ознакомившись с представленными функциями, нетрудно заметить, что многие из них должны быть переопределены в производных классах, что и сделано в оставшихся классах группы представлений.

Класс CCtrlView

Это базовый класс для классов элементов управления, включающих CEditView, CListView, CRichEditView и CTreeView, которые специально адаптированы к архитектуре "документ/представление". Его назначение во многом напоминает назначение класса CControlBar для панелей элементов управления. Для самого класса CCtrlView приведем две общедоступные переменные и только конструктор:

CString CCtrlView::**m_strClass** —

содержит имя класса окна Windows для класса представления.

DWORD CCtrlView::**m_dwDefaultStyle** —

содержит стиль, устанавливаемый по умолчанию для класса представления. Этот стиль применяется при создании соответствующего окна.

CCtrlView::**CCtrlView**(
LPCTSTR lpszClass,
DWORD dwStyle) —

создает объект класса. Этот конструктор вызывается библиотекой MFC при создании нового фрейма или разбиении окна на области. Для инициализации представления, после того как оно присоединено к соответствующему документу, необходимо переопределить функцию CView::OnInitialUpdate, а для создания объекта Windows следует использовать функцию CWnd::Create или CWnd::CreateEx. Передаваемые в конструктор параметры определяют имя класса Windows — lpszClass и стиль, используемый по умолчанию, — dwStyle и записывают эти значения в соответствующие переменные класса.

Помимо перечисленных членов класса, в нем еще переопределены функции OnDraw и PreCreateWindow, а также реализован обработчик OnPaint сообщения WM_PAINT.

Как видите, этот класс служит лишь для установки общих параметров, а вся основная работа ложится на его производные классы, к рассмотрению которых мы и переходим.

Класс CEditView

Этот класс определяет представление, которое, подобно классу CEdit, инкапсулирует функциональные возможности элемента управления EDIT и может быть использовано для реализации простейшего текстового редактора. При этом, поскольку он создан на базе CView, объекты этого класса могут использоваться с документами и шаблонами документов, как и любое другое представление. Помимо тех возможностей, которые он унаследовал от своих "родительских" классов, в нем дополнительно реализована поддержка функций поиска и замены фрагментов текста, а также обработка стандартных команд ID_EDIT_SELECT_ALL, ID_EDIT_REPLACE, ID_EDIT_REPEAT, ID_FILE_PRINT и ID_EDIT_FIND.

Присущие элементу управления EDIT ограничения в полной мере относятся и к рассматриваемому классу. Объект класса CEditView (или производного от него) имеет следующие ограничения:

- Не поддерживает редактирование по принципу WYSIWYG (What You See Is What You Get, "что видишь, то и получаешь"), т. к. предназначен, в основном, для чтения с экрана
- Может изображать текст только одним шрифтом и не поддерживает специальное форматирование символов
- Так же, как для элемента управления EDIT, размер текста, который может содержать объект класса CEditView, ограничен

После небольшого обзора, в результате которого должно было сложиться общее представление о возможностях класса, рассмотрим его основные члены.

static AFX_DATA const DWORD CEditView::**dwStyleDefault** —

определяет стиль, используемый по умолчанию, для всех объектов класса CEditView.

CEditView::**CEditView**( ) —

конструктор, создает объект класса. После создания объекта, но до использования элемента управления EDIT необходимо вызвать функцию CWnd::Create для создания окна Windows. Если на базе этого класса определен производный, который добавлен в шаблон документа при помощи функции CWinApp::AddDocTemplate, то библиотека сама вызывает и конструктор и функцию Create.

Рассматриваемый класс до появления 32-разрядных версий Windows являлся одним из наиболее часто используемых и потому достаточно полно адаптирован для различных режимов работы. В нем реализована поддержка печати, предварительного просмотра, установка величины табуляции и многое другое.

void CEditView::**SetTabStops**(int nTabStops) —

устанавливает величину табуляции (параметр nTabStops) в единицах блока диалога (1/4 среднего значения ширины символов используемого шрифта), используемую и для экрана, и для принтера. В этом классе поддерживается только одно значение табуляции, в отличие от класса CEdit.

void CEditView::**GetSelectedText**(CString &strResult) —

копирует выделенный текст или символы, предшествующие первому символу "возврат каретки", в объект класса CString, на который ссылается параметр strResult.

UINT CEditView::**GetBufferLength**( ) —

возвращает число символов (без завершающего нулевого), находящихся в текущий момент в буфере элемента управления EDIT.

LPCTSTR CEditView::**LockBuffer**( ) —

блокирует буфер элемента управления EDIT в определенном месте памяти, запрещая его модификацию и перемещение, и возвращает указатель на него.

void CEditView::**UnlockBuffer**( ) —

разрешает модификацию и перемещение буфера, заблокированного функцией LockBuffer.

void CEditView::**SerializeRaw**(CArchive &ar) —

служит для сериализации содержимого объекта CEditView в текстовый файл. Параметр ar хранит преобразованный текст. Эта функция является внутренней реализацией функции Serialize для класса CEditView и отличается от нее тем, что читает и записывает только текст без предварительного описания данных объекта. Таким образом, для этого представления данные хранятся именно в нем, а не в документе.

Еще одна группа функций облегчает работу с блоком диалога **Find/Replace** (Найти и заменить), который выводится на экран в ответ на стандартные команды ID_EDIT_FIND и ID_EDIT_REPLACE, предоставляя программистам обработчики команд кнопок **Find** (найти), **Replace** (заменить) и **Replace All** (заменить все).

BOOL CEditView::**FindText**(
LPCTSTR lpszFind, BOOL bNext = TRUE,
BOOL bCase = TRUE) —

осуществляет поиск текста, заданного параметром lpszFind, в буфере элемента управления, начиная с текущей позиции и в направлении, определяемом параметром bNext; если он имеет значение TRUE, то поиск осуществляется от начала к концу буфера, в противном случае — наоборот. Параметр bCase определяет, следует ли различать строчные и прописные буквы при поиске (TRUE — учитывать регистр, FALSE — не учитывать). Если в результате проведенного поиска текст не найден, то функция возвращает FALSE. Обычно она вызывается из переопределенной функции OnFindText.

virtual void CEditView::**OnFindText**(
LPCTSTR lpszFind, BOOL bNext = TRUE,
BOOL bCase = TRUE) —

аналогична предыдущей функции, однако вызывается в ответ на нажатие кнопки **Find Next** (следующий) в стандартном блоке диалога **Find** (поиск), выводимом на экран в ответ на выбор команды ID_EDIT_FIND. По умолчанию просто вызывает функцию FindText. Если текст не найден, то вызывается функция OnTextNotFound.

virtual void CEditView::**OnReplaceAll**(
LPCTSTR lpszFind, LPCTSTR lpszReplace,
BOOL bCase) —

вызывается, когда пользователь выбирает кнопку **Replace All** в стандартном блоке диалога **Replace** (замена). Она ищет в буфере текст, заданный параметром lpszFind, начиная с текущей позиции, и, при нахождении, заменяет его на другой, определенный в lpszReplace. Параметр bCase задает чувствительность к регистру. Поиск осуществляется посредством вызова функции FindText. В случае, если заданный текст не найден, вызывается функция OnTextNotFound. Для выполнения каких-либо специфических действий эту функцию следует переопределить.

virtual void CEditView::**OnReplaceSel**(
LPCTSTR lpszFind, BOOL bNext,
BOOL bCase, LPCTSTR lpszReplace) —

вызывается, когда пользователь выбирает кнопку **Replace** (заменить) в стандартном блоке диалога **Replace** (замена). После замены *выделенного* текста функция ищет следующий фрагмент, совпадающий с определяемым параметром lpszFind в направлении, определяемом параметром bNext. Сам поиск осуществляется посредством функции FindText. В случае, если заданный текст не найден, вызывается функция OnTextNotFound. Для выполнения каких-либо специфических действий эту функцию следует переопределить.

virtual void CEditView::**OnTextNotFound**(LPCTSTR lpszFind) —

вызывается библиотекой MFC, если текст, определенный в параметре lpszFind, не найден. По умолчанию вызывает функцию Windows MessageBeep. Если нужны какие-либо другие действия, функцию следует переопределить.

Работа с представлением, реализованным на базе класса CEditView, практически аналогична работе с элементом управления EDIT. Поэтому не будем останавливаться на этом классе более подробно, а в качестве примера работы с классом рассмотрим фрагменты кода приложения NoteDraw, которые демонстрируют, каким образом осуществляется организация режимов поиска и замены в рассматриваемом представлении. На рис. 20.5 и 20.6 представлен внешний вид блоков диалога **Find** и **Replace**, соответственно в режимах поиска и замены.

Прежде всего необходимо зарегистрировать новое сообщение.

```
// Для работы с блоком диалога Find или Replace
// при регистрации сообщения необходимо использовать
// строковую константу FINDMSGSTRING
static const UINT nNewMsg =
                ::RegisterWindowMessage(FINDMSGSTRING);
```

В карту сообщений класса необходимо добавить макрокоманды, связывающие команды с соответствующими обработчиками:

```
BEGIN_MESSAGE_MAP(CTextView, CEditView)
  //{{AFX_MSG_MAP(CTextView)
  ...
  ON_COMMAND(ID_EDIT_FIND, OnEditFind)
  ON_COMMAND(ID_EDIT_REPLACE, OnEditReplace)
  //}}AFX_MSG_MAP
  ON_REGISTERED_MESSAGE(nNewMsg, OnNewCmd)
END_MESSAGE_MAP()
```

И, наконец, в реализацию класса необходимо добавить обработчики команд:

```
void CTextView::OnEditFind()
{
// Создаем объект "блок диалога"
  CFindReplaceDialog *pDlg = new CFindReplaceDialog;
  bFindOnly = TRUE;
```

```
// Создаем объект Windows для блока диалога Find
  pDlg->Create(TRUE, NULL, NULL, FR_HIDEWHOLEWORD | FR_DOWN, this);
}
```

Рис. 20.5. Стандартный блок диалога **Find**

```
void CTextView::OnEditReplace()
{
// Создаем объект "блок диалога"
  CFindReplaceDialog *pDlg = new CFindReplaceDialog;
  bFindOnly = FALSE;
  // создаем объект Windows для блока диалога Replace
  pDlg->Create(FALSE, NULL, NULL, FR_NOWHOLEWORD | FR_DOWN, this);
}
LRESULT CTextView::OnNewCmd(WPARAM, LPARAM lParam)
{
// Получаем указатель на объект "стандартный блок диалога"
// Find/Replace
  CFindReplaceDialog* pDialog =
                        CFindReplaceDialog::GetNotifier(lParam);

  if (pDialog->IsTerminating())  {
    // Если пользователь завершил работу по поиску/замене, то
    // сообщаем библиотеке, что объект "блок диалога" больше не нужен
    pDialog = NULL;
  }
  else if (pDialog->FindNext())  {
    // Получаем строку, которую требуется найти
    CString strFind = pDialog->GetFindString();
    // Осуществляем поиск.
    // Параметры поиска получаем прямо из блока диалога
    OnFindNext(strFind,
               pDialog->SearchDown(), pDialog->MatchCase());
  }
  else if (pDialog->ReplaceCurrent())  {
```

```
    // Сюда вход разрешен только для режима замены
    ASSERT(!bFindOnly);
    // Пользователь нажал кнопку Replace: требуется
    // найти определенную строку и заменить ее на заданную.
    // Параметры получаем прямо из блока диалога
    OnReplaceSel(pDialog->GetFindString(),
                 pDialog->SearchDown(), pDialog->MatchCase(),
                 pDialog->GetReplaceString());
  }
  else if (pDialog->ReplaceAll())
  {
    ASSERT(!bFindOnly);

    // Пользователь нажал кнопку Replace All:
    // требуется найти все вхождения определенной строки
    // и заменить их на заданную.
    // Параметры получаем прямо из блока диалога
    OnReplaceAll(pDialog->GetFindString(),
                 pDialog->GetReplaceString(),
                 pDialog->MatchCase());
  }
  return 0;
}
void CTextView::OnTextNotFound(LPCTSTR lpszFind)
{
  // Если искомая строка не найдена, то выводим
  // сообщение об этом
  AfxMessageBox("Текст не найден");
}
```

Рис. 20.6. Стандартный блок диалога **Replace**

Классы CListView и CTreeView

Эти классы позволяют значительно упростить использование элементов управления Windows — LIST VIEW и TREE VIEW и инкапсулировавших их функциональные возможности классов CListCtrl и CTreeCtrl в архитектуре "документ/представление".

Они содержат всего по две функции:

CListView::**CListView**( )

и

CTreeView::**CTreeView**( ) —

создают объект соответствующего класса.

CListCtrl& CListView::**GetListCtrl**( )

и

CTreeCtrl& CTreeView::**GetTreeCtrl**( ) —

позволяют получить доступ к элементам управления LIST VIEW и TREE VIEW, ассоциированным с соответствующими представлениями.

Все остальные возможности они наследуют от своих базовых классов и соответствующих элементов управления.

Класс CScrollView

Согласитесь, что достаточно редко нам "хватает" одного окна для отображения всей необходимой информации. Гораздо чаще мы вынуждены организовать либо прокрутку документа, либо изменение его масштаба, либо чередовать и то и другое. Если базироваться на классе CView, то решение такой задачи целиком ложится на плечи программиста: необходимо самому обрабатывать сообщения от полос прокрутки, используя для этого функции CWnd::OnHScroll и CWnd::OnVScroll. Во избежание такой, достаточно типичной, ситуации в библиотеке классов MFC реализован специальный класс — CScrollView, который берет на себя всю работу по поддержке автоматической прокрутки и масштабирования. При таком подходе на вашу долю остается совсем немного: нужно рассчитать величину прокрутки, базируясь на размере всего документа, и из переопределенной функции CView::OnInitialUpdate или CView::OnUpdate вызвать функцию SetScrollSizes. Ее вызов устанавливает режим отображения для представления, его полный размер и величины горизонтальной и вертикальной прокрутки. Все размеры даются в логических единицах, которые обычно рассчитываются исходя из данных, хранящихся в документе.

Вместо прокрутки класс CScrollView может автоматически установить шкалу представления относительно текущих размеров окна. В этом режиме представление не имеет полос прокрутки и логическое представление сжато или

растянуто точно по размеру рабочей области окна. Для использования этой возможности следует вызвать CScrollView::SetScaleToFitSize.

Примечание

Нельзя использовать функции SetScaleToFitSize и SetScrollSizes одновременно.

Полоса прокрутки в представлении появляется в трех случаях:

- Если для представления установлен стандартный стиль Windows, включающий стили WS_HSCROLL и/или WS_VSCROLL
- Когда во фрейм, содержащий представление, добавлен элемент управления "полоса прокрутки"; при этом библиотека посылает сообщения WM_HSCROLL и/или WM_VSCROLL из фрейма в текущее активное представление
- Когда объект CScrollView расположен в CSplitterWnd с раздельными полосами прокрутки; библиотека посылает сообщения WM_HSCROLL и/или WM_VSCROLL из CSplitterWnd в текущую активную разделенную область

Рассмотрим основные члены этого класса.

CScrollView::**CScrollView**( ) —

создает объект класса. До использования этого объекта необходимо вызвать одну из функций — SetScrollSizes или SetScaleToFitSize.

Для работы в режиме прокрутки необходимо, прежде всего, установить размеры представления:

void CScrollView::**SetScrollSizes**(
int nMapMode,
SIZE sizeTotal,
const SIZE &sizePage = sizeDefault,
const SIZE &sizeLine = sizeDefault) —

служит для регулировки характеристик прокрутки текущего представления. Параметр nMapMode задает текущий режим отображения, и может принимать одно из следующих значений: MM_TEXT, MM_HIMETRIC, MM_TWIPS, MM_HIENGLISH, MM_LOMETRIC и MM_LOENGLISH. Общие размеры представления определены в sizeTotal. Они могут быть как фиксированными, так и зависеть от размера ассоциированного документа. Величина вертикальной и горизонтальной прокрутки "на страницу" задается параметром sizePage, а "на строку" — параметром sizeLine. Обычно функция вызывается из функции OnUpdate или OnInitialUpdate.

```
void CNoteView::OnUpdate(CView*, LPARAM, CObject *pHint)
{
  // Если определено представление, требующее перерисовки,
  if(pHint != NULL)
    // и оно является объектом класса CNoteView ...
    if(pHint->IsKindOf(RUNTIME_CLASS(CNoteView)))    {
```

```
        // ... изменяем размеры этого представления
        CNDApp *pApp = (CNDApp*)AfxGetApp();

        // Параметры размера бумаги хранятся у приложения
        // и надо получить доступ к ним
        CSize sizeTotal;

        // Формируем размеры области вывода, учитывая
        // как размер бумаги, так и поля
        sizeTotal.cx = (pApp->m_sizePaper.cx   -
                        pApp->m_rtMargin.right -
                        pApp->m_rtMargin.left - 120);
        sizeTotal.cy = (pApp->m_sizePaper.cy -
                        pApp->m_rtMargin.bottom -
                        pApp->m_rtMargin.top - 120);
        // В соответствии с текущими размерами бумаги и полей
        // устанавливаем новые параметры прокрутки
        SetScrollSizes(MM_LOMETRIC, sizeTotal);
      }
    // Обновляем все, хотя можно и ограничить размеры области
    Invalidate(TRUE);
  }
```

Установленные значения размеров области прокрутки можно в любой момент запросить, используя функции:

CSize CScrollView::**GetTotalSize**( ) —

позволяет узнать полные текущие горизонтальный и вертикальный размеры (в логических единицах) прокручиваемого представления.

и

void CScrollView::**GetDeviceScrollSizes**(
int &nMapMode,
SIZE &sizeTotal,
SIZE &sizePage,
SIZE &sizeLine) —

возвращает текущие значения режима отображения (параметр nMapMode), полного размера (параметр sizeTotal), а также величины прокрутки на страницу (параметр sizePage) и на строку (параметр sizeLine). Все размеры даются в единицах устройства, и для их перевода в логические, вне функций OnDraw и OnPrint, следует создать контекст устройства

```
CClientDC dc(this);
```

и преобразовать единицы

```
dc.DPtoLP(sizeTotal);
```

Кроме представленных, в классе реализованы знакомые по работе с элементом управления SCROLL BAR функции получения и установки позиции ползунка полосы прокрутки.

CPoint CScrollView::**GetDeviceScrollPosition**( )

и

CPoint CScrollView::**GetScrollPosition**( ) —

вызываются, когда необходимо узнать горизонтальную и вертикальную позиции ползунка в зоне прокрутки. Возвращаемая пара координат соответствует размещению в документе, в котором был перемещен левый верхний угол представления (рис. 20.1). Они полезны для преобразования позиции указателя мыши в позицию прокручиваемого представления. Первая функция возвращает значения в единицах устройства, а вторая — в логических единицах.

void CScrollView::**ScrollToPosition**(POINT pt) —

устанавливает прокрутку в заданную позицию представления. При этом поле pt.sx должно быть больше, или равно нулю, но меньше полного размера представления. Это же справедливо для поля pt.sy, если режим отображения установлен в MM_TEXT. В противном случае эта величина должна быть отрицательной. Если работа с представлением осуществляется в режиме масштабирования, то функцию вызывать нельзя.

Для перехода в режим масштабирования достаточно вызвать функцию:

void CScrollView::**SetScaleToFitSize**(SIZE sizeTotal) —

осуществляет автоматическое масштабирование области вывода представления относительно размера текущего окна. При использовании прокрутки только часть логического представления может быть видима в текущий момент времени. При использовании масштабирования представление не имеет полос прокрутки, и его логический размер изменяется так, чтобы точно совпасть со всей рабочей областью окна. Если окно изменяет размер, то промасштабированное представление отображает свои данные в новой области (рис. 20.7). Наиболее часто она используется для реализации операции "Zoom to fit".

Рис. 20.7. Разделенное окно представления в режиме масштабирования

Рис. 20.8. Фрейм документа до выполнения функции ResizeParentToFit

Рис. 20.9. Фрейм документа после выполнения функции ResizeParentToFit(FALSE)

Рис. 20.10. Фрейм документа после выполнения функции ResizeParentToFit(TRUE)

Обе эти функции (SetScrollSizes и SetScaleToFitSize) рекомендуется вызывать в обработчике OnInitialUpdate. Очевидно, что перед их использованием должны быть установлены размеры самого окна представления, например, следующим образом:

```
GetParentFrame()->RecalcLayout();
```

Для согласования размеров фрейма документа и представления применяется функция:

void CScrollView::**ResizeParentToFit**(BOOL bShrinkOnly = TRUE) —

приводит размеры родительского фрейма в соответствие с размерами представления. Характер изменения размера определяется параметром bShrinkOnly. Если он имеет значение TRUE, то родительский фрейм сжимается до размеров представления; в противном случае представление подгоняется точно под размер фрейма, что может иметь неожиданные последствия в случае с фреймом MDI-приложения или с экраном.

Класс CSplitterWnd

Прокрутка позволяет работать с документом, размер которого больше, чем можно отобразить в окне, но при этом большая часть документа скрыта от пользователя. Вторым вариантом, который мы также рассматривали, является масштабирование. Но если размеры документа достаточно велики, то этот способ хорош только для получения общей информации о его структуре, т. к. при этом на экране трудно рассмотреть его содержимое. Для того чтобы пользователь мог одновременно просматривать различные части документа, в Windows используется так называемое "разделенное окно", которое представляет собой одно окно, разбитое на несколько областей. Такое окно содержит "блок разделителя" (splitter box) в верхней части вертикальной и левой части горизонтальной полос прокрутки, двойной щелчок на котором приводит к разделению области, соответственно, на вертикальные или горизонтальные внутренние области. Сами области отделены друг от друга специальной *полосой разделения* (splitter bar), которая служит также для перераспределения площади окна между соседними областями. *Пересечение горизонтальной и вертикальной полос прокрутки* (splitter intersection) позволяет одновременно регулировать и вертикальный и горизонтальный размеры областей.

Для поддержки разделенных окон в библиотеке классов MFC реализован специальный класс CSplitterWnd. Каждая область представляет собой отдельное окно, которое управляется объектом CSplitterWnd и обычно является определенным объектом приложения, производным от CView (или его производного класса), но может быть и произвольным объектом CWnd, который имеет соответствующий идентификатор дочернего окна. При этом класс, на основе которого создается область, должен использовать макросы DECLARE_DYNCREATE и IMPLEVENT_DYNCREATE для поддержки режима динамического создания. Очевидно, что для областей гораздо легче (и лучше) использовать классы представлений, чем "обычные" оконные классы. В зависимости от того, каким образом — по горизонтали или по

вертикали, разбито окно, для соответствующих областей используются названия "строка" и "столбец". Исходное окно при этом идентифицируется как "нулевая строка" и "нулевой столбец".

Класс CSplitterWnd поддерживает два различных стиля разделенных окон: статические и динамические.

В статических разделенных окнах (рис. 20.11) области создаются только один раз, в момент создания самого окна. Их число и порядок расположения не могут быть изменены в процессе работы. Полосы разделителей используются для изменения размеров различных областей, которые обычно имеют различные классы представлений.

Рис. 20.11. Статическое разделенное окно с двумя горизонтальными областями

Рис. 20.12. Динамическое разделенное окно с двумя горизонтальными и двумя вертикальными областями

В динамических разделенных окнах (рис. 20.12) дополнительные области могут создаваться и разрушаться в произвольные моменты времени. Эти окна создаются с единственным представлением, и полосы разделителей предоставляют начальное разделение. Если представление разделено в одном направлении, то дополнительный объект представления динамически создается в представленной новой области. Если представление разделяется в двух направлениях, то три новых представления создаются в трех новых областях. Когда разделитель активен, блок разделителя изображается в виде разделительной полосы между областями. Дополнительные объекты представления разрушаются, когда пользователь удаляет разделитель, но начальное представление (нулевая строка, нулевой столбец) остается, пока не будет разрушено его окно.

При создании любого из этих разделенных окон необходимо сначала определить максимальное число строк и столбцов, которыми разделитель будет управлять. Для статического разделенного окна области должны быть созданы и все заполнены строками и столбцами. Для динамического разделяемого окна первая область создается автоматически при создании объекта CSplitterWnd.

В обоих случаях число областей, на которые можно разделять окно, ограничено: для статических разделяемых окон — 16 строк и 16 столбцов; для динамических — 2 строки и 2 столбца. Одна из многочисленных рекомендаций фирмы Microsoft касается и конфигураций разделенных окон:

Строка/Столбец	Рекомендации фирмы Microsoft по использованию
Для статических окон	
1 строка × 1 столбец	Обычно областями, основанными на различных объектах
2 строки × 1 столбец	
2 строки × 2 столбца	Обычно областями, основанными на одинаковых объектах
Для динамических окон	
1 строка × 2 столбца	Для данных, расположенных в колонках
2 строки × 1 столбец	Для текстовых или других данных
2 строки × 2 столбца	Для сеточных номограмм и табличных данных

Для каждой строки, как и для каждого столбца, существуют минимальные размеры, и если какая-либо область имеет размеры меньше минимальных, то соответствующая область не отображается на экране. Для статических разделяемых окон эти значения равны 0, а для динамических определяются соответствующим параметром, задаваемым при создании окна этой области. Эти минимальные размеры в дальнейшем могут быть изменены.

Как и многие другие окна Windows, окно класса CSplitterWnd может иметь полосы прокрутки, которые определяются либо указанием соответствующего стиля окна при его создании — WS_HSCROLL и/или WS_VSCROLL, либо могут представлять собой дочерний элемент управления. В этом плане они ничем не отличаются от других типов окон. Но наряду с этим, класс CSplitterWnd поддерживает специальный тип полос прокрутки, которые могут использоваться совместно различными областями. В этом случае, когда пользователь перемещает ползунок полосы прокрутки, соответствующее сообщение посылается во все области.

Полосы прокрутки создаются при создании разделяемого окна. Например, объект класса CSplitterWnd, который имеет одну строку, два столбца и стиль WS_VSCROLL, изображает вертикальную полосу прокрутки, которая используется совместно двумя областями, и сообщение WM_VSCROLL посылается в обе.

Теперь, после небольшого экскурса по разделяемым окнам, перейдем к рассмотрению тех возможностей, которые можно использовать для реализации перечисленных свойств.

Обычно объект класса CSplitterWnd внедряется в свой родительский объект класса CFrameWnd для SDI- и CMDIChildWnd для MDI-приложений. В этом случае вызывается конструктор класса:

CSplitterWnd::**CSplitterWnd**( ) —

создает объект соответствующего класса.

Поскольку, как мы выяснили, окно этого объекта тесно связано с рабочей областью родительского окна, прежде всего, необходимо переопределить функцию родительского фрейма CFrameWnd::OnCreateClient, из которой вызвать одну из двух функций, отвечающих за создание окна Windows и присоединение его к соответствующему объекту.

BOOL CSplitterWnd::**Create**(
CWnd *pParentWnd,
int nMaxRows,
int nMaxCols,
SIZE sizeMin,
CCreateContext *pContext,
DWORD dwStyle = WS_CHILD | WS_VISIBLE | WS_VSCROLL |
WS_HSCROLL | SPLS_DYNAMIC_SPLIT,
UINT nID = AFX_IDW_PANE_FIRST) —

создает динамическое разделяемое окно, которое обычно используется для создания и прокрутки определенного числа индивидуальных областей (или представлений) того же самого документа. Все области, используемые в таком окне, должны быть одного класса. При успешном создании окна Windows функция возвращает TRUE, в противном слу-

чае — FALSE. В качестве параметров она использует: pParentWnd — указатель на родительский фрейм; nMaxRows и nMaxCols — максимальное число строк и столбцов, которое может иметь разделенное окно (оба эти параметра не могут быть больше 2); sizeMin — минимальная высота строки или ширина столбца, при котором область будет изображаться на экране; pContext — указатель на структуру CCreateContext, который в большинстве случаев является соответствующим параметром родительского фрейма или может быть равен NULL; nID — идентификатор дочернего окна, в качестве которого может использоваться значение, указанное по умолчанию, если это окно не используется внутри другого разделяемого окна; dwStyle — стиль окна, в дополнение к стандартным оконным стилям, используется SPLS_DYNAMIC_SPLIT, наличие которого указывает на то, что создается динамическое разделяемое окно.

```
BOOL CDrawFrame::OnCreateClient(LPCREATESTRUCT lpcs,
                                CCreateContext* pContext)
{
// Создаем динамическое разделяемое окно с двумя строками
// и двумя столбцами, устанавливая некоторые начальные
// минимальные размеры, чтобы сначала представление занимало
// все разделяемое окно
   return m_splitWnd.Create(this, 2, 2, CSize(10, 10), pContext);
}
```

BOOL CSplitterWnd::**CreateStatic**(
CWnd *pParentWnd,
int nRows, int nCols,
DWORD dwStyle = WS_CHILD | WS_VISIBLE,
UINT nID = AFX_IDW_PANE_FIRST) —

создает статическое разделяемое окно, в котором пользователь может изменять только размер индивидуальных областей, но не их число или порядок. Количество областей задается параметрами nRows и nCols, значения которых не должны быть больше 16. Функция автоматически устанавливает минимальные высоту строки и ширину столбца в нулевое значение. Для добавления в это окно полос прокрутки необходимо дополнительно установить стили WS_VSCROLL и WS_HSCROLL к значению по умолчанию параметра dwStyle. При успешном создании окна Windows функция возвращает TRUE, в противном случае — FALSE.

В отличие от использования функции Create, при создании статического разделяемого окна необходимо самостоятельно позаботиться о создании всех его областей, которые могут принадлежать различным оконным классам. Обычно для этого используется функция CreateView, но можно создавать любой оконный класс, не обязательно основанный на представлении. Это следует реализовать до завершения функции OnCreateClient родительского фрейма для корректного изображения окна.

BOOL CSplitterWnd::**CreateView**(
int row, int col,
CRunTimeClass *pViewClass,

SIZE sizeInit,
CCreateContext *pContext) —

создает область статического разделяемого окна. Строка и столбец, в которых будет размещаться вновь создаваемая область, задаются в параметрах row и col. Тип области передается в функцию через pViewClass — указатель на структуру CRunTimeClass, определяющую класс представления. Параметр sizeInit устанавливает начальный размер нового представления.

```
BOOL CNoteFrame::OnCreateClient(LPCREATESTRUCT lpcs,
                                CCreateContext* pContext)
{
// Создаем разделенное окно Windows
// с двумя строками и одним столбцом
  if(!m_spltWnd.CreateStatic(this, 2, 1))
  {
    TRACE0("Ошибка при создании разделенного окна\n");
    return FALSE;
  }
  // Создаем представление в первой строке (горизонтальной области)
  if(!m_spltWnd.CreateView(0, 0, pContext->m_pNewViewClass,
      CSize(lpcs->cx, lpcs->cy*2/3), pContext))
  {
    TRACE0("Ошибка при создании представления для вывода\n");
    return FALSE;
  }
  // Создаем представление во второй строке (горизонтальной
области)
  if(!m_spltWnd.CreateView(1, 0, RUNTIME_CLASS(CTextView),
      CSize(0, 0), pContext))
  {
    TRACE0("Ошибка при создании представления для ввода\n");
    return FALSE;
  }
  // Устанавливаем фокус на область (1, 0)
  SetActiveView((CView*)m_spltWnd.GetPane(1, 0));

  return TRUE;
}
```

Еще раз отметим: для динамического разделяемого окна вызов функции CreateView осуществляется автоматически, а при создании всех областей статического разделяемого окна эту функцию необходимо вызывать явно.

Рассмотрев процесс создания, перечислим функции, которые позволяют получать информацию и изменять параметры разделяемого окна.

int CSplitterWnd::**GetRowCount**( )

и

int CSplitterWnd::**GetColumnCount**( ) —

позволяют узнать, соответственно, текущее число строк или столбцов в разделяемом окне. Для статического окна это одновременно и максимальное число.

void CSplitterWnd::**GetRowInfo**(
int row,
int &cyCur,
int cyMin)

и

void CSplitterWnd::**GetColumnInfo**(
int col,
int &cxCur,
int cxMin) —

позволяют получить информацию о строке (столбце), определяемой параметром row (col). Через параметр cyCur (cxCur) возвращается информация о текущей, а через cyMin (cxMin) — о минимальной высоте строки (ширине столбца). Все размеры даются в пикселях.

void CSplitterWnd::**SetRowInfo**(
int row,
int &cyIdeal,
int cyMin)

и

void CSplitterWnd::**SetColumnInfo**(
int col,
int &cxIdeal,
int cxMin) —

позволяют установить новые значения минимальной (параметры cyMin, cxMin) и идеальной (параметры cyIdeal, cxIdeal) высоты строки (ширины столбца), заданной в параметре row (col). Когда библиотека MFC изображает разделяемое окно, она располагает области в строках и столбцах в соответствии с их идеальными размерами, начиная от левого верхнего до правого нижнего угла его рабочей области.

int CSplitterWnd::**IdFromEowCol**(
int row, int col) —

позволяет получить идентификатор дочернего окна для области, заданной своими строкой row и столбцом col. Эта функция используется в случае, если при создании соответствующей области применялось "обычное", а не производное от CView, окно.

void CSplitterWnd::**RecalcLayout**( ) —

обновляет изображение разделяемого окна после того, как были изменены размеры строк(и) и/или столбцов(а). Функцию следует применять в случае, если окно уже видимо. Библиотека MFC вызывает ее, когда пользователь изменил размер окна или его области.

DWORD CSplitterWnd::**GetScrollStyle**( ) —

возвращает текущий стиль совместно используемых полос прокрутки для разделяемых окон. Возможны следующие значения: WS_HSCROLL и WS_VSCROLL. Если окно не имеет совместно используемых полос прокрутки, то возвращается нуль.

void CSplitterWnd::**SetScrollStyle**(DWORD dwStyle) —

устанавливает новый стиль полосы прокрутки для разделяемого окна, которое поддерживает совместно используемые полосы прокрутки. Если функция вызывается без установленного стиля, то полоса прокрутки убирается с экрана, но не удаляется. После выполнения этой функции необходимо вызвать функцию RecalcLayout для перерисовки разделяемого окна.

Следующая группа функций может быть переопределена в производных классах для реализации индивидуальных возможностей.

virtual void CSplitterWnd::**OnDrawSplitter**(
CDC *pDC,
ESplitType nType,
const CRect &rect) —

вызывается библиотекой MFC для отображения и установки точных характеристик разделяемого окна. Параметр pDC задает контекст устройства, и если он равен NULL, то вызывается функция CWnd::RedrawWindow, и рисование производится для неразделяемого окна. Область, которая перерисовывается, определяется параметром rect. Параметр nType может принимать одно из следующих значений:

splitBox	Блок разделителя
splitBar	Полоса, которая изображается между двумя разделенными областями
splitIntersection	Точка пересечения разделенных областей; не используется для Windows 95/98
splitBorder	Границы разделенного окна

Если образ элемента по каким-либо причинам вас не устраивает, то для изображения своего варианта необходимо переопределить эту функцию.

virtual void CSplitterWnd::**DeleteView**(
int row,
int col) —

вызывается библиотекой MFC, когда представление было удалено из динамического разделяемого окна. Если удаленное представление было активно, то активным становится следующее представление. Параметры row и col определяют область (по строке и столбцу), в которой удаляется представление.

virtual BOOL CSplitterWnd::**SplitRow**(int cyBefore)

и

virtual BOOL CSplitterWnd::**SplitColumn**(int cxBefore) —

вызываются библиотекой MFC при создании горизонтальной (вертикальной) области в динамическом разделяемом окне. Параметр cyBefore (cxBefore) определяет позицию (в пикселях), в которой производится разделение.

virtual void CSplitterWnd::**DeleteRow**(int rowDelete)

и

virtual void CSplitterWnd::**DeleteColumn**(int colDelete) —

вызываются библиотекой MFC, когда из динамического разделяемого окна *была* удалена строка rowDelete (столбец colDelete).

virtual CWnd* CSplitterWnd::**GetActivePane**(
int *pRow = NULL,
int *pCol = NULL) —

используется для определения параметров (номера строки — pRow и столбца — pCol) активной области в разделяемом окне. Если таковой нет, функция возвращает NULL.

virtual void CSplitterWnd::**SetActivePane**(
int row,
int col,
CWnd *pWnd = NULL) —

устанавливает фокус на соответствующую область внутри фрейма. Активизируемая область определяется либо номером строки row и столбца col, если указатель на оконный объект pWnd равен NULL, либо непосредственно самим указателем. Эта функция вызывается библиотекой MFC, когда пользователь изменил фокус активной области, но может быть вызвана и непосредственно для принудительной смены фокуса.

virtual BOOL CSplitterWnd::**CanActivateNext**(BOOL bPrev = FALSE) —

проверяет возможность выполнения команд **Next Pane** (Следующее окно) или **Previous Pane** (Предыдущее окно). Команда определяется параметром bPrev: если его значение TRUE, то проверяется предшествующая, если FALSE — следующая область.

virtual void CSplitterWnd::**ActivateNext**(BOOL bPrev = FALSE) —

активизирует следующую (bPrev = FALSE) или предшествующую (bPrev =TRUE) область разделенного окна.

virtual BOOL CSplitterWnd::**DoScroll**(
CView *pViewFrom,
UINT nScrollCode,
BOOL bDoScroll = TRUE) —

вызывается библиотекой MFC для выполнения синхронизации прокрутки разделенных окон, когда представление получает сообщение о прокрутке. Параметр pView определяет указатель на объект представления, откуда поступило сообщение. Надо ли действительно осуществлять прокрутку, определяется в параметре bDoScroll. Если он равен TRUE, то прокрутку необходимо выполнять, если FALSE, то прокрутка не производится. Последнее имеет место, если, например, соответствующее представление не имеет полосы прокрутки. Параметр nScrollCode определяет код полосы прокрутки и состоит из двух частей: младший байт задает тип прокрутки по горизонтали, а старший — по вертикали.

virtual BOOL CSplitterWnd::**CreateScrollBarCtrl**(
DWORD dwStyle,
UINT nID) —

создает разделяемый элемент управления "полоса прокрутки". Параметр dwStyle определяет его оконный стиль, а nID — идентификатор дочернего окна, куда этот элемент управления будет вставлен; он может иметь значение AFX_IDW_PANE_FIRST, если заданное разделенное окно не вложено в другое разделенное окно.

На этом мы заканчиваем общее рассмотрение классов документов и представлений и переходим к достаточно интересной, и тесно связанной с рассматриваемой, теме — печать и предварительный просмотр документа перед печатью.

Глава 21

Печать и предварительный просмотр документов

Для взаимодействия приложений с графическими устройствами, такими как экран дисплея, принтер или плоттер, в Windows используется интерфейс графических устройств (GDI). И когда приложению требуется выполнить операцию вывода графического изображения, оно осуществляет это посредством GDI и работает не с реальным (физическим), а с логическим устройством. Выполняя запрос приложения, GDI обращается к драйверу соответствующего устройства вывода, который уже непосредственно работает с физическим устройством. Такой подход позволяет приложениям Windows единообразно работать на любом оборудовании, лишь бы был установлен соответствующий драйвер. Само приложение обращается к устройству вывода посредством контекста устройства, который представляет собой структуру данных, содержащую информацию о том, как нужно выполнять операции вывода на данном устройстве. Эта структура при ее создании ассоциируется с драйвером конкретного физического устройства, преобразующего все направляемые ему вызовы в такие, которые им поддерживаются.

Все сказанное в полной мере относится и к выводу на принтер: прежде всего, необходимо создать контекст отображения, связанный с принтером, куда затем и направлять весь графический вывод (как и при выводе на экран). Однако отличия все-таки есть. И связаны они, прежде всего, с тем, что при печати графический вывод осуществляется постранично, а листы бумаги могут иметь различные размеры и/или ориентацию (книжную или альбомную). За отслеживание этих параметров целиком отвечает программист. Есть и некоторые другие отличия. Но от слов пора переходить к делу.

Следует совершенно четко представлять, что вывод на принтер осуществляется исключительно через буфер, который, однако, не просто служит для временного хранения данных. Все несколько сложнее. Дело в том, что при вызове функций рисования для контекста принтера, эти команды GDI выполняются не сразу, а накапливаются в специальном метафайле. И только после того, как приложение завершит рисование одной страницы докумен-

та, созданный метафайл "проигрывается" в контексте принтера. Печать происходит именно в этот момент. Сам механизм скрыт от приложения, и единственное, что требуется, — это сообщить GDI о начале и завершении процесса печати листа.

Вторым важным моментом является то, что печать в системе Windows организована через специальную очередь, которая организуется приложением Print Manager (Диспетчер печати). Для печати документа приложения помещают в эту очередь свои данные (задания на печать), которые будут выводиться на принтер в фоновом режиме, в порядке поступления.

Для печати документов Win32 API предоставляет следующие функции:

- StartDoc — формирует задание на печать нового документа
- StartPage — подготавливает контекст устройства вывода для печати новой страницы — готовит метафайл; необходимо вызвать эту функцию перед выводом в контекст устройства
- EndPage — завершает программный процесс печати одной страницы — формирование метафайла, после чего он выводится непосредственно на принтер
- EndDoc — завершает процесс печати документа
- AbortDoc — служит для принудительного завершения процесса печати
- SetAbortProc — используется для обеспечения возможности фоновой печати и принудительного завершения процесса печати
- ResetDC — позволяет настроить индивидуальные параметры печати отдельных листов документа

Как видите, функций не много. А это значит, что всю основную работу необходимо выполнить самостоятельно. Сам же процесс графического вывода на принтер достаточно прост:

1. Прежде всего, формируется контекст отображения для принтера.
2. Затем необходимо обеспечить возможность принудительного (аварийного) завершения печати, заполнить структуру DOCINFO и вызвать функцию StartDoc.
3. После завершения подготовительных действий организуется цикл печати страниц документа, где должны быть проверены специально созданные флаги отмены и завершения печати. Вывод на принтер в Windows организуется постранично, поэтому перед началом печати каждой страницы необходимо вызвать функцию StartPage, которая устанавливает параметры специального метафайла, куда будет направляться вывод для всей страницы.
4. После того как вся страница записана в метафайл, следует закрыть его и отправить непосредственно на принтер. Для этой цели используется функция EndPage. Цикл выполняется для всех страниц и завершается ли-

бо аварийно, либо после вывода всего документа. В первом случае следует вызвать функцию AbortDoc, а во втором — EndDoc.

5. В заключение необходимо освободить используемый контекст устройства.

Примечание

Пока мы описываем этот процесс в упрощенном виде, не вдаваясь в некоторые детали, например, как определять окончание страницы или как настроить параметры принтера.

Подобный процесс, с теми или иными вариациями, выполняется при любом выводе документа на печать.

После того как сложилось общее представление об организации процесса печати в Windows, вернемся к библиотеке классов MFC и рассмотрим, как в ней реализован вывод данных документа на принтер, и другие связанные с этим вопросы. Начнем с выбора и настройки параметров принтера.

Выбор и настройка параметров принтера

Поскольку вся информация об установленных принтерах хранится в самой операционной системе, то вполне логичным выглядит то, что за рассматриваемый процесс отвечает приложение. В классе CWinApp реализован специальный обработчик стандартной команды ID_FILE_PRINT_SETUP:

afx_msg void CWinApp::**OnFilePrintSetup**( ) —

выводит на экран стандартный блок диалога **Print Setup** (настройка печати) для настройки текущей конфигурации принтера.

Для подключения обработчика к команде, в карту сообщений объекта "приложение" необходимо добавить оператор

```
ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
```

В большинстве случаев вполне достаточно этого стандартного обработчика, но, при необходимости, можно создать и что-нибудь свое. Изобретать велосипед мы не будем, а для иллюстрации рассмотрим лишь, каким образом можно реализовать индивидуальную настройку стандартного блока диалога **Print Setup**, класс для работы с которым был описан в главе 15. Прежде всего, в карту сообщений класса приложения добавляем оператор:

```
ON_COMMAND(<ID_PRINT_SETUP>, <OnPrintSetup>)
```

Примечание

Имена в угловых скобках могут быть произвольными.

Затем в описании класса регистрируем этот обработчик:

```
class CNDApp : public CWinApp
{
...
  //{{AFX_MSG(CNDApp)
  afx_msg void OnPrintSetup();
  //}}AFX_MSG
};
```

И, наконец, реализуем его. В данном примере при инициализации блока диалога устанавливаем размер бумаги **Letter $8^1/_2$ ×11 in** и альбомную ориентацию (рис. 21.1).

```
void CNDApp::OnPrintSetup()
{
  // Создаем объект "стандартный блок диалога
  // Print Setup (Настройка печати)"
  CPrintDialog dlgPD(TRUE);
  // Для получения параметров принтера, установленного
  // по умолчанию, создаем объект
  PRINTDLG pd;
  // Устанавливаем указатель на глобальный блок памяти в NULL
  // с тем, чтобы запросить у системы параметры принтера
  pd.hDevMode = NULL;
  // Позволяем получить текущую конфигурацию принтера
  GetPrinterDeviceDefaults(&pd);
  // Поскольку параметр pd.hDevMode указывает на глобальный
  // перемещаемый блок памяти, перед началом работы с ним
  // необходимо его зафиксировать, иначе ничего не получится
  DEVMODE *dm = (DEVMODE *)LocalLock(pd.hDevMode);
  // Теперь можно заполнять поля структуры DEVMODE значениями,
  // необходимыми для инициализации блока диалога Print Setup
  dm->dmFields = DM_ORIENTATION | DM_PAPERSIZE;
  dm->dmOrientation = DMORIENT_LANDSCAPE;
  dm->dmPaperSize = DMPAPER_LETTER;
  // Передаем указатель на заполненную структуру DEVMODE
  // объекту класса CPrintDialog
  dlgPD.m_pd.hDevMode  = pd.hDevMode;
  // Теперь зафиксированный блок памяти можно освободить
  LocalUnlock(pd.hDevMode);
  // Выводим блок диалога на экран
  if(dlgPD.DoModal() == IDOK)  {
```

```
    // Здесь можно зафиксировать выбор пользователя
    ...
  }
}
```

Рис. 21.1. Стандартный блок диалога **Print Setup**

Возможностей для индивидуальных настроек достаточно много, но, повторяем, библиотека MFC достаточно хорошо выполняет значительную часть работы, и в большинстве случаев нет необходимости вмешиваться в этот процесс.

Для того чтобы узнать текущую конфигурацию принтера, мы воспользовались функцией:

BOOL CWinApp::**GetPrinterDeviceDefaults**(PRINTDLG *pd) —

возвращает информацию либо о принтере, заданном по умолчанию, либо о последней конфигурации принтера, установленной с помощью блока диалога **Print Setup**. Параметр pd указывает на структуру PRINTDLG, которая служит для настройки параметров блоков диалога **Print** (печать) и **Print Setup** и подробно была описана в главе 15:

```
...
PRINTDLG pd;
// Запрашиваем параметры текущего
// или установленного по умолчанию принтера
GetPrinterDeviceDefaults(&pd);
...
```

Эта часть работы, конечно же, достаточно важна, и обеспечить ее поддержку необходимо, но библиотека MFC делает это прекрасно, а нас интересует дальнейший процесс. Итак, принтер выбран, его параметры установлены —

можно печатать. Первоочередной задачей является, естественно, создание контекста устройства для выбранного принтера.

Создание контекста устройства

Библиотека MFC и здесь приходит на помощь — у класса приложения (CWinApp) имеется специальная функция:

BOOL CWinApp::**CreatePrinterDC**(CDC &dc) —

создает контекст устройства (параметр dc) для выбранного в текущий момент принтера.

Перед тем как выводить что-либо на печать, достаточно вызвать эту функцию:

```
...
CDC dc;
// Создаем контекст устройства для принтера
AfxGetApp()->CreatePrinterDC(&dc);
// Теперь можно выводить информацию в контекст устройства dc
```

Вот и все. Не нужно каждый раз выполнять "утомительные" настройки специальных структур, как это приходится делать при использовании Win32 API.

Для того чтобы полностью разобраться в рассматриваемом вопросе, нам осталось только рассмотреть организацию собственно процесса печати.

Печать документов и библиотека MFC

Дальнейшее изложение мы будем вести применительно к архитектуре "документ/представление". Это не случайно. Дело в том, что в классах представлений реализована достаточно мощная поддержка этого процесса, которая избавит вас от многих дополнительных трудностей, освободив время для решения конкретных задач.

В процессе печати основная роль принадлежит объектам двух классов библиотеки MFC — контексту устройства и представлению, которые тесно взаимодействуют между собой. На рис. 21.2 представлена схема цикла печати, реализованная в библиотеке классов MFC.

Как видите, представленный цикл очень напоминает используемый в Windows для любого вывода на печать: на экран выводится блок диалога **Print**, создается контекст устройства для принтера и вызывается функция StartDoc объекта CDC. Далее для каждой страницы документа библиотека вызывает функцию StartPage контекста принтера, осуществляет вывод данных в метафайл (функция OnPrint объекта "представление"), и завершает печать страницы вызовом функции EndPage объекта CDC. Когда весь документ выведен на принтер, библиотека вызывает функцию EndDoc, завершая

цикл печати. Разница заключается только в том, что как сам цикл, так и все участвующие здесь функции, уже реализованы. Наряду с этим, благодаря механизму виртуальных функций программисту предоставляются широкие возможности для "более тонкой" настройки всего процесса печати. Все эти функции (OnPreparePrinting, OnBeginPrinting, OnPrepareDC, OnPrint и OnEndPrinting) являются виртуальными, определены в классе CView и при их переопределении необходимо соблюдать логику, которая заложена в библиотеку классов MFC. Но сначала рассмотрим сами эти функции:

Рис. 21.2. Цикл печати библиотеки MFC

virtual BOOL CView::**OnPreparePrinting**(CPrintInfo *pInfo) —

вызывается библиотекой классов MFC до начала процесса печати или предварительного просмотра документа. Это самый удобный момент для определения длины документа и настройки параметров печати. Для реализации последней возможности необходимо просто вызвать функцию DoPreparePrinting:

```
BOOL CView::OnPreparePrinting(CPrintInfo *pInfo)
{
  return DoPreparePrinting(pInfo);
}
```

В качестве параметра функция принимает указатель на структуру CPrintInfo, которая хранит информацию о задачах печати или предварительного просмотра:

```
struct CPrintInfo
{
  CPrintInfo();              // Конструктор
  ~CPrintInfo();             // Деструктор
  // Указатель на объект "блок диалога" Print
  CPrintDialog *m_pPD;
  BOOL m_bPreview;           // TRUE, если предварительный просмотр
  BOOL m_bDirect;            // TRUE, если без отображения на экране
                             // блока диалога Print
  BOOL m_bContinuePrinting;  // FALSE для прерывания печати
  UINT m_nCurPage;           // Номер текущей печатаемой страницы
  UINT m_bNumPreviewPages;   // Число просматриваемых страниц
  CString m_strPageDesc;     // Номер изображаемой страницы
  LPVOID m_lpUserData;       // Указатель на данные пользователя
  CRect m_rectDraw;          // Прямоугольник, определяющий текущую
                             // используемую область страницы
  void SetMinPage(UINT nMinPage); // Устанавливает номер первой
                                  // печатаемой страницы документа
  void SetMaxPage(UINT nMaxPage); // Устанавливает номер последней
                                  // печатаемой страницы документа
  UINT GetMinPage() const;        // Получает номер первой
                                  // печатаемой страницы документа
  UINT GetMaxPage() const;        // Получает номер последней
                                  // печатаемой страницы документа
  UINT GetFromPage() const;       // Получает номер первой
                                  // печатаемой страницы диапазона
  UINT GetToPage() const;         // Получает номер последней
                                  // печатаемой страницы диапазона
};
```

Объект этой структуры используется для обмена информацией между библиотекой и классом представления во время процесса печати. Он создается при выборе команд ID_FILE_PRINT_PREVIEW, ID_FILE_PRINT_DIRECT или ID_FILE_PRINT и разрушается после их выполнения.

Если возникнет необходимость инициализации блока диалога **Print** значениями, отличными от заданных по умолчанию, то лучше всего это сделать именно здесь (до вызова функции DoPreparePrinting).

BOOL CView::**DoPreparePrinting**(CPrintInfo *pInfo) —

возвращает TRUE, если можно начинать печать или предварительный просмотр документа, и FALSE в противном случае. Поведение этой функции зависит от того, вызывается ли она для печати документа или для его предварительного просмотра, что определяется параметром m_bPreview структуры, на которую указывает pInfo. В первом случае на экран выводится блок диалога **Print**. После его закрытия функция создает контекст устройства на основе установок, сделанных пользователем в этом блоке диалога для принтера, который будет использоваться при печати документа. Если функция вызвана для предварительного просмотра документа, то контекст устройства создается на основе текущих установок принтера.

Для осуществления всех корректных установок в переопределенной версии этой функции обязательно следует вызвать функцию базового класса CView::DoPreparePrinting.

В приведенном ниже фрагменте кода демонстрируется принцип инициализации блока диалога **Print** в зависимости от того, имеется ли во введенном тексте выделенный фрагмент (рис. 21.3).

```
BOOL CNoteView::OnPreparePrinting(CPrintInfo* pInfo)
{
  if (pInfo != NULL && !pInfo->m_bPreview)  {
    // Документ не будет иметь больше одной страницы
    pInfo->SetMaxPage(1);
    // Получаем указатель на документ, с которым
    // ассоциировано текущее представление
    CNoteDoc* pDoc = (CNoteDoc*)GetDocument();
    // Получаем позицию первого представления, хранящегося
    // в списке представлений, ассоциированных с документом
    POSITION pos = pDoc->GetFirstViewPosition();
    // Получаем указатель на первое представление в списке
    pDoc->GetNextView(pos);
    // Получаем указатель на второе представление в списке
    CTextView *pText = (CTextView *)pDoc->GetNextView(pos);
    CString str;
    // Читаем данные представления
    pText->GetSelectedText(str);
    if(!str.IsEmpty())    {
      // Если есть выделенный фрагмент, то переключатель
      // Print range (Печатать) устанавливаем в положение
      // Selection (Выделенный фрагмент)
      CPrintDialog *pPD = pInfo->m_pPD;
      pPD->m_pd.Flags &= ~PD_NOSELECTION;
      pPD->m_pd.Flags |= PD_SELECTION;
    }
```

```
  }
  // Если же выделенного фрагмента нет, то используем
  // установки по умолчанию
...
  // Не забываем вызвать функцию базового класса
  // для проведения корректной инициализации
  return DoPreparePrinting(pInfo);
}
```

Рис. 21.3. Стандартный блок диалога **Print**

Если необходим предварительный доступ к контексту устройства, то это можно сделать, переопределив функцию:

virtual void CView::**OnBeginPrinting**(
CDC *pDC,
CPrintInfo *pInfo) —

вызывается библиотекой классов MFC перед началом процесса печати или предварительного просмотра после того, как была вызвана функция OnPreparePrinting.

Эта функция используется для размещения необходимых для печати ресурсов GDI, таких как карандаши или шрифты. Сами объекты GDI выбираются в контекст устройства в функции OnPrint отдельно для каждой страницы, которая их использует. Если одно и то же представление используется для вывода на экран и на принтер, рекомендуется использовать различные переменные для ресурсов GDI, необходимых для каждого изображения, что позволит обновлять их на экране непосредственно во время процесса печати. Кроме того, эту функцию можно использовать для инициализации, зависящей от свойств контекста устройства принтера, например, от числа пе-

чатаемых страниц. В качестве параметров используются указатели на контекст устройства (параметр pDC) и на структуру pPrintInfo, которая описывает текущее задание печати (параметр pInfo).

Для корректного завершения процесса печати или предварительного просмотра документа необходимо использовать функцию:

virtual void CView::**OnEndPrinting**(
CDC *pDC, CPrintInfo *pInfo) —

вызывается библиотекой классов MFC после окончания процесса печати или предварительного просмотра. Если в функции OnBeginPrinting были задействованы какие-либо ресурсы GDI, то здесь их необходимо вернуть системе.

Если две последние рассмотренные функции используются для документа в целом, то для настройки параметров отдельной страницы разработчики реализовали функцию:

virtual void CView::**OnPrepareDC**(
CDC *pDC, CPrintInfo *pInfo = NULL) —

вызывается библиотекой классов MFC до функций OnDraw (предназначенной для изображения на экране) или OnPrint (используемой для каждой печатаемой или просматриваемой страницы). Если функция вызвана для печати документа, то она проверяет информацию о странице, имеющуюся в структуре, на которую указывает параметр pInfo. Если длина документа не определена, она принудительно прерывает печать после того, как будет выведена одна страница. Остановить процесс печати можно принудительно — записав в поле m_bContinuePrinting структуры pInfo значение FALSE. Параметры: pDC — указатель на контекст устройства, используемый для представления образа документа и pInfo — указатель на структуру CPrintInfo, описывающую текущее задание. В режиме печати или предварительного просмотра поле m_nCurPage структуры CPrintInfo определяет номер печатаемой страницы (если функция вызвана для вывода на экран, этот параметр должен быть установлен в NULL). При переопределении этой функции, прежде всего, следует вызвать обработчик OnPrepareDC базового класса. Переопределение этой функции необходимо в следующих случаях:

- Настройка атрибутов контекста устройства, необходимых для определенной страницы
- Динамическое определение числа страниц документа непосредственно во время печати (например, при распечатке неизвестного заранее числа записей базы данных)
- Посылка на принтер ESC-кодов с использованием функции Escape объекта класса CDC, на который указывает параметр pDC

Теперь, когда общая структура процесса печати достаточно ясна, самое время рассмотреть, где же осуществляется собственно вывод (или, если угодно, рисование) данных документа. В предыдущей главе рассматривалась функция OnDraw, которая в рамках архитектуры "документ/представление" отвечает за вывод на любое устройство, будь то экран, принтер или что-либо другое. Его конкретный тип задается объектом контекста устройства, передаваемого функции в качестве параметра. Например, для вывода на экран она получает в качестве параметра указатель на объект класса CPaintDC, а

для печати документа — на объект класса CDC, ассоциированного с текущим принтером.

Все это, конечно, справедливо, и можно было бы просто переопределить функцию OnDraw для организации вывода на принтер, но ведь печать осуществляется на листы бумаги, имеющие вполне определенные размеры, которые, по меньшей мере, необходимо учитывать. И для того чтобы не перегружать функцию OnDraw, разработчики библиотеки для работы с принтером реализовали в базовом классе всех представлений CView отдельную функцию:

virtual void CView::**OnPrint**(
CDC *pDC, CPrintInfo *pInfo = NULL) —

вызывается библиотекой классов MFC после вызова функции OnPrepareDC для печати или предварительного просмотра каждой страницы документа. Реализация по умолчанию просто вызывает функцию OnDraw с контекстом устройства принтера.

Переопределение этой функции необходимо, например, в следующих случаях:

- При печати многостраничных документов
- Для установки различных образов при выводе на принтер и экран, что допустимо, если приложение не поддерживает режима WYSIWYG
- Для реализации изображения колонтитулов и примечаний

Здесь есть два тонких момента, требующих пояснения. Пока речь идет о печати одной страницы, то все в порядке — можно использовать реализацию по умолчанию, в которой просто вызывается функция OnDraw. Но, как вы помните, для вывода на принтер нескольких страниц необходимо организовать цикл печати. И такой цикл действительно организован в обработчике:

void CView::**OnFilePrint**( )

Именно из него осуществляются вызовы всех рассмотренных функций, и в большинстве случаев вполне достаточно переопределить те из них, которые необходимы для индивидуальной организации печати документа.

Второй момент, на который следует обратить внимание, связан с принудительным завершением вывода данных на принтер, т. е. с функциями AbortDoc и SetAbortProc из Win32 API. Как уже отмечалось, нужно установить при помощи функции SetAbortProc специальную функцию, которая и будет отслеживать требование на завершение печати, периодически вызывая функцию CDC::QueryAbort, сообщающую, можно ли продолжать процесс печати или его нужно срочно завершить.

Всех этих сложностей можно избежать, если оставить функцию CView::OnFilePrint в покое. Пусть она делает свое дело, тем более, что у нее это получается хорошо.

Остановившись на последнем варианте (не переопределяя функцию OnFilePrint), продолжим рассмотрение процесса печати и того, как он реализован в библиотеке классов MFC.

Начиная по командам ID_FILE_PRINT или ID_FILE_PRINT_DIRECT процесс печати, библиотека MFC вызывает функцию OnPreparePrinting вашего объекта "представление", передавая ей в качестве параметра указатель на уже подготовленную структуру CPrintInfo. Поскольку в этот момент приложению неизвестно количество страниц печатаемого документа, то значения первой и последней печатаемой страницы устанавливаются, соответственно, в 1 и −1 (0×FFFF). Здесь удобнее всего установить длину документа (число его страниц), вызвав функцию SetMaxPage структуры CPrintInfo. Второй задачей функции OnPreparePrinting является вызов DoPreparePrinting, которая выводит на экран блок диалога **Print** и создает контекст устройства для принтера. Если по каким-либо причинам вас не устраивает внешний вид блока диалога или начальные установки принтера, то их можно изменить перед вызовом этой функции. Те параметры, которые пользователь установил в блоке диалога, доступны для использования через поля структуры CPrintInfo. Так, например, если пользователь установил номера страниц, которые требуется напечатать, то их можно получить при помощи функций GetFromPage и GetToPage. Лаконичность последней фразы не должна создавать иллюзию, что все так уж просто. Для того чтобы воспользоваться функциями GetFromPage и GetToPage, сначала необходимо проверить состояние группы переключателей **Page range** (диапазон печати). Функции вернут достоверные значения только в том случае, если установлен переключатель **Pages** (страницы).

Примечание

Следует иметь в виду, что при выполнении команды ID_FILE_PRINT_DIRECT блок диалога не выводится, а непосредственно начинается процесс печати документа.

Следующей функцией, вызываемой библиотекой MFC, является функция OnBeginPrinting. Ее наличие в цикле печати свидетельствует о похвальном желании разработчиков библиотеки предоставить программистам возможность именно здесь определиться и получить от системы все необходимые ресурсы GDI. Правда, на практике мало кто использует эту возможность — обычно ресурсы запрашиваются у системы непосредственно в функциях OnDraw или OnPrint, и там же, кстати говоря, и возвращаются ей обратно.

После того как функция CDC::StartDoc сформирует задание на печать, начинается собственно цикл, включающий в себя две интересующих нас функции — OnPrepareDC и OnPrint, которые библиотека вызывает для каждой печатаемой страницы, сообщая представлению ее номер в поле m_nCurPage структуры CPrintInfo. Первая функция служит для решения трех основных задач:

- Определение области вывода (view port)
- Осуществление проверки на конец документа
- Посылки на принтер специальных ESC-кодов (при необходимости)

Следует также отметить, что эта функция вызывается как для печати документа, так и для отображения его на экране. Различие заключается в первом передаваемом ей параметре — указателе на контекст принтера или экрана, и в том, что при выводе на экран в качестве второго параметра передается NULL. Для самой печати страницы предназначена функция OnPrint, поскольку передаваемые ей параметры несут всю необходимую для этого информацию. Однако, особенно если посмотреть на ее реализацию по умолчанию в классе CView:

```
void CView::OnPrint(CDC* pDC, CPrintInfo*)
{
  ASSERT_VALID(pDC);
  OnDraw(pDC);
},
```

приходишь к выводу, что более правильным подходом является такой, при котором эта функция "занимается своим делом", т. е. отвечает за изображение данных, специфичных именно для вывода на принтер, например, для печати колонтитулов и примечаний, оставляя всю остальную часть работы функции OnDraw.

Завершается цикл печати вызовом функций CDC::EndDoc и OnEndPrinting. И если первая функция завершает процесс печати документа, то вторая призвана освободить все использованные при печати ресурсы GDI.

Таким образом, для того чтобы "организовать" процесс печати, используя возможности, предоставляемые библиотекой классов MFC, достаточно переопределить некоторые (или все) из функций OnPreparePrinting, OnBeginPrinting, OnPrepareDC, OnPrint и OnEndPrinting. Правда, и это немало — чего стоит только определение длины документа!

В заключение обсуждения организации вывода документа на принтер рассмотрим пример переопределения перечисленных функций. Подробные комментарии делают излишним более детальное их обсуждение. Скажем лишь, что рассматриваемые фрагменты демонстрируют вывод колонтитула и номера страницы при печати документа.

```
void CNoteView::OnPrint(CDC* pDC, CPrintInfo* pInfo)
{
CNDApp *pApp = (CNDApp*)AfxGetApp();
  // Создаем контекст устройства для принтера
  CDC dc;
  pApp->CreatePrinterDC(dc);
  // Определяем размеры бумаги, исходя из установок принтера
  CSize sizePaper = CSize(dc.GetDeviceCaps(HORZSIZE)*10,
                          dc.GetDeviceCaps(VERTSIZE)*10);
```

```
  CRect rect(0, 0, sizePaper.cx, sizePaper.cy);
...
// Учитываем величины полей
  rect.left   += pApp->m_rtMargin.left;
  rect.right  -= pApp->m_rtMargin.right;
  rect.top    -= pApp->m_rtMargin.top;
  rect.bottom += pApp->m_rtMargin.bottom;
// Преобразуем логические координаты в физические
  pDC->LPtoDP(&rect);
// Изменяем параметры прямоугольника рисования
  pInfo->m_rectDraw.left   = rect.left;
  pInfo->m_rectDraw.right  = rect.right;
  pInfo->m_rectDraw.top    = rect.top;
  pInfo->m_rectDraw.bottom = rect.bottom;
// Устанавливаем новые координаты области вывода
  pDC->SetViewportOrg(rect.left, rect.top);
  CString strHeader = GetDocument()->GetTitle();
// Печатаем заголовок
  PrintHeader(pDC, pInfo, strHeader);
// Рисуем данные документа
  OnDraw(pDC);
}
void CNoteView::PrintHeader(CDC *pDC,CPrintInfo *pInfo,CString strHeader)
{
  // Устанавливаем выравнивание влево
  pDC->SetTextAlign(TA_LEFT);
  // Для учета величины смещения получаем характеристики шрифта
  TEXTMETRIC tm;
  pDC->GetTextMetrics(&tm);
  int y = tm.tmHeight + 15;
  // Выводим заголовок
  pDC->TextOut(25, y, strHeader);
  // Формируем смещение для линии
  y -= tm.tmHeight + 5;
  // Определяем длину текста
  CSize len = pDC->GetTextExtent(strHeader);
  // Рисуем разделительную линию
  pDC->MoveTo(0, y);
  pDC->LineTo(len.cx + 250, y);
```

```
// Устанавливаем выравнивание вправо
  pDC->SetTextAlign(TA_RIGHT);
  TCHAR szBuf[32];
  // Формируем строку с номером страницы
  wsprintf(szBuf, "%d", pInfo->m_nCurPage);
  // Выводим номер страницы в правом нижнем углу
  pDC->TextOut(pInfo->m_rectDraw.right - 525,
           pInfo->m_rectDraw.bottom - 1250, szBuf);
}
```

Предварительный просмотр документа

Помимо рассмотренных возможностей вывода документа на экран и на принтер, библиотека классов MFC поддерживает еще один не менее важный и интересный режим. Речь идет о предварительном просмотре документа перед печатью, т. е. получении на экране образа одной или нескольких страниц в том виде, в каком они будут выведены на принтер.

Для начала перечислим те возможности, которые библиотека предоставляет "по умолчанию". Другими словами — то, что получаешь, добавив к своей программе единственную строчку кода в карту сообщений своего класса представления:

```
ON_COMMAND(ID_FILE_PRINT_PREVIEW, CScrollView::OnFilePrintPreview)
```

Примечание

...если за вас это еще не сделал мастер AppWizard (надеемся, что не требует пояснения возможность использования вместо класса CScrollView любого другого класса представления).

А возможности достаточно широки и включают, помимо отображения уменьшенного образа одной или двух страниц документа, печать изображаемых страниц или всего документа, переход к предыдущей и/или следующей странице, масштабирование изображения и выход из режима предварительного просмотра. А если предварительно сообщить библиотеке MFC размер документа, т. е. число страниц, то она дополнительно выведет полосу прокрутки для перехода от одной страницы к другой.

Но... Нет предела совершенству — всегда хочется чего-то большего! И библиотека классов MFC предоставляет достаточно широкие возможности для реализации собственных замыслов. Начнем с самого простого, опустив вопросы различия контекстов принтера и экрана.

Для режима предварительного просмотра характерна та же последовательность действий, что и при выводе на печать (см. рис. 21.2). Библиотека

MFC сама формирует необходимый контекст устройства. Поэтому, как и при печати документа, задачей программиста является, всего лишь, переопределение функций OnPreparePrinting, OnBeginPrinting, OnPrepareDC, OnPrint и OnEndPrinting. Поскольку эти действия достаточно просты, ограничимся несколькими фрагментами кода программы (с подробными комментариями), которые иллюстрируют описываемый материал примером вывода рамки вокруг текста в режиме предварительного просмотра (рис. 21.4).

```
BOOL CNoteView::OnPreparePrinting(CPrintInfo* pInfo)
{
  if (pInfo != NULL && !pInfo->m_bPreview)  {
    // Подготовка данных для инициализации блока диалога
    // Print (Печать)
    ...
  }
  // Не забываем вызвать функцию базового класса
  return DoPreparePrinting(pInfo);
}
void CNoteView::OnPrint(CDC* pDC, CPrintInfo* pInfo)
{
  ...
  // В режиме предварительного просмотра рисуем рамку вокруг текста
  if (pInfo != NULL && pInfo->m_bPreview)
    DrawMargins(pDC, rect);
  // Выводим сам текст
  OnDraw(pDC);
}
void CNoteView::DrawMargins(CDC* pDC, CRect &rt)
{
  CNDApp *pApp = (CNDApp*)AfxGetApp();
  if (pDC->m_hAttribDC != NULL)  {
    CRect rect, rectMargin = pApp->m_rtMargin;
    // Устанавливаем границы для рамки
    rect.left   =  pApp->m_rtMargin.left - 5;
    rect.right  =  rt.right - pApp->m_rtMargin.right + 10;
    rect.top    = -pApp->m_rtMargin.top;
    rect.bottom = -rt.bottom + pApp->m_rtMargin.bottom;
    // Создаем "карандаш"
    CPen pen(PS_DOT, 0, pDC->GetTextColor());
    // Выбираем его в контекст устройства
```

```
    CPen* ppen = pDC->SelectObject(&pen);
    // Рисуем рамку
    pDC->MoveTo(0, rect.top);
    pDC->LineTo(10000, rect.top);
    pDC->MoveTo(rect.left, 0);
    pDC->LineTo(rect.left, -10000);
    pDC->MoveTo(0, rect.bottom);
    pDC->LineTo(10000, rect.bottom);
    pDC->MoveTo(rect.right, 0);
    pDC->LineTo(rect.right, -10000);
    // Удаляем карандаш из контекста устройства
    pDC->SelectObject(ppen);
  }
}
```

Рис. 21.4. Режим предварительного просмотра документа

Рассмотренной техники вполне достаточно в подавляющем большинстве случаев. Однако разработчики библиотеки MFC изменили бы себе, если бы не предоставили дополнительные возможности для работы в этом режиме.

Итак, при выполнении команды ID_FILE_PRINT_PREVIEW вместо непосредственного отображения образа документа на принтере необходима его эмуляция на экране. Для этого в библиотеке MFC реализованы два специальных класса: CPreviewDC и CPreviewView. Основную роль в эмуляции принтера играет объект класса CPreviewDC, производного от CDC. Как известно, все объекты CDC содержат члены двух контекстов устройства — собственно контекст устройства и контекст отображения, которые обычно совпадают. Но у объекта класса CPreviewDC они используются для разных целей: контекст устройства представляет принтер, а контекст отображения — экран, на ко-

торый и осуществляется вывод. И когда приложение выполняет какие-либо операции, которые изменяют характеристики контекста принтера, библиотека MFC автоматически выполняет необходимые операции эмуляции для контекста экрана. Например, если приложение устанавливает какой-либо шрифт для печати, то библиотека выбирает шрифт для экрана, который эмулирует шрифт принтера.

Вторым классом, участвующим в организации работы режима предварительного просмотра, является CPreviewView, базирующийся на классе CView. Этот класс предоставляет поддержку для уникальных особенностей окна предварительного просмотра, таких как отображение и функционирование панели инструментов, возможность просмотра одной или двух страниц, изменение масштаба изображения, например, увеличение просматриваемого образа. Класс спроектирован достаточно органично и в большинстве случаев нет необходимости непосредственно вызывать или переопределять какую-либо его функцию. Другое дело, если требуется реализовать свой интерфейс для предварительного просмотра (например, если необходима поддержка редактирования в этом режиме). Для тех, кому это может понадобиться, рассмотрим необходимую последовательность действий.

Поскольку все начинается с выбора команды ID_FILE_PRINT_PREVIEW, которой соответствует обработчик OnFilePrintPreview, рассмотрим его реализацию, предоставляемую по умолчанию, поскольку при переопределении придется делать нечто подобное:

```
void CView::OnFilePrintPreview()
{
  CPrintPreviewState* pState = new CPrintPreviewState;
  if (!DoPrintPreview(AFX_IDD_PREVIEW_TOOLBAR, this,
                      RUNTIME_CLASS(CPreviewView), pState))  {
    TRACE0("Error: DoPrintPreview failed.\n");
    AfxMessageBox(AFX_IDP_COMMAND_FAILURE);
    delete pState;
  }
}
```

Для нас интерес здесь представляют структура библиотеки MFC CPrintPreviewState и функция DoPrintPreview. Начнем со структуры:

```
struct CPrintPreviewState {
  UINT nIDMainPane;       // Идентификатор скрываемой основной области
  HMENU hMenu;            // Сохраненный дескриптор меню
  DWORD dwStates;     // Состояние видимости панелей элементов управления
  CView *pViewActiveOld; // Сохраненное представление
// Указатель на функцию, позволяющую "безопасно" выйти
// из режима предварительного просмотра
```

```
  BOOL(CALLBACK *lpfnCloseProc)(CFrameWnd *pFrameWnd);
  HACCEL hAccelTable;    // Сохраненная таблица акселераторов
  CPrintPreviewState();  // Конструктор
};
```

Основным ее назначением является сохранение параметров активного фрейма, которые он имел перед переходом в режим предварительного просмотра. При необходимости их можно использовать в этом режиме.

Основную роль играет, конечно, функция DoPrintPreview:

BOOL CView::**DoPrintPreview**(
UINT nIDResource,
CView* pPrintView,
CRuntimeClass* pPreviewViewClass,
CPrintPreviewState* pState) —

выполняет основную работу для подготовки и перехода в режим предварительного просмотра. Параметры: nIDResource — идентификатор ресурса панели инструментов в режиме предварительного просмотра; pPrintView — указатель на объект "представление", для которого выполняется предварительный просмотр (чаще всего используется значение this, но оно может быть и другим, если режим предварительного просмотра реализуется не для одного представления); pPreviewViewClass — указатель на структуру времени выполнения класса CPreviewView, что необходимо для динамического создания объекта этого класса во время выполнения функции; pState — указатель на структуру CPrintPreviewState (или производную от нее, если требуются какие-либо дополнительные параметры), которая должна создаваться именно здесь — функция DoPrintPreview немодальная, и структура существует до завершения работы в этом режиме.

Для того чтобы совершенно ясно представлять важность действий, выполняемых этой функцией, перечислим основные задачи, которые она решает:

- Создание объекта класса CPreviewView — он обеспечивает поддержку уникальных особенностей окна предварительного просмотра, таких, например, как отображение и функционирование панели инструментов
- Создание панели инструментов на основе шаблона, задаваемого своим идентификатором, — это может быть как стандартный (AFX_IDD_PREVIEW_TOOLBAR), так и созданный программистом блок диалога
- Создание и перевод в активное состояние окна Windows, базирующегося на объекте класса CPreviewView
- Установка параметров структуры CPrintInfo, необходимой как при печати, так и при предварительном просмотре документа — в последнем случае полю m_bPreview присваивается значение TRUE
- Вызов рассмотренных ранее функций OnPreparePrinting и OnBeginPrinting
- Определение размеров текущего листа бумаги и вывод на экран текущей страницы

Как видите, задач много, и если кто-то вместо предоставляемой захочет реализовать свою функцию, то не следует выпускать какую-либо из них из по-

ля зрения. Но прежде чем это делать, стоит задаться вопросом — "а зачем?", и только после получения убедительного ответа продолжать работать в этом направлении. В действительности библиотека MFC предоставляет достаточный простор для творчества и без такого рода работы, позволяя активно использовать механизм виртуальных функций. Вот и для режима предварительного просмотра она реализует две такие функции:

virtual void CFrameWnd::**OnSetPreviewMode**(
BOOL bPreview,
CPrintPreviewState *pModeStuff) —

переводит главное окно приложения в режим предварительного просмотра и обратно, в зависимости от значения параметра bPreview. Реализация по умолчанию блокирует все стандартные панели инструментов и скрывает основное меню и рабочую область главного окна приложения, т. е. переводит фрейм MDI во временный фрейм SDI. При переопределении следует вызывать аналогичную функцию базового класса для проведения необходимых инициализирующих действий. Параметр pModeStuff содержит указатель на структуру CPrintPreviewState.

virtual void CFrameWnd::**OnEndPrintPreview**(
CDC *pDC,
CPrintInfo *pInfo
POINT point,
CPreviewView *pView) —

вызывается библиотекой классов MFC, когда пользователь выходит из режима предварительного просмотра. Реализация этой функции по умолчанию вызывает функцию OnEndPrinting и восстанавливает главное окно приложения в том состоянии, которое оно имело до начала предварительного просмотра. При ее переопределении необходим вызов функции базового класса OnEndPrintPreview после всех дополнительных действий. Параметр point определяет последнюю изображаемую точку страницы в режиме предварительного просмотра, а pView — указывает на используемый объект "представление". Именно внутри этой функции следует освободить память, занимаемую структурой CPrintPreviewState, на которую указывает pState, значение которого устанавливается функцией DoPrintPreview.

Представленных функций вполне достаточно в подавляющем большинстве случаев. Переопределяя предназначенные специально для этого функции, можно без лишних усилий решать такие, например, задачи:

- Как ввести полосу прокрутки для доступа к страницам документа в режиме предварительного просмотра
- Как в режиме предварительного просмотра отображать страницы не с первой, а с текущей
- Как организовать различный вывод на принтер и в окно предварительного просмотра

Естественно, список может быть значительно шире. Мы же в заключение этого раздела приведем небольшой пример, иллюстрирующий некоторые из перечисленных возможностей.

```
void CDrawView::OnFilePrintPreview()
{
  // При переопределении функции CView::OnFilePrintPreview
  // необходимо самим создать структуру для сохранения
  // текущего состояния фрейма
  CPrintPreviewState* pState = new CPrintPreviewState;

  // Обязательный вызов, в котором задаем идентификатор
  // блока диалога для отображаемой панели инструментов
  if (!DoPrintPreview(IDD_PREVIEW, this,
                      RUNTIME_CLASS(CPreviewView), pState))
  {
    // Для отображения в окне отладки
    TRACE0("Ошибка: DoPrintPreview отработала неправильно.\n");
    // Для отображения на экране
    AfxMessageBox(AFX_IDP_COMMAND_FAILURE);
    delete pState;
  }
}
```

Внешний вид новой панели инструментов, активизирующейся в режиме предварительного просмотра, представлен на рис. 21.5.

Рис. 21.5. Окно предварительного просмотра с измененной панелью инструментов

Поскольку как при выводе на принтер, так и в режиме предварительного просмотра необходимо привязываться к размерам листа бумаги, в качестве еще одного примера рассмотрим настройку блока диалога **Page Setup** (параметры страницы) (рис. 21.6).

```
class CNoteView : public CScrollView
{
...
public:
  //{{AFX_MSG(CNoteView)
  afx_msg void OnPageSetup();
  //}}AFX_MSG
DECLARE_MESSAGE_MAP()
};
BEGIN_MESSAGE_MAP(CNoteView, CScrollView)
  //{{AFX_MSG_MAP(CNoteView)
  ON_COMMAND(ID_PAGE_SETUP, OnPageSetup)
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()
void CNoteView::OnPageSetup()
{
  CNDApp *pApp = (CNDApp*)AfxGetApp();
  // Создаем объект блока диалога Page Setup
  CPageSetupDialog dlg;
  // Настройка внешнего вида осуществляется с помощью структуры
  PAGESETUPDLG& psd = dlg.m_psd;
  // Будем настраивать значения полей, а размеры задаются
  // в сотых долях миллиметра
  psd.Flags |= PSD_MARGINS | PSD_INHUNDREDTHSOFMILLIMETERS;
  // Устанавливаем необходимые значения
  psd.rtMargin.left   = pApp->m_rtMargin.left*10;
  psd.rtMargin.right  = pApp->m_rtMargin.right*10;
  psd.rtMargin.top    = pApp->m_rtMargin.top*10;
  psd.rtMargin.bottom = pApp->m_rtMargin.bottom*10;
  // Для фиксации значений получаем параметры принтера
  PRINTDLG pd;
  pd.hDevNames = NULL;
  pd.hDevMode = NULL;
  pApp->GetPrinterDeviceDefaults(&pd);
  psd.hDevNames = pd.hDevNames;
  psd.hDevMode  = pd.hDevMode;
```

```
  // Выводим блок диалога на экран
  if(dlg.DoModal() == IDOK)  {
    // Если пользователь нажал кнопку OK, фиксируем его установки
    pApp->m_rtMargin.left   = psd.rtMargin.left/10;
    pApp->m_rtMargin.right  = psd.rtMargin.right/10;
    pApp->m_rtMargin.top    = psd.rtMargin.top/10;
    pApp->m_rtMargin.bottom = psd.rtMargin.bottom/10;
    // При необходимости заменяем принтер, используемый по умолчанию
    pApp->SelectPrinter(psd.hDevNames, psd.hDevMode, FALSE);
    CDC dc;
    pApp->CreatePrinterDC(dc);
    // Получаем размеры бумаги
    pApp->m_sizePaper = CSize(dc.GetDeviceCaps(HORZSIZE)*10,
                              dc.GetDeviceCaps(VERTSIZE)*10);
    // После того как были изменены параметры страницы,
    // необходимо обновить изображение данных на экране
    CNoteDoc* pDoc = (CNoteDoc*)GetDocument();
    // Сообщение об обновлении передается всем представлениям,
    // ассоциированным с документом, но реально предназначается
    // только объекту класса, на который указывает this
    pDoc->UpdateAllViews(NULL, 0, this);
  }
}
```

Рис. 21.6. Блок диалога **Page Setup**

ЭЛЕМЕНТЫ УПРАВЛЕНИЯ

Глава 22

Обзор стандартных элементов управления

Библиотека MFC предоставляет набор классов, которые облегчают создание стандартных элементов управления Windows. Они включают кнопки различных видов, статический текст, поле редактирования текста, полосы прокрутки, список и комбинированный список. В табл. 22.1 приведены названия классов и стандартные элементы управления, создаваемые этими классами.

Таблица 22.1. *Классы MFC для создания стандартных элементов управления*

Класс	Элемент управления
CStatic	Статический текст или изображение
CButton	Командные кнопки, срабатывающие по нажатию, переключатели и флажки
CListBox	Список
CComboBox	Комбинированный блок списка и текстового поля
CEdit	Текстовое поле
CScrollBar	Полоса прокрутки

Кроме вышеперечисленных классов, библиотека предоставляет разработчикам еще четыре класса — CBitmapButton, CComboBoxEx, CCheckListBox и CDragListBox, которые, базируясь на стандартных элементах управления и соответствующих классах, существенно упрощают создание и управление элементами управления с расширенными возможностями. Эти возможности ранее можно было реализовать только самостоятельно, написав обработчики для ряда сообщений. Теперь можно просто пользоваться тем, что сделано другими.

Создание элементов управления

Любой стандартный элемент управления можно создать двумя способами:

- В редакторе ресурсов
- Прямо в исходном тексте приложения

Создание элементов управления в редакторе ресурсов

Элементы управления можно создавать во время создания шаблона блока диалога при помощи редактора ресурсов. При этом пользователь может просто перетаскивать с помощью мыши необходимые элементы управления из панели с их изображениями в создаваемый блок диалога, изменяя по своему желанию размеры, расположение и параметры выбранного элемента управления. При этом редактор ресурсов помещает описание создаваемого элемента блока диалога в файл ресурсов. Когда блок диалога будет создаваться (вызовом функций Create или DoModal), система Windows сама создаст элементы управления и разместит их в этом блоке диалога. Причем это верно для всех окон, создаваемых на базе шаблона блока диалога, — модальных и немодальных блоков диалога и форматных представлений.

Создание элементов управления в тексте приложения

Элементы управления можно создать непосредственно в исходном тексте приложения. Как и другие объекты классов библиотеки MFC, инкапсулирующие визуальные объекты Windows, элементы управления создаются в два этапа: сначала необходимо создать объект данного конкретного класса, а затем вызвать функцию-член Create для создания соответствующего элемента управления Windows и связывания его с созданным объектом класса. Обычно для блока диалога это выполняется в функции OnInitDialog, а для обычного окна — в функции OnCreate.

Следующий пример показывает создание поля редактирования в тексте приложения.

Объект CEdit объявлен в декларации класса CMarsDialog:

```
class CMarsDialog : public CDialog
{
...
protected:
  CEdit m_edit;
...
```

```
public:
  virtual BOOL OnInitDialog();
...
};
```

Затем в функции OnInitDialog для него задается прямоугольник, определяющий его размеры и расположение, после чего вызывается функция Create для создания поля редактирования Windows и связывания его с созданным объектом CEdit:

```
BOOL CMarsDialog::OnInitDialog()
{
  CDialog::OnInitDialog();
  ...
  CRect rect(20, 20, 100, 50);
  m_edit.Create(WS_CHILD | WS_VISIBLE | WS_TABSTOP |
                ES_AUTOHSCROLL | WS_BORDER, rect, this, IDC_EDIT);
  m_edit.SetFocus();
  ...
  return FALSE;
}
```

После создания элемента управления функцией Create была вызвана функция SetFocus для передачи фокуса ввода созданному полю редактирования. И затем функция OnInitDialog вернула значение FALSE — в этом случае фокус ввода устанавливается на первый элемент управления.

Для создания элементов управления можно в равной мере пользоваться как первым (используя редактор ресурсов) — этот способ применяется чаще, так и вторым способом (создавая элементы управления в коде программы).

Если элемент управления создается в блоке диалога, то он автоматически уничтожается, когда пользователь закрывает диалоговое окно. Если элемент управления создается программно внутри окна, то его необходимо уничтожать. Если он создан с помощью функции new, то, соответственно, необходимо вызывать функцию delete для уничтожения объекта.

Глава 23

Классы стандартных элементов управления

Статические элементы управления

В случаях, когда требуется вывести текст (например, заголовок другого элемента управления или поясняющий текст) или добавить какие-либо дополнительные рисованные элементы в блоке диалога, функции GDI, как правило, не используются, для этого в Windows существуют статические элементы управления.

Класс CStatic позволяет выводить на экран текстовые строки, различные закрашенные прямоугольники, значки, курсоры, битовые массивы и расширенные метафайлы. Часто они используются для объединения или разделения других элементов управления. Обычно статические элементы управления не требуют ввода и вывода информации, однако они могут уведомлять родительское окно о нажатии клавиши (в случае, если они созданы со стилем SS_NOTIFY).

Следуя выбранному нами принципу изложения материала, рассмотрим компоненты этого класса.

Создание

Как и все классы библиотеки MFC, класс CStatic имеет конструктор, который, создавая объект класса, никак не воздействует на систему, т. е. не создает объект Windows:

CStatic::**CStatic**( )

В отличие от конструктора, функция Create создает статический элемент управления Windows и присоединяет его к объекту класса. Функция возвращает значение TRUE в случае удачи и FALSE — в случае ошибки.

```
BOOL CStatic::Create(
    LPCTSTR lpszText, DWORD dwStyle,
    const RECT& rect, CWnd* pParentWnd,
    UINT nID = 0xffff)
```

Параметры: lpszText — задает текст, который должен быть помещен на элемент управления. Если этот параметр равен NULL, то не будет выведено никакого текста, что вполне естественно, т. к. при задании некоторых стилей статического элемента текст и не нужен. dwStyle — задает стиль статического элемента. Параметр может содержать комбинацию стилей, объединенных операцией OR. Необходимо всегда указывать стиль WS_CHILD, т. к. элементы управления не могут быть окнами верхнего уровня. Для того чтобы элемент управления был видимым, сразу после отображения его родительского окна необходимо указать стиль WS_VISIBLE. Если нужно, чтобы созданный элемент управления был недоступен для использования, то можно указать стиль WS_DISABLE. Кроме того, для создания требуемого статического элемента необходимо использовать следующие стили:

SS_BLACKFRAME	Задает прямоугольник с рамкой того же цвета, что и рамка окна, по умолчанию этот цвет — черный
SS_BLACKRECT	Задает прямоугольник, закрашенный цветом, совпадающим с цветом рамки окна, по умолчанию этот цвет — черный
SS_CENTER	Определяет прямоугольник и выводит заданный текст в его центре; текст форматируется до того, как он будет выведен на экран; слова, не помещающиеся на одной строке, автоматически переносятся на следующую, что, однако, не гарантирует отображения всего текста
SS_GRAYFRAME	Задает прямоугольник с рамкой того же цвета, что и цвет фона клиентской области родительского окна; для блоков диалога этот цвет по умолчанию серый
SS_GRAYRECT	Задает прямоугольник, закрашенный цветом, совпадающим с цветом фона клиентской области родительского окна; для блоков диалога этот цвет по умолчанию серый
SS_ICON	Задает значок для ее отображения в родительском окне; передаваемый текст (параметр lpszText) является именем значка (но не именем файла), определенного в файле ресурса; параметры nWidth и nHeight при этом игнорируются, размеры значка определяют размеры элемента управления
SS_LEFT	Определяет прямоугольник и выводит текст, выровненный по левому краю; текст форматируется до того, как он будет выведен на экран; слова, не помещающиеся на одной строке, автоматически переносятся на следующую, что, однако, не гарантирует отображения всего текста
SS_LEFTNOWORDWRAP	Определяет прямоугольник и выводит текст, выровненный по левому краю; символы табуляции заменяются пробелами, но слова автоматически не переносятся на новую строку. В качестве символа переноса используются символы "\n"; текст, не помещающийся в строке, обрезается
SS_NOPREFIX	Если этот стиль не определен, то Windows интерпретирует символ амперсанда (&) в тексте элемента управления как префикс символа-акселератора; в этом случае, символ "&" удаляется и следующий за ним символ в строке подчеркивается; если статический элемент управления содержит текст, где эта особенность не требуется, то может быть добавлен данный стиль; чаще всего стиль используется, когда имена файлов или другие строки, которые будут выведены в статическом элементе управления, могут содержать символ амперсанда

SS_RIGHT	Определяет прямоугольник и выводит текст, выровненный по правому краю; текст форматируется до того, как он будет выведен на экран; слова, не помещающиеся в строке, автоматически переносятся на следующую, что, однако, не гарантирует отображения всего текста
SS_SIMPLE	Задает прямоугольник и выводит строку текста, выровненного по левому краю; строка текста не может быть укорочена или изменена; при этом родительское окно или блок диалога не должны обрабатывать сообщение WM_CTLCOLOR
SS_USERITEM	Статический элемент управления, определяемый пользователем
SS_WHITEFRAME	Задает прямоугольник с рамкой, которая по умолчанию имеет белый цвет
SS_WHITERECT	Задает закрашенный прямоугольник по умолчанию белого цвета

Если необходимо вывести на экран битовый массив, значок, курсор или метафайл, то нужно использовать следующие стили:

SS_BITMAP	Стиль для битовых массивов
SS_ICON	Стиль для курсоров и значков
SS_ENHMETAFILE	Стиль для расширенных метафайлов

Для курсоров, битовых массивов или значков можно использовать также стиль:

SS_CENTERIMAGE	Располагает изображение в центре статического элемента управления

rect — задает положение и размер статического элемента управления. Это может быть структура типа RECT или объект класса CRect. pParentWnd — задает родительское окно объекта CStatic, обычно это объект класса CDialog. Этот параметр должен быть обязательно определен, т. е. не должен быть равен NULL. nID — задает идентификатор статического элемента управления.

В следующем фрагменте кода создается объект класса CStatic:

```
class CNeptuneDialog : public CDialog
{
...
  protected:
    CStatic m_static;
public:
  virtual BOOL OnInitDialog();
};
```

Затем вызывается функция Create для создания элемента управления и связывания его с объектом m_static:

```
BOOL CNeptuneDialog::OnInitDialog()
{
```

```
...
  CDialog::OnInitDialog();
  CRect rect(20, 20, 100, 40);
  m_static.Create("Visual C++ и библиотека классов MFC",
    WS_CHILD | WS_VISIBLE | SS_LEFT, rect, this, IDC_STATIC);
...
}
```

В результате в диалоговой панели будет выведен статический текст "Visual C++ и библиотека классов MFC".

Отображение битового массива

Для работы с битовым массивом предназначены две функции GetBitmap и SetBitmap.

HBITMAP CStatic::**GetBitmap**( ) —

возвращает дескриптор битового массива, ранее установленного с помощью SetBitmap, в случае отсутствия связанного битового массива с элементом управления или ошибки возвращается NULL.

HBITMAP CStatic::**SetBitmap**(HBITMAP hBitmap) —

задает дескриптор битового массива для его вывода в рамках статического элемента управления (параметр hBitmap).

Вторая функция возвращает дескриптор предыдущего битового массива, связанного со статическим элементом управления, или NULL, если до этого с ним не был связан битовый массив.

Битовый массив будет автоматически выведен в границах элемента управления. По умолчанию, левые верхние углы будут совпадать, а размер элемента управления будет изменен в соответствии с размером изображения. Для элемента, отображающего битовый массив, можно использовать различные стили окон и статических элементов управления, включая следующие:

SS_BITMAP	Всегда используется, когда выводится битовый массив
SS_CENTERIMAGE	Для центрирования изображения в рамках элемента управления; если изображение больше, чем элемент управления, то оно будет обрезано; если изображение меньше, чем элемент управления, то оставшееся место вокруг изображения будет заполнено цветом первого пикселя левого верхнего угла битового массива

Код создания статического элемента управления в середине окна может выглядеть примерно так:

```
CStatic bmpStatic;
int CVenusFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
```

```
{
  ...
  CBitmap bitmap;
  HBITMAP bm;
  if (!bitmap.LoadBitmap("IDB_BITMAP"))
  {
    bitmap.GetBitmap(&bm);
    CRect rect(0, 0, lpCreateStruct->cx, lpCreateStruct->cy);
    bmpStatic.Create(NULL, WS_CHILD | WS_VISIBLE |
      SS_BITMAP | SS_CENTERIMAGE, rect, this, IDC_BMPSTATIC);
    bmpStatic.SetBitmap(bm);
  }
  ...
}
```

Отображение значка

Для работы со значками в классе реализованы две функции GetIcon и SetIcon.

HICON CStatic::**GetIcon**( ) —

возвращает дескриптор значка, ранее установленной функцией SetIcon, или NULL.

HICON CStatic::**SetIcon**(HICON hIcon) —

задает дескриптор значка, который будет выведен в рамках статического элемента управления (параметр hIcon).

Размеры элемента управления будут изменены в соответствии с размерами значка. Для элемента, отображающего значок, можно использовать различные стили окон и статических элементов управления, включая следующие:

SS_ICON	Всегда используется при выводе курсоров и значков
SS_CENTERIMAGE	Применяется для центровки изображения в рамках элемента управления; если изображение больше, чем элемент управления, то оно будет обрезано; если изображение меньше, чем элемент управления, то пустое место вокруг изображения будет заполнено цветом фона статического элемента управления

Отображение курсора

Для работы с курсором предназначены функции SetCursor и SetCursor.

HCURSOR CStatic::**GetCursor**( ) —

возвращает дескриптор изображения курсора, ранее установленного функцией SetCursor, или NULL, если ранее курсор не был установлен.

HCURSOR CStatic::**SetCursor**(HCURSOR hCursor) —

задает дескриптор курсора, который должен изображаться в рамках статического элемента управления (параметр hCursor). Функция возвращает дескриптор ранее установленного курсора и NULL — в противном случае.

Действия по размещению курсора, задаваемых параметров и т. д. полностью аналогичны отображению значков.

Отображение расширенных метафайлов

Для работы с расширенными метафайлами имеются функции:

HENHMETAFILE CStatic::**GetEnhMetaFile**( ) —

возвращает дескриптор расширенного метафайла, ранее установленного функцией SetEnhMetaFile, или NULL, если ранее расширенный метафайл не был установлен.

HENHMETAFILE CStatic::**SetEnhMetaFile**(HENHMETAFILE hMetaFile) —

задает дескриптор расширенного метафайла для его отображения в рамках статического элемента управления (параметр hMetaFile).

Для элемента, отображающего метафайл, наряду с использованием стилей окон и статических элементов управления, необходимо указывать следующий:

SS_ENHMETAFILE	Всегда используется при выводе расширенных метафайлов

Изменение цвета

Следующий пример иллюстрирует, как в блоке диалога в ответ на сообщение WM_CTLCOLOR изменяется "кисть" для закрашивания фона статических элементов управления.

Для этого необходимо создать обработчик ON_WM_CTRLCOLOR. Проще всего это сделать с помощью мастера ClassWizard. Обработчик получает параметр, сообщающий тип элемента управления, для которого запрашиваются параметры рисования. В нашем случае задаем только другую "кисть", т. к. единственный статический элемент управления блока диалога не имеет текста, и достаточно лишь заменить цвет и стиль закрашивания фона.

```
//Файл StaticDlg.cpp
BEGIN_MESSAGE_MAP(CStaticDlg, CDialog)
  //{{AFX_MSG_MAP(CStaticDlg)
  ON_WM_PAINT()
  ON_WM_QUERYDRAGICON()
  ON_WM_CTLCOLOR()
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

```
BOOL CStaticDlg::OnInitDialog()
{
  CDialog::OnInitDialog();
  SetIcon(m_hIcon, TRUE);  // Set big icon
  SetIcon(m_hIcon, FALSE); // Set small icon
  // m_brush — член данных класса CStaticDlg типа CBrush.
  // Создаем "кисть" со штриховкой в виде "решетки",
  // штрих белого цвета
  m_brush.CreateHatchBrush(HS_CROSS, RGB(255, 255, 255));
  return TRUE;
}
// Обработчик сообщения WM_CTLCOLOR
HBRUSH CStaticDlg::OnCtlColor(
  CDC* pDC, CWnd* pWnd, UINT nCtlColor)
{
  HBRUSH hbr = CDialog::OnCtlColor(pDC, pWnd, nCtlColor);
  // Заменяем "кисть" только для статических
  // элементов управления
  if (nCtlColor == CTLCOLOR_STATIC)
    hbr = m_brush; // Происходит вызов функции-оператора
                   // CBrush::operator HBRUSH()
  return hbr;
}
//Файл Static.rc
// Dialog
IDD_STATIC_DIALOG DIALOGEX 0, 0, 185, 92
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
EXSTYLE WS_EX_APPWINDOW
CAPTION "Static"
FONT 8, "MS Sans Serif"
BEGIN
  DEFPUSHBUTTON   "OK",IDOK,128,7,50,14
  PUSHBUTTON      "Cancel",IDCANCEL,128,23,50,14
  LTEXT           "",IDC_STATIC,0,0,122,92,
                  WS_BORDER,WS_EX_TRANSPARENT
END
```

В результате блок диалога будет иметь следующий вид (рис. 23.1):

Рис. 23.1. Блок диалога приложения Static

Кнопки

Класс CButton предназначен для облегчения создания такого элемента управления Windows, как "кнопка". Кнопка — это маленькое прямоугольное дочернее окно, кнопка может предоставлять пользователю возможность инициировать определенные приложением действия или позволяет осуществлять выбор. Кнопки могут использоваться по отдельности или группами, могут быть с нанесенным изображением или без него. Кнопка обычно изменяет свой внешний вид, когда пользователь "нажимает" ее.

Типичные примеры кнопок (рис. 23.2) — это командные кнопки, срабатывающие по нажатию (PUSHBUTTON), флажки (CHECK BOX) и переключатели (RADIOBUTTON). Класс CButton позволяет создать любую из них, задав нужный стиль при создании кнопки с помощью функции Create. Кроме типовых кнопок, комбинируя те или иные стили, можно создавать самые разнообразные элементы управления. Таких стилей довольно много, и мы их сгруппировали по назначению.

Рис. 23.2. Кнопки с различными стилями

Ниже приведены стили, определяющие типы кнопок:

BS_PUSHBUTTON	Определяет простую кнопку, срабатывающую по нажатию; она посылает сообщение WM_COMMAND окну-владельцу, когда пользователь "нажимает"

BS_CHECKBOX	Определяет маленький квадратик с расположенным обычно справа текстом (если он не определен вместе со стилем BS_LEFTTEXT, располагающим текст по левой стороне кнопки); такая кнопка может иметь два состояния: отмеченное ("checked" — квадратик помечен крестиком) и неотмеченное ("unchecked" — квадратик пуст)
BS_RADIOBUTTON	Определяет небольшую круглую кнопку с расположенным справа от нее текстом (если не включен дополнительно стиль BS_LEFTTEXT, располагающий текст слева); кнопки этого типа обычно используются для реализации выбора одной возможности из нескольких (естественно, для этого следует определить несколько кнопок того же типа)

Кроме создания трех основных типов кнопок есть возможность объединения кнопок в группу, задав следующий стиль:

BS_GROUPBOX	Определяет прямоугольную рамку, объединяющую другие кнопки; в левом углу рамки может располагаться текст

Кроме основных вышеперечисленных стилей кнопок, существует еще несколько стилей, которые позволяют создать следующие виды кнопок:

BS_DEFPUSHBUTTON	Определяет кнопку, выделенную жирной рамкой; при нажатии клавиши <Enter> срабатывает именно эта кнопка; этот стиль позволяет пользователю быстро выбрать наиболее предпочтительный (с точки зрения разработчика приложения) элемент управления в диалоговом окне (команду или кнопку по умолчанию)
BS_AUTOCHECKBOX	То же, что и BS_CHECKBOX, но он автоматически меняет состояние при выборе
BS_AUTORADIOBUTTON	То же, что и BS_RADIOBUTTON, но кнопка-переключатель автоматически помечается при ее выборе пользователем, а с остальных кнопок группы отметка автоматически снимается
BS_3STATE	То же, что и BS_CHECKBOX, но элемент управления имеет три состояния; к двум первым (включен/выключен) добавляется третье, обозначающее некоторое среднее состояние; часто используется в программах-инсталляторах для индикации того, что хотя бы одна составляющая часть программного продукта выбрана для инсталляции
BS_AUTO3STATE	То же, что и BS_3STATE, но смена состояний производится автоматически
BS_LEFTTEXT	Может комбинироваться со стилями BS_CHECKBOX и BS_RADIOBUTTON для расположения текста слева от кнопок

BS_OWNERDRAW	Задает кнопку, внешний вид которой определяется пользователем; когда необходимо произвести перерисовку кнопки, вызывается функция-член CButton::DrawItem. Этот стиль необходимо устанавливать, если используется класс CBitmapButton

В Windows 95/98 и Windows NT определены дополнительные стили для кнопок. Задание их позволяет пользоваться и некоторыми функциями класса CButton:

BS_TEXT	Кнопка с текстом
BS_ICON	Кнопка со значком или изображением курсора
BS_BITMAP	Кнопка с битовым массивом
BS_LEFT	Выравнивание по левому краю в области вывода текста или изображения
BS_RIGHT	Выравнивание по правому краю в области вывода текста или изображения
BS_CENTER	Горизонтальное выравнивание по центру в области вывода текста или изображения
BS_TOP	Выравнивание по верхнему краю в области вывода текста или изображения
BS_BOTTOM	Выравнивание по нижнему краю в области вывода текста или изображения
BS_VCENTER	Вертикальное выравнивание по центру в области вывода текста или изображения
BS_PUSHLIKE	Кнопка изображается так же, как обычная кнопка, срабатывающая по нажатию (для флажков или переключателей)
BS_MULTILINE	Текст на кнопке может располагаться в несколько строк
BS_NOTIFY	При нажатии или двойном нажатии обычная кнопка посылает уведомляющее сообщение родительскому окну; в рамках библиотеки классов MFC этот стиль не имеет серьезного значения — и то и другое событие можно обрабатывать непосредственно при помощи соответствующих функций (см. ниже)
BS_FLAT	Кнопка изображается "плоской" без трехмерной рамки
BS_RIGHTBUTTON	То же, что и BS_LEFTTEXT

Кроме вышеперечисленных стилей, следует использовать следующие оконные стили:

WS_CHILD	Следует указывать всегда при создании кнопки
WS_VISIBLE	Следует указывать, если кнопка должна быть изначально видимой
WS_DISABLED	Следует указывать, если кнопка должна быть изначально недоступной

WS_GROUP	Необходим для объединения кнопок в группу, позволяет переходить от кнопки к кнопке при помощи клавиш со стрелками; обязательно должен быть использован для первого переключателя группы таких же переключателей и для следующего после последнего переключателя элемента управления
WS_TABSTOP	Необходим, если кнопка должна входить в группу кнопок, которым по очереди передается фокус ввода при нажатии клавиши <Tab>

Если требуется обрабатывать сообщения, посылаемые кнопкой родительскому окну (обычно это объект класса, производного от CDialog), то для каждого в таблицу (карту) сообщений нужно добавить обработчик.

Для кнопок определены только два вида таких сообщений:

ON_BN_CLICKED	Посылает родительскому окну сообщение, если пользователь "нажал" кнопку
ON_BN_DOUBLECLICKED	Посылает родительскому окну сообщение, если пользователь дважды подряд "нажал" кнопку (double click) (речь, естественно, идет о двойном щелчке кнопки мыши на данной кнопке)

Ниже приводится описание компонентов класса. Следует иметь в виду, что, как обычно, конструктор лишь создает объект класса CButton.

Создание

BOOL CButton::**Create**(
LPCTSTR lpszCaption, DWORD dwStyle,
const RECT& rect, CWnd* pParentWnd,
UINT nID) —

создает кнопку — элемент управления Windows и присоединяет ее к объекту Cbutton; lpszCaption — задает текст, который будет помещен на кнопке; dwStyle — задает тип кнопки (допускается любая комбинация стилей кнопок); rect — задает размер и расположение кнопки, это может быть объект класса CRect или структура типа RECT; pParentWnd — задает родительское окно кнопки, обычно это объект типа CDialog (этот параметр должен быть обязательно определен, т. е. не должен быть равен NULL); nID — задает идентификатор кнопки.

Примечание

На самом деле не имеет смысла задавать сразу все типы кнопок для создаваемого объекта. В этом случае либо возникает ошибка времени выполнения, либо результат не определен.

Определение и изменение текущего состояния кнопок

Для определения состояния кнопок предназначены следующие две функции: GetState и SetState.

Первая, GetState, позволяет узнать текущее состояние кнопки любого типа:

UINT CButton::**GetState**( ) —

возвращает текущее состояние кнопки: выбрана или нет, выделена или нет, имеет фокус ввода или нет.

Кнопки автоматически выделяются, если пользователь нажимает и удерживает левую кнопку мыши в момент, когда курсор мыши находится на кнопке. Выделение с кнопки снимается, как только пользователь отпускает кнопку мыши или перемещает курсор вне кнопки.

Необходимо использовать следующие маски для извлечения информации о состоянии кнопки:

0x0003 Для определения того, включена кнопка или нет (только для переключателей и кнопок выбора); 0 — кнопка не включена, 1 — кнопка включена; (переключатель включен, если он помечен жирной точкой; флажок включен, если его квадратик перечеркнут крестиком); 2 — кнопка находится в неопределенном состоянии (только для флажков, имеющих три состояния — флажок в неопределенном состоянии высвечивается серым цветом)

0x0004 Для определения того, выделена кнопка или нет (ненулевое значение показывает, что кнопка выделена)

0x0008 Для определения состояния фокуса ввода (ненулевое значение означает, что кнопка имеет фокус ввода)

Функция SetState позволяет изменять только одно из состояний:

void CButton::**SetState**(BOOL bHighlight) —

устанавливает кнопку в выделенное состояние; bHighlight — определяет, должна ли быть кнопка выделена цветом. Значение TRUE выделяет кнопку, FALSE — снимает выделение кнопки. Данная функция не влияет на состояние *флажков* и *переключателей*.

Следующие две функции предназначены только для переключателей и флажков:

int CButton::**GetCheck**( ) —

возвращает текущее состояние кнопки, которое для кнопок, созданных со стилями BS_AUTOCHECKBOX, BS_AUTORADIOBUTTON, BS_AUTO3STATE, BS_3STATE BS_CHECKBOX или BS_RADIOBUTTON может принимать следующие значения:

0 Кнопка не выбрана

1 Кнопка выбрана

2 Состояние кнопки не определено (только для кнопок, имеющих стиль BS_3STATE или BS_AUTO3STATE)

Если кнопка имеет какой-либо другой стиль кроме указанных выше, то возвращается нулевое значение.

void CButton::**SetCheck**(int nCheck) —

устанавливает состояние кнопки (выбрана или нет); nCheck — задает состояние кнопки. Параметр может принимать те же значения, что возвращает функция GetCheck.

Определение и изменение стиля кнопки

Для определения текущих стилей кнопки следует воспользоваться функцией GetButtonStyle, которая возвращает только стили, определенные для кнопок, т. е. имеющие префикс BS_:

UINT CButton::**GetButtonStyle**( )

А для изменения или установки стилей уже созданной кнопки в классе CButton определена функция SetButtonStyle:

void CButton::**SetButtonStyle**(
  UINT nStyle,
  BOOL bRedraw = TRUE) —

параметр nStyle задает стиль кнопки (см. выше), а параметр bRedraw определяет, должна ли быть кнопка перерисована после изменения стилей (значение TRUE приводит к перерисовке кнопки). По умолчанию параметр принимает значение TRUE, и кнопка будет перерисована немедленно. Возможность отложенной перерисовки позволяет поменять стили сразу нескольких элементов управления, возможно, проводя какие-либо длительные проверки, и только затем одновременно отобразить изменения.

Назначение изображений

Кнопкам, имеющим стили BS_ICON и BS_BITMAP, можно назначать изображения: значки, изображения курсоров (BS_ICON) и битовые массивы (BS_BITMAP). По умолчанию значок, битовый массив или курсор помещается в центре кнопки. Однако манипулируя соответствующими стилями, эти изображения можно расположить и иначе. Важно то, что если размер кнопки недостаточен для размещения значка, битового массива или курсора, то размещаемое изображение будет обрезано по размеру кнопки.

Все функции этой группы позволяют получать дескрипторы ранее назначенных изображений. Кроме того, отдельно реализованы функции назначения изображений (с префиксом Set) и функции только возвращающие дескрипторы (с префиксом Get).

Перечислим функции назначения или связывания изображений с объектом-кнопкой:

HICON CButton::**SetIcon**(HICON hIcon),
HCURSOR CButton::**SetCursor**(HCURSOR hCursor),
HBITMAP CButton::**SetBitmap**(HBITMAP hBitmap) —

параметры hIcon, hCursor и hBitmap являются дескрипторами соответствующих типов.

В отличие от класса CBitmapButton, который позволяет задавать до четырех изображений на одну кнопку для различных ее состояний, класс CButton использует только одно — когда кнопка нажимается, изображение сдвигается вниз и вправо.

Переопределяемые функции

Если при создании кнопки был задан стиль BS_OWNERDRAW, библиотека классов требует обязательного создания производного класса от класса CButton, в котором переопределена функция DrawItem. Дело в том, что наличие этого стиля при перерисовке объекта-кнопки приводит к вызову функции CButton::DrawItem или той же функции в переопределенном классе. Отсутствие переопределения функции приведет к аварийному завершению приложения в момент первой же попытки отображения кнопки. Связано это с тем, что реализация функции DrawItem просто генерирует ошибку:

```
void CButton::DrawItem(LPDRAWITEMSTRUCT)
{
  ASSERT(FALSE);
}
```

Все это происходит, однако, только в том случае, если объект CButton или производный от него связан с объектом Windows. Иначе кнопка просто не будет отображена — нет обработчика, нет и кнопки.

В заключение приведем определение функции, в которой происходит рисование пользовательской кнопки:

virtual void CButton::**DrawItem**(
LPDRAWITEMSTRUCT lpDrawItemStruct) —

параметр lpDrawItemStruct является указателем на структуру DRAWITEMSTRUCT. Эта структура содержит информацию о том, что и как должно быть нарисовано. В одном из полей этой структуры передается дескриптор контекста устройства, для которого следует вызывать функции рисования. Следует иметь в виду, что приложение должно восстановить все использовавшиеся объекты графического интерфейса (GDI), выбранные для этого контекста устройства, перед тем, как функция DrawItem будет завершена.

Простейший текстовый редактор

Класс CEdit позволяет создать такой элемент управления Windows, как "текстовое поле", который является простейшим редактором текста. Он представляет собой прямоугольное дочернее окно, в котором можно вводить текст. Когда фокус ввода принадлежит текстовому полю, пользователь может вводить в нем текст, редактировать его, перемещать курсор по тексту, выделять текст, копировать его в буфер обмена и обратно. Используя класс CEdit, можно создать как многострочный редактор текста, так и просто поля для ввода текста.

В большинстве случаев, особенно когда "текстовое поле" используется для реализации поля ввода, он функционирует автономно — приложение запрашивает введенный текст на момент запроса, никак не контролируя сам процесс ввода информации. Однако "текстовое поле" предусматривает возможность уведомления (извещения) родительского окна о некоторых собы-

тиях. Правильнее следует говорить не о возможности уведомления, а о возможности обработки уведомлений. Для связывания функций обработки предусмотрены макросы, добавляющие обработчик в карту сообщений родительского окна, созданного на базе производных классов от класса CWnd. Все эти макросы перечислены ниже совместно с описанием соответствующих уведомляющих сообщений:

ON_EN_CHANGE	Пользователь произвел действие, которое, возможно, изменило текст в "текстовом поле"; в отличие от сообщения EN_UPDATE, это сообщение посылается после того, как Windows уже обновила окно "текстового поля"
ON_EN_ERRSPACE	Элемент управления не может получить достаточно памяти для размещения текста
ON_EN_HSCROLL	Пользователь щелкнул на горизонтальной полосе прокрутки "текстового поля"; родительское окно уведомляется до того, как будет произведено обновление экрана
ON_EN_KILLFOCUS	"Текстовое поле" теряет фокус ввода
ON_EN_MAXTEXT	Текущая вставка превышает заданное число символов для "текстового поля" и поэтому она усечена; сообщение посылается, если "текстовое поле" не имеет стиля ES_AUTOHSCROLL, и число символов для вставки превышает ширину "текстового поля"; также посылается, если "текстовое поле" не имеет стиля ES_AUTOVSCROLL, и высота общего числа строк с текстом для вставки превышает высоту окна "текстового поля"
ON_EN_SETFOCUS	Посылается, когда "текстовое поле" получает фокус ввода
ON_EN_UPDATE	Сообщает, что "текстовое поле" собирается отобразить измененный текст на экране; посылается после того, как элемент управления отформатировал текст, но еще не вывел изменения на экране (с тем, чтобы можно было в случае необходимости изменить размеры окна)
ON_EN_VSCROLL	Пользователь щелкнул на вертикальной полосе прокрутки; родительское окно уведомляется до того, как экран будет перерисован

Инициализация

Использование класса CEdit может понадобиться как для элементов редактирования, созданных с помощью шаблона блока диалога, так и при чисто программной реализации элемента управления. Однако для создания собственно элементов управления требуется вызвать функцию Create:

BOOL CEdit::**Create**(
DWORD dwStyle, const RECT & rect,
CWnd* pParentWnd, UINT nID) —

создает простейшее "текстовое поле" Windows и присоединяет его к объекту CEdit. В случае успешного завершения функция возвращает значение TRUE и FALSE — в случае неудачи.

Параметр dwStyle задает стили создаваемого "текстового поля". Можно комбинировать следующие стили:

ES_AUTOHSCROLL	"Текстовое поле" автоматически прокручивает текст вправо на 10 символов, если пользователь вводит символ в конце строки; если пользователь нажимает клавишу ввода, то происходит переход к началу следующей строки
ES_AUTOVSCROLL	Автоматически прокручивает текст на одну строку, если пользователь нажал клавишу ввода на последней строке
ES_CENTER	Центрирует текст в многострочном "текстовом поле"
ES_LEFT	Выравнивает текст по левому краю
ES_LOWERCASE	Все символы, набираемые в "текстовом поле", преобразуются в строчные
ES_MULTILINE	Задает многострочное "текстовое поле" (по умолчанию "текстовое поле" создается как однострочное); если задан стиль ES_AUTOVSCROLL, то "текстовое поле" прокручивает при необходимости текст по вертикали; если же стиль ES_AUTOVSCROLL не задан, то "текстовое поле" показывает столько строк, сколько помещается в окне, а затем сообщает звуковым сигналом, если нажимается клавиша ввода, но уже нет места в окне для вывода строк; если задан стиль ES_AUTOHSCROLL, то многострочное "текстовое поле" автоматически прокручивает текст по горизонтали, когда каретка достигает правого края; для того чтобы начать новую строку, пользователь должен нажать клавишу ввода; если стиль ES_AUTOHSCROLL не задан, то "текстовое поле" автоматически переносит слова на начало новой строки; перенос слов определяется размерами окна; если размеры окна изменяются, то текст выводится заново с учетом нового размера окна; многострочное "текстовое поле" может иметь полосы прокрутки; "текстовое поле" с полосой прокрутки само обрабатывает сообщения от собственной полосы прокрутки; "текстовое поле" без полосы прокрутки осуществляет прокрутку, как описано выше, и обрабатывает сообщения для прокрутки, посылаемые ему родительским окном
ES_NOHIDESEL	"Текстовое поле" снимает выделение с текста, если теряет фокус ввода, и вновь его устанавливает при получении; задание стиля ES_NOHIDESEL отключает эту возможность, включенную по умолчанию
ES_OEMCONVERT	Текст, вводимый в "текстовое поле", конвертируется из набора символов ANSI в набор OEM, при передаче его обратно из "текстового поля" текст конвертируется в набор ANSI; это обеспечивает правильное преобразование символов, когда приложение вызывает функцию Win32 API ::AnsiToOem для преобразования строки ANSI в строку символов OEM; эта функция может быть полезна для "текстовых полей", содержащих имена файлов

ES_PASSWORD	Все символы, набираемые в "текстовом поле", изображаются на экране звездочками (*); приложение может использовать функцию CEdit::SetPasswordChar для изменения символов, которые выводятся на экран
ES_RIGHT	Выравнивает текст по правому краю в многострочном "текстовом поле"
ES_UPPERCASE	Все символы, набираемые в "текстовом поле", преобразуются в прописные
ES_READONLY	Предотвращает изменение текста в "текстовом поле" (текст предназначен только для чтения, но не для редактирования)
ES_WANTRETURN	Устанавливает режим, в котором при нажатии клавиши <Enter> должен вставляться символ возврата каретки; этот стиль не действует в однострочном "текстовом поле"; отсутствие этого стиля приведет при нажатии клавиши <Enter> к выполнению команды, заданной по умолчанию; чтобы этого избежать, следует нажимать комбинацию клавиш <Ctrl>+<Enter>

Кроме вышеперечисленных, можно использовать следующие стили:

WS_CHILD	Следует указывать всегда при создании текстового поля
WS_VISIBLE	Следует указывать, если текстовое поле должно быть изначально видимым
WS_DISABLED	Следует указывать, если текстовое поле должно быть изначально недоступным
WS_GROUP	Необходим для объединения элементов управления в группу (что вряд ли имеет смысл — до окна текстового поля, например, от кнопки можно "добраться", но "уйти" от него — нет, т. к. этот элемент управления самостоятельно обрабатывает нажатия клавиш)
WS_TABSTOP	Если текстовое поле должно входить в группу элементов управления, которым по очереди передается фокус ввода при нажатии клавиши табуляции

Параметр rect задает размер и расположение окна "текстового поля". Может быть объектом CRect или структурой RECT. Параметр pParentWnd задает родительское окно (обычно это CDialog). Значение этого параметра не должно быть равно NULL. Параметр nID задает идентификатор "текстового поля".

Выделение текста

Для того чтобы определить положение текущего выделения текста, у "текстового поля" есть следующие функции:

DWORD CEdit::**GetSel**( )

и

void CEdit::**GetSel**(int& nStartChar, int& nEndChar) —

первая версия функции возвращает начальную позицию выделенного текста в младшем слове и позицию первого невыделенного, следующего за выделенным текстом, символа в старшем слове. Вторая версия имеет параметры nStartChar и nEndChar, которые являются ссылками на переменные целого типа, принимающие позиции первого выделенного и первого невыделенного символов соответственно.

Для того чтобы изменить выделение текста, у "текстового поля" есть функции:

void CEdit::**SetSel**(
  DWORD dwSelection,
  BOOL bNoScroll = FALSE),

void CEdit::**SetSel**(
  int nStartChar,
  int nEndChar,
  BOOL bNoScroll = FALSE) —

dwSelection — в младшем слове этого параметра задается начальная позиция, в старшем слове — конечная позиция для выделения. Если в младшем слове содержится 0 и в старшем –1, то весь текст в "текстовом поле" будет выделен. Если в младшем слове содержится –1, то выделение текста в "текстовом поле", если оно есть, будет снято. bNoScroll — флаг, определяющий, должна ли быть каретка (маркер ввода с клавиатуры) установлена в зону видимости, возможно при помощи прокрутки текста в окне "текстового поля". Если он равен FALSE, то каретка устанавливается в зону видимости, если TRUE — то нет. nStartChar — задает начальную позицию: nStartChar = 0 и nEndChar = –1 — весь текст в "текстовом поле" будет выделен; nStartChar= –1 — все выделение текста в "текстовом поле" будет снято; nEndChar — задает конечную позицию.

void CEdit::**ReplaceSel**(LPCTSTR lpszNewText) —

замещает текущий выделенный текст в "текстовом поле" на заданный текст.

lpszNewText — указывает на строку, содержащую текст, на который производится замена. Если в момент вызова функции текст в "текстовом поле" не выделен, то замещающий текст будет вставлен, начиная с текущей позиции каретки. Этой функцией следует пользоваться для замены части текста, если нужно заменить весь текст в "текстовом поле", то лучше использовать функцию CWnd::SetWindowText.

Работа со строками

int CEdit::**GetFirstVisibleLine**( ) —

возвращает индекс самой верхней видимой строки в редакторе. Для однострочного редактора возвращается ноль.

void CEdit::**LineScroll**(
  int nLines,
  int nChars = 0) —

прокручивает текст в многострочном "текстовом поле". nLines — задает число строк, которые должны быть прокручены по вертикали; nChars — задает число позиций символов для прокрутки по горизонтали. Этот параметр игнорируется, когда "текстовое поле" имеет стили ES_RIGHT или ES_CENTER. Знак этих параметров указывает направление: отрицательные значения приводят к прокрутке вверх и влево, а положительные — вниз и вправо соответственно.

Данная функция работает только для многострочного "текстового поля". Она не прокручивает по вертикали текст дальше последней строки в "текстовом поле". Если сумма номера текущей строки и числа строк для прокрутки, заданного параметром nLines, превышает общее число строк в "текстовом поле", то значение устанавливается таким образом, что последняя строка текста устанавливается вверху окна "текстового поля".

Функция может быть использована для прокрутки текста по горизонтали за последний символ любой строки.

int CEdit::**LineFromChar**(int nIndex = −1) —

позволяет получить номер строки (параметр nIndex), содержащей заданный *символьный индекс*, где под символьным индексом понимается число символов от начала текста, содержащегося в "текстовом поле".

Если параметр nIndex равен –1, то возвращается номер строки, содержащей первый символ выделенного текста. Если же в "текстовом поле" нет выделенного текста, то в этом случае будет возвращен номер текущей строки (строки, на которой находится каретка). Эта функция также предназначена только для многострочного "текстового поля".

int CEdit::**LineLength**(int nLine = −1) —

определяет длину строки, содержащейся в "текстовом поле". Когда функция вызывается для многострочного "текстового поля", возвращается длина строки (в байтах), заданной параметром nLine. Когда функция вызывается для однострочного "текстового поля", возвращается длина текста (в байтах), содержащегося в "текстовом поле". Если параметр nLine равен –1, то возвращается длина текущей строки (содержащей каретку), не включая длину любого выделенного в строке текста. Когда функция вызывается для однострочного "текстового поля", этот параметр игнорируется.

Для получения номера строки в многострочном "текстовом поле" следует использовать функцию CEdit::LineIndex.

Для получения общего количества строк в редакторе следует воспользоваться функцией GetLineCount, которая возвращает единицу в случае, если "текстовое поле" пусто.

Следующие функции позволяют получить строку из "текстового поля":

int CEdit::**GetLine**(
  int nIndex,
  LPTSTR lpszBuffer)

и

int CEdit::**GetLine**(
  int nIndex,
  LPTSTR lpszBuffer,
  int nMaxLength) —

обе функции возвращают число реально скопированных байт текста и 0, если номер строки, заданный в nIndex, превышает количество строк в "текстовом поле". Параметр nIndex задает номер строки, которую необходимо получить из "текстового поля". Нумерация строк начинается с нуля. Параметр lpszBuffer указывает на буфер, в который будет скопирована строка, причем первое слово буфера должно содержать число, равное максимальному числу байт, которые можно скопировать в буфер. Вторая версия функции помещает в буфер это число самостоятельно на основании значения, заданного пара-

метром nMaxLength. Обратите внимание, что строка, скопированная в буфер, *не будет* содержать символ окончания строки.

Чтобы получить номер первого символа строки от начала буфера, следует воспользоваться функцией CEdit::LineIndex. Эта функция работает только в многострочном "текстовом поле".

Работа с блоками текста

void CEdit::**Clear**( ) —

удаляет выделенный текст (если он есть).

Для восстановления удаленного текста (откат назад) можно вызвать функцию-член CEdit::**Undo**. Для удаления текущего выделения в "текстовом поле" и одновременного помещения его буфер обмена следует воспользоваться функцией CEdit::Cut.

void CEdit::**Copy**( ) —

копирует выделенный текст (если он есть) в буфер обмена в формате CF_TEXT.

void CEdit::**Cut**( ) —

удаляет (вырезает) выделенный текст в "текстовом поле" (если он есть) и копирует удаленный текст в буфер обмена в формате CF_TEXT.

void CEdit::**Paste**( ) —

вставляет данные из буфера обмена в "текстовое поле" с текущей позиции курсора. Данные вставляются только в том случае, если буфер обмена содержит данные в формате CF_TEXT.

BOOL CEdit::**Undo**( ) —

отменяет последнюю операцию, произведенную в "текстовом поле" (откат на один шаг назад). Для однострочного "текстового поля" возвращаемое значение всегда TRUE. Для многострочного "текстового поля" возвращается TRUE в случае успешного завершения и FALSE, если операция отката завершилась неуспешно.

Над операцией отката также можно произвести операцию отката. Например, если некоторый текст в "текстовом поле" был удален, а затем восстановлен вызовом функции CEdit::Undo, то, вызвав еще раз CEdit::Undo, можно опять удалить текст.

BOOL CEdit::**CanUndo**( ) —

позволяет определить, может ли быть отменена последняя операция, совершенная в "текстовом поле", или нет. Значение TRUE, возвращенное функцией, означает, что последняя операция, совершенная в "текстовом поле", может быть отменена вызовом функции CEdit::Undo.

void CEdit::**EmptyUndoBuffer**( ) —

сбрасывает флаг отмены последней операции в "текстовом поле". В этом случае отмена последней выполненной операции становится невозможной. Флаг отмены последней

операции устанавливается всякий раз, когда действие операции внутри "текстового поля" может быть отменено. Флаг отмены последней операции автоматически сбрасывается всякий раз, когда вызываются функции CWnd::SetWindowText и CEdit::SetHandle.

Параметры и флаги

Для того чтобы определить, изменялся ли текст в "текстовом поле", имеется функция GetModify:

BOOL CEdit::**GetModify**( ) —

возвращает значение TRUE, если содержимое "текстового поля" изменилось, и FALSE — в противном случае.

Windows поддерживает внутренний флаг (флаг изменения), показывающий, менялось ли содержимое "текстового поля". Этот флаг сбрасывается, когда "текстовое поле" создается, он также может быть сброшен (или установлен) функцией SetModify:

void CEdit::**SetModify**(BOOL bModified = TRUE) —

устанавливает (bModified = TRUE) или сбрасывает (bModified = FALSE) флаг изменения.

Следующая функция позволяет установить или отменить режим "текстового поля" "только для чтения", при котором можно только просматривать текст в "текстовом поле".

Примечание

При этом цвет фона клиентской области окна становится таким же, как и цвет трехмерных объектов, например, кнопок. Однако в отличие от состояния, при котором элемент является недоступным (стиль WM_DISABLE), цвет символов текста тот же, что и в доступных для изменения "текстовых полях".

BOOL CEdit::**SetReadOnly**(BOOL bReadOnly = TRUE) —

возвращает значение TRUE в случае успешного установления требуемого режима и FALSE — в случае неудачи. Параметр bReadOnly устанавливает (TRUE) или снимает (FALSE) режим "только для чтения".

Текущее состояние можно узнать, проанализировав состояние флага ES_READONLY, вызвав функцию CWnd::GetStyle и проанализировав возвращенное ею значение.

void CEdit::**GetRect**(LPRECT lpRect) —

позволяет получить границы форматирования "текстового поля", которые представляют собой прямоугольник, размеры которого не зависят от размера окна "текстового поля".

Размеры прямоугольника, в пределах которого форматируется текст в многострочном "текстовом поле", могут быть изменены вызовом функций CEdit::SetRect и CEdit::SetRectNP.

void CEdit::**LimitText**(int nChars = 0) —

ограничивает длину текста, который пользователь может ввести в окне "текстового поля". Параметр nChars задает длину (в байтах) текста, который может ввести пользователь. Если параметр равен нулю, то длина текста устанавливается равной UINT_MAX (максимальное значение, которое может быть сохранено в переменной типа unsigned int). Это же значение задано по умолчанию.

Функция задает длину текста, который пользователь может ввести, но она не влияет на длину текста, уже находящегося в "текстовом поле". Также она не влияет на длину текста, копируемого функцией CWnd::SetWindowText.

Если приложение использует функцию SetWindowText, чтобы поместить в "текстовое поле" большее количество текста, чем задано функцией LimitText, пользователь сможет только удалять часть текста (изменение и удаление исключаются) в "текстовом поле".

Рекомендуется использовать функцию CEdit::SetLimitText.

При создании приложений, в которых пользователю требуется ввести с клавиатуры какой-либо пароль, могут быть полезны функции, позволяющие вместо вводимых с клавиатуры символов выводить на экран последовательность той же длины, состоящую из одного и того же символа (по умолчанию это символ "*"). Класс CEdit имеет в своем составе функцию для получения установленного в данный момент символа замены (GetPasswordChar) и функцию назначения такого символа (SetPasswordChar). Эти функции не действуют для многострочных "текстовых полей".

Дополнительные функции

Для определения и изменения полей в "текстовом поле" предназначены следующие две функции:

DWORD CEdit::**GetMargins**( ) —

возвращает значения границ полей "текстового поля" в пикселях, ширина левого края содержится в младшем слове, ширина правого — в старшем.

void CEdit::**SetMargins**(
UINT nLeft,
UINT nRight) —

устанавливает значения границ полей, параметры nLeft и nRight определяют новую ширину полей в пикселях.

Для определения и изменения максимального размера текста, который может содержаться в "текстовом поле", предназначены функции GetLimitText и SetLimitText.

Для определения номера символа и номера строки по указанным координатам предназначена функция CharFromPos:

int CEdit::**CharFromPos**(CPoint pt) —

номер символа возвращается в младшем слове, а номер строки — в старшем.

Следующая функция позволяет произвести обратную операцию: по номеру символа определить его координаты:

CPoint CEdit::**PosFromChar**(UINT nChar) —

если значение параметра nChar больше, чем количество символов, содержащихся в "текстовом поле", то будут возвращены координаты последнего символа.

Список

С помощью класса CListBox можно создать такой элемент управления Windows, как "список". Список представляет собой прямоугольник, внутри которого находится последовательность тестовых или пользовательских элементов, которые пользователь может пролистывать и выбирать. Список может использоваться, например, для просмотра имен файлов на диске.

Можно создавать списки, позволяющие выбирать только одну запись (элемент) или несколько одновременно (списки множественного выбора). Когда пользователь выбирает запись в списке, — она выделяется, и окно-список посылает родительскому окну извещение. Вообще, список посылает или, правильнее сказать, может посылать несколько извещений родительскому окну. Для них всех предусмотрены перечисленные ниже специальные макросы, позволяющие добавить в карту сообщений окна функции обработки. Функции, добавленные первыми тремя макросами, будут вызваны только для списков, содержащих текстовые строки и созданных со стилем (о которых чуть позже) LBS_NOTIFY. Большинство уведомлений посылается родительскому окну тогда и только тогда, когда список является текущим элементом управления.

ON_LBN_DBLCLK	Добавляет обработчик двойного щелчка кнопки мыши в области списка
ON_LBN_SELCANCEL	Добавляет обработчик снятия выбора с элементов списка
ON_LBN_SELCHANGE	Добавляет обработчик смены выбора элемента списка, однако он не вызывается при смене выбора функцией CListBox::SetCurSel; для списков множественного выбора обработчик вызывается при любом нажатии клавиш управления курсором, независимо от того, была ли смена выбора
ON_LBN_SETFOCUS	Добавляет обработчик получения списком фокуса ввода
ON_WM_CHARTOITEM	Добавляет обработчик уведомляющего сообщения для самоотображаемых списков (owner-draw), не имеющих строковых элементов при получении ими сообщения WM_CHAR

ON_WM_VKEYTOITEM	Добавляет обработчик уведомляющего сообщения для списков, имеющих стиль LBS_WANTKEYBOARDINPUT, при получении ими сообщения WM_KEYDOWN
ON_LBN_ERRSPACE	Добавляет обработчик уведомления о невозможности получить достаточно запрашиваемой памяти
ON_LBN_KILLFOCUS	Добавляет обработчик потери фокуса ввода списком

Окно-список можно создавать как в шаблоне блока диалога, так и непосредственно в коде программы.

Инициализация

Создание объекта CListBox и собственно списка Windows в коде программы требует создания (вызова) конструктора класса и функции Create:

BOOL CListBox::**Create**(
DWORD dwStyle,
const RECT & rect,
CWnd* pParentWnd,
UINT nID) —

создает окно-список и присоединяет его к объекту CListBox. Она возвращает значение TRUE, если завершилась успешно, и FALSE в противном случае.

Параметр *dwStyle* задает стиль списка, который может принимать комбинацию следующих значений:

LBS_EXTENDEDSEL	Пользователь может выбирать несколько записей одновременно, используя клавишу <Shift> и мышь или специальную комбинацию клавиш
LBS_HASSTRINGS	Задает список, который может содержать произвольные строки; приложение может заполнять строками список и извлекать из него строки при помощи функции CListBox::GetText
LBS_MULTICOLUMN	Определяет список, отображаемый в несколько столбцов, который может осуществлять горизонтальную прокрутку; ширина всех столбцов в пикселях устанавливается функцией CListBox::SetColumnWidth
LBS_MULTIPLESEL	Создает список множественного выбора, в котором при выборе элемента его состояние меняется (выделен — не выделен), не влияя на состояние других элементов
LBS_NOINTEGRALHEIGHT	Размер окна списка точно соответствует размеру, указанному приложением при его создании; иначе Windows подгоняет его размер, с тем чтобы показывалось целое число элементов
LBS_NOREDRAW	Изменения, производимые со списком, не отражаются автоматически в окне; этот стиль может быть изменен в любой момент посылкой сообщения WM_SETREDRAW
LBS_NOTIFY	Родительское окно будет получать извещения при возникновении ряда событий (см. выше)

LBS_OWNERDRAWFIXED	Отображение содержимого окна возлагается на функцию родительского окна, при этом элементы списка имеют одинаковую высоту
LBS_OWNERDRAWVARIABLE	Отображение содержимого окна возлагается на функцию родительского окна, при этом элементы списка могут иметь разную высоту
LBS_SORT	Строки списка сортируются по алфавиту
LBS_STANDARD	Строки списка сортируются по алфавиту и родительское окно получает извещение, если пользователь производит одиночный или двойной щелчок на строке списка; сам список окружен рамкой
LBS_USETABSTOPS	При отображении строковых элементов используются символы табуляции; по умолчанию позиция табуляции задается равной 32 диалоговым единицам; диалоговые единицы вычисляются как четверть базовой ширины, возвращаемой функцией ::GetDialogBaseUnits Win32 API
LBS_WANTKEYBOARDINPUT	Родительское окно списка получает сообщение WM_VKEYTOITEM или WM_CHARTOITEM, если пользователь нажимает клавишу в момент, когда список имеет фокус ввода
LBS_DISABLENOSCROLL	Список показывает скрытую вертикальную полосу прокрутки, даже когда список не имеет достаточного количества записей для прокрутки в окне

Кроме вышеперечисленных стилей, следует использовать следующие стили:

WS_CHILD	Следует указывать всегда при создании списка
WS_VISIBLE	Следует указывать, если список изначально должен быть видимым
WS_DISABLED	Следует указывать, если список изначально должен быть недоступным
WS_GROUP	Необходим для объединения списков в группу с другими элементами
WS_TABSTOP	Необходим, если кнопка должна входить в группу кнопок, которым по очереди передается фокус ввода при нажатии клавиши <Tab>
WS_VSCROLL	Необходим для добавления вертикальной полосы прокрутки
WS_HSCROLL	Необходим для добавления горизонтальной полосы прокрутки

Параметр rect задает размер и расположение списка на экране, параметр pParentWnd задает родительское окно (обычно объект CDialog), который *не может* быть равен NULL, а параметр nID задает идентификатор списка.

Изменение параметров списка

Не все параметры списка можно задать при помощи редактора ресурсов или функции Create. Далее приведены описания функций-членов класса CListBox, использование которых позволяет настраивать список в соответствии с требованиями приложения.

Для изменения ширины списка предназначены следующие две функции (обратите внимание, что речь идет не о размерах окна, ограничивающего список, а о поле списка, только часть которого может быть видима "через" это окно):

int CListBox::**GetHorizontalExtent**( ) —

возвращает ширину списка в пикселях, в пределах которой список может прокручиваться по горизонтали. Функцию можно использовать только для списков, имеющих горизонтальную полосу прокрутки.

void CListBox::**SetHorizontalExtent**(int cxExtent) —

устанавливает ширину списка по горизонтали (в пикселях), в пределах которой список может прокручиваться. Если размер окна списка меньше этого значения, то строки в списке можно будет прокручивать с помощью горизонтальной полосы прокрутки. Если окно списка такого же размера или больше, то горизонтальная полоса прокрутки делается невидимой. Для того чтобы список реагировал на SetHorizontalExtent, нужно, чтобы он был создан со стилем WS_HSCROLL. Для списков, имеющих несколько столбцов, вместо этой функции следует использовать функцию CListBox::SetColumnWidth.

Для получения размера прямоугольника, который ограничивает текущую изображаемую запись списка, предназначена функция:

int CListBox::**GetItemRect**(
int nIndex,
LPRECT lpRect) —

nIndex задает индекс записи, а lpRect является указателем на структуру RECT, возвращающую координаты прямоугольника, в который выводится запись списка. В случае ошибки возвращает LB_ERR.

Следующая пара функций позволяет получить и изменить высоту записей в списке:

int CListBox::**GetItemHeight**(int nIndex) —

возвращает высоту записи в списке в пикселях. Если список создан с использованием стиля LBS_OWNERDRAWVARIABLE, то функция возвращает высоту записи списка с указанным индексом (параметр nIndex). В противном случае возвращается LB_ERR.

int CListBox::**SetItemHeight**(
int nIndex,
UINT cyItemHeight) —

устанавливает высоту записи в списке, где nIndex задает индекс записи в списке. Этот параметр используется лишь в том случае, если список имеет стиль LBS_OWNERDRAWVARIABLE, иначе он должен быть равен нулю. Параметр cyItemHeight задает высоту записи в пикселях. Функция возвращает LB_ERR, если индекс или высота записи имеют неверное значение.

Для изменения ширины столбцов предназначена функция SetColumnWidth:

void CListBox::**SetColumnWidth**(int cxWidth) —

устанавливает ширину столбцов (в пикселях) в списке, имеющем несколько столбцов (созданном со стилем LBS_MULTICOLUMN), где cxWidth задает ширину в пикселях всех столбцов.

Имеется возможность задания позиций табуляции в списке. Для этого предназначены следующие функции:

void CListBox::**SetTabStops**( ) —

вариант функции, не имеющий аргументов, устанавливает позиции табуляции по умолчанию (в 32 диалоговые единицы).

BOOL CListBox::**SetTabStops**(const int& cxEachStop) —

устанавливает значение позиции табуляции, заданной параметром cxEachStop. Данное значение позиции табуляции устанавливается для всех одно и отсчитывается от предыдущей позиции.

Напротив, следующая функция позволяет указывать различные значения позиций для нескольких табуляций:

BOOL CListBox::**SetTabStops**(
int nTabStops,
LPINT rgTabStops) —

позиции табуляции будут установлены в соответствии с каждым элементом массива rgTabStops, размер которого задан параметром nTabStops. Значения позиций табуляций отсчитываются от левого края окна списка, таким образом все элементы массива как бы отсортированы по возрастанию. Для возможности использования функций SetTabStops список должен быть создан с использованием стиля LBS_USETABSTOPS.

Для того чтобы правильно обрабатывать строки, необходимо задание параметра, идентифицирующего локализацию. Для этого предназначен идентификатор локализации LCID, представляющий собой 32-разрядное значение, идентифицирующее язык локализации программы, которое определяет таблицы символов для хранения и отображения текстовых ресурсов, правила сортировки строк, раскладку клавиатуры, форматы даты и времени. Это необходимо, например, для правильной сортировки строк в списке. Для определения и изменения этого идентификатора служат следующие две функции:

LCID CListBox::**GetLocale**( ) —

возвращает идентификатор локализации для данного списка.

LCID CListBox::**SetLocale**(LCID nNewLocale) —

устанавливает идентификатор локализации для списка, где nNewLocale — новый локальный идентификатор для списка.

Если функция SetLocale не вызывалась, то значение LCID по умолчанию берется из операционной системы. Значение по умолчанию в системе может изменяться путем изменения установок с помощью программ настройки Windows (установка языка и национальных стандартов).

Функции для работы с содержимым списка

После того как список был создан, необходимо его заполнить. Следующие функции предназначены для работы со строками списка. Они позволяют выполнить такие основные операции, как добавление и удаление строк из

списка, вставку строк в определенное место в списке, поиск строк, удаление всех строк, а также заполнение списка имен файлов. К этой же группе относятся функции получения информации о конкретных элементах списка и сопоставления с ними дополнительных данных. Последняя возможность позволяет создавать списки объектов самых разных типов, возлагая при этом работу по отображению их описания на Windows, исключая таким образом необходимость создания функций отображения элементов, что нужно для самоотображаемых элементов управления.

Начнем с наиболее часто используемой функции, которая добавляет строку текста в качестве элемента в список:

Примечание

Все функции добавления элементов в список в случае ошибки возвращают значение LB_ERR, а если недостаточно места для хранения новой строки — LB_ERRSPACE. При этом может генерироваться соответствующее уведомление.

int CListBox::**AddString**(LPCTSTR lpszItem) —

возвращает индекс новой строки в списке. Параметр lpszItem указывает на завершающуюся нулевым символом строку, которая должна быть добавлена к списку.

Если список не имеет стиля LBS_SORT, то строка будет добавлена в конец списка. Если же этот стиль был задан при создании списка, то строка будет добавлена в список, и список будет заново отсортирован. Если же список имеет стиль LBS_SORT, но не имеет стиля LBS_HASSTRINGS, то список будет сортироваться путем многократного вызова функции CListBox::CompareItem, которую требуется переопределить, создав соответствующий производный класс.

Списки, не имеющие стиля LBS_SORT, могут использовать функцию InsertString для вставки строкового элемента в заданную позицию:

int CListBox::**InsertString**(
  int nIndex,
  LPCTSTR lpszItem) —

возвращает индекс новой строки в списке. Если список имеет стиль LBS_SORT, то значение параметра nIndex, который и определяет желаемую позицию строки, может не соответствовать возвращаемому значению функции; что вполне естественно. Если nIndex = –1, то строка добавляется в конец списка. Параметр lpszItem является указателем на оканчивающуюся нулевым символом строку, которая должна быть вставлена в список.

Очень часто в приложениях списки применяются для того, чтобы вывести список файлов на диске, для их просмотра и/или выбора.

Примечание

Приводим описание функции для лучшего понимания возможностей библиотеки классов MFC, да и Win32 API, т. к. сама библиотека и собственно Windows имеют средства отображения файлов и каталогов (папок).

Для того чтобы облегчить заполнение списка в этих случаях, существует функция Dir:

int CListBox::**Dir**(
UINT attr,
LPCTSTR lpszWildCard) —

заполняет список содержимым директориев. Параметр attr определяет, файлы с какими атрибутами будут добавлены в список. Он может быть любой комбинацией из следующих значений:

0x0000	Файлы, доступные для чтения и записи
0x0001	Файлы, доступные только для чтения
0x0002	Скрытые файлы
0x0004	Системные файлы
0x0010	Подкаталоги
0x0020	Файлы с установленным битом архивации
0x4000	Включать имена дисков
0x8000	Специальный флаг; если он установлен, то только файлы указанного типа (заданного в lpszWildCard) включаются в список; если не установлен, то файлы заданного типа включаются в дополнение к остальным файлам

Параметр lpszWildCard является указателем на строку, специфицирующую файлы. Строка может содержать символы "*" или "?", которые имеют тот же смысл, что принят в MS-DOS (например, *.cpp, *.c?). К сожалению, поддержка шаблонов, принятых в UNIX, не реализована.

Если необходимо добавить в список большое количество строк (например, больше 100), то лучше воспользоваться следующей функцией:

int CListBox::**InitStorage**(
int nItems, UINT nBytes) —

позволяет выделить память для хранения строк списка, определяемую с помощью значений параметров nItems (количество записей) и nBytes (размер памяти в байтах на один элемент). Эта функция позволяет ускорить инициализацию списка с большим количеством записей. При ее применении выполнение функций CListBox::AddString, CListBox::InsertString и CListBox::Dir занимает минимальное время. Если при вызове функции памяти будет запрошено меньше, чем нужно для размещения всех строк, то будет использовано обычное выделение памяти для записей, которым не хватило выделенного объема.

Как и любой список вообще, "список" — элемент управления Windows и вместе с ним класс CListBox требуют, наряду с функциями добавления/заполнения элементов, и функции удаления/очистки.

int CListBox::**DeleteString**(UINT nIndex) —

удаляет из списка элемент, определяемый индексом (параметр nIndex), и возвращает количество оставшихся элементов. Значение LB_ERR возвращается, если заданный индекс записи на удаление больше числа записей в списке.

void CListBox::**ResetContent**( ) —

удаляет все элементы из списка.

Следующие две функции предназначены для поиска строки в списке, совпадающей, начиная с первого символа, с заданной строкой.

int CListBox::**FindString**(
int nStartAfter,
LPCTSTR lpszItem) —

осуществляет только поиск строки в списке.

int CListBox::**SelectString**(
int nStartAfter,
LPCTSTR lpszItem) —

ищет и, кроме того, выделяет найденную строку, причем, если строка найдена, то она выделяется и, если нужно, список прокручивается с тем, чтобы строка стала видна.

Обе функции возвращают индекс найденной строки в списке или LB_ERR в случае неудачи. Параметр nStartAfter должен содержать индекс строки списка, находящейся перед строкой, с которой будет начат поиск. Если достигнут конец списка, то поиск будет продолжен с начала списка до строки, заданной в nStartAfter. Если значение параметра nStartAfter = −1, то поиск будет осуществляться по всему списку от его начала. Параметр lpszItem является указателем на оканчивающуюся нулевым символом строку, которая должна быть найдена, причем поиск идет без учета регистра.

Вторая из описанных функций не может использоваться для списков, созданных со стилем LBS_MULTIPLESEL.

Операции над элементами списка

Функции этой группы предназначены для получения информации о конкретных элементах списка или, наоборот, для выделения элементов списка и связывания с ними 32-разрядных значений и указателей.

Естественно, чтобы работать со списком любого вида, требуется знать, сколько элементов он содержит. Для этого в классе CListBox предназначена функция GetCount:

int CListBox::**GetCount**( ) —

возвращает либо количество элементов в списке, либо LB_ERR в случае ошибки.

Для получения информации о том, выбран ли определенный элемент или нет, следует использовать функцию GetSel, которая может работать как со списками, позволяющими выбирать одну запись (списки одиночного выбора), так и со списками, позволяющими выбирать сразу несколько записей (списки множественного выбора).

Для получения индекса текущей выбранной записи и, наоборот, задания конкретной записи в качестве выбранной следует использовать функции GetCurSel, SetCurSel, GetSelItems, SetSel.

Примечание

Выбор элемента приведет при необходимости к прокручиванию списка таким образом, чтобы выбранный элемент стал видимым в окне списка.

Первые две функции предназначены, прежде всего, для списков одиночного выбора, а вторые две — для списков множественного выбора. Однако нет никаких ограничений по использованию функций со стороны библиотеки классов и Windows, и не удивляйтесь, если функции будут вести себя неадекватно ситуации. Например, функция GetCurSel возвращает значение LB_ERR как в случае отсутствия выбранного элемента, так и в случае ее вызова для списка множественного выбора, независимо от того, есть фактически выбранные элементы или нет. Наиболее интересная из перечисленных — функция GetSelItems:

int CListBox::**GetSelItems**(
    int nMaxItems,
    LPINT rgIndex) —

позволяет получить индексы строк, которые выбраны в данный момент в списке, и это значение может быть больше, чем размерность буфера (параметр rgIndex), заданная параметром nMaxItems. Причем, при использовании этой функции для списка одиночного выбора она возвратит значение LB_ERR.

В добавление к описанным функциям, класс CListBox содержит функцию (SetItemRange), позволяющую выбирать несколько записей, следующих подряд.

Для получения содержимого текстовых элементов списка класс CListBox предусматривает целый набор функций. Во-первых, для надежных программ, которые не полагаются на предположение о длине запрашиваемых данных, существует функция GetTextLen, которая возвращает длину строкового элемента в байтах, исключая завершающий нуль-символ. Во-вторых, для получения собственно текста следует использовать функцию GetText, предоставляющую буфер для копирования в него символов.

Примечание

Эта функция имеет версию, где в качестве приемника указывается ссылка на объект класса CString. Использование этой функции исключает необходимость самостоятельного определения размера строкового элемента, выделения памяти под принимающий буфер, а затем и освобождения памяти.

С первого взгляда может показаться, что элемент управления "список" предназначен, в основном, для текстовых элементов или для визуализации в

виде списка произвольных данных, скорее, временного использования. В отличие от чисто "программных" классов-контейнеров, описываемые списки отличаются в сторону визуализации и обладают меньшей гибкостью, требуемой для хранения объектов самых разных типов. Действительно, элемент управления "список" существует только тогда, когда он виден или, по крайней мере, может быть немедленно отображен.

Тем не менее, и описываемые списки имеют средства для сопоставления элементов списка с любыми объектами, как пользовательскими, так и системными. Для этого списки поддерживают возможность хранения 32-разрядных чисел или указателей для каждого элемента списка. Причем даже в случае изменения индекса элемента, что может происходить вследствие добавления, удаления и пересортировки списка, значение, связанное с конкретным элементом, будет связано именно с ним, независимо от индекса элемента в списке.

Перечислим функции, реализующие возможность как связывания чисел (указателей) с конкретным элементом, так и обратной операции: GetItemData, SetItemData, GetItemDataPtr, SetItemDataPtr.

В ряде случаев бывает необходимо определить, какой из элементов является видимым. Для определения и изменения первой видимой записи списка предназначены функции GetTopIndex и SetTopIndex.

И последняя функция этой группы, позволяющая определить, около какого элемента списка находится курсор мыши:

UINT CListBox::**ItemFromPoint**(
CPoint pt,
BOOL & bOutside) —

возвращает индекс записи списка, ближайшей к указанной точке, заданной параметром pt, причем координаты указаны относительно левого верхнего угла клиентской области окна списка. Параметр bOutside — ссылка на переменную типа BOOL, которая по завершении функции будет установлена в TRUE, если указанная точка лежит вне, и в FALSE, если точка находится внутри клиентской области окна списка.

Переопределяемые функции

Если создается список, внешний вид которого определяется программистом (списки, имеющие стили LBS_OWNERDRAWFIXED или LBS_OWNERDRAWVARIABLE), то *необходимо* переопределить две описанные ниже функции.

Функция перерисовки элементов списка:

virtual void CListBox::**DrawItem**(
LPDRAWITEMSTRUCT lpDrawItemStruct) —

вызывается библиотекой, если необходима перерисовка элемента самоотображаемого списка, внешний вид которого определяется пользователем. Параметр lpDrawItemStruct

является указателем на структуру DRAWITEMSTRUCT, которая содержит информацию о типе требуемой перерисовки.

Рассматривая класс CButton, мы уже упоминали эту структуру. Пришло время рассмотреть ее подробнее.

Структура DRAWITEMSTRUCT имеет следующее определение:

```
typedef struct tagDRAWITEMSTRUCT {
  UINT CtlType;
  UINT CtlID;
  UINT itemID;
  UINT itemAction;
  UINT itemState;
  HWND hwndItem;
  HDC hDC;
  RECT rcItem;
  DWORD itemData;
} DRAWITEMSTRUCT;
```

CtlType — тип самоотображаемого элемента управления, внешний вид которого определяется пользователем:

ODT_BUTTON	Кнопка
ODT_COMBOBOX	Комбинированный список
ODT_LISTBOX	Список
ODT_MENU	Меню

CtlID — идентификатор кнопки, списка или комбинированного списка. Для меню этот параметр не используется. itemID — идентификатор пункта меню или индекс записи в списке или комбинированном списке. Для списка или комбинированного списка, не содержащих записей, этот параметр должен иметь отрицательное значение, в результате чего будет выведен прямоугольник с координатами, заданными в rcItem, и не содержащий никаких элементов. itemAction — определяет, как должна осуществляться перерисовка. Он может быть комбинацией следующих битов:

ODA_DRAWENTIRE	Весь элемент управления требует перерисовки
ODA_FOCUS	Устанавливается, если элемент управления получает или теряет фокус ввода; поле itemState должно быть проверено для определения, имеет ли элемент управления фокус ввода
ODA_SELECT	Устанавливается, если сделан выбор; поле itemState должно быть использовано для определения нового выбора

itemState — задает состояние записи после перерисовки:

ODS_DISABLED	Запись недоступна
ODS_FOCUS	Запись имеет фокус ввода
ODS_SELECTED	Запись выбрана

hwndItem — дескриптор окна элемента управления. hDC — контекст устройства вывода. rcItem — прямоугольник, задающий границы перерисовки. itemData — для списка или комбинированного списка этот параметр содержит значение, установленное функциями CComboBox::AddString, CComboBox::InsertString, CListBox::AddString или CListBox::InsertString, а для меню — функциями CMenu::AppendMenu, CMenu::InsertMenu или CMenu::ModifyMenu.

Функция определения размеров элементов списка:

virtual void CListBox::**MeasureItem**(
LPMEASUREITEMSTRUCT lpMeasureItemStruct) —

вызывается при создании списка, внешний вид которого определяется пользователем, для определения размеров элемента списка. Параметр lpMeasureItemStruct является указателем на структуру MEASUREITEMSTRUCT.

По умолчанию эта функция не производит никаких действий. Необходимо переопределить ее, заполнив структуру типа MEASUREITEMSTRUCT, чтобы ОС Windows могла определить размеры списка. Если список создан со стилем LBS_OWNERDRAWVARIABLE, то эта функция вызывается для каждой записи списка. Если нет, то функция будет вызвана всего один раз.

Структура MEASUREITEMSTRUCT имеет следующий вид:

```
typedef struct tagMEASUREITEMSTRUCT {
  UINT CtlType;
  UINT CtlID;
  UINT itemID;
  UINT itemWidth;
  UINT itemHeight;
  DWORD itemData
} MEASUREITEMSTRUCT;
```

CtlType — содержит информацию о типе элемента управления (см. описание структуры DRAWITEMSTRUCT). CtlID — идентификатор элемента управления. itemID — идентификатор элемента в списке (для списка и комбинированного списка). Не используется для списков и комбинированных списков с фиксированной высотой элементов. itemWidth — используется для задания ширины элемента в меню (для списков и комбинированных списков не используется). itemHeight — задает высоту записи в списке (максимальное значение 255). itemData — назначение этого поля аналогично одноименному полю структуры DRAWITEMSTRUCT.

Функция, вызываемая при сортировке списков, не имеющих стиля LBS_HASSTRINGS и имеющих стиль LBS_SORT, обязательно должна быть переопределена:

virtual int CListBox::**CompareItem**(
LPCOMPAREITEMSTRUCT lpCompareItemStruct) —

вызывается библиотекой для определения расположения нового элемента в списке, внешний вид которого определяется пользователем и который должен осуществлять сортировку записей (список имеет стиль LBS_SORT).

Параметр lpCompareItemStruct является указателем на структуру COMPAREITEMSTRUCT, в которой определяется относительное положение двух сравниваемых элементов списка. Функция возвращает следующие значения:

–1	Первый элемент меньше второго
0	Первый элемент равен второму
1	Первый элемент больше второго

Пример самоотображаемого списка

Рассмотрим пример приложения диалогового типа, которое содержит один элемент управления — "список". Причем этот список создан со стилями LBS_OWNERDRAWVARIABLE и LBS_NOINTEGRALHEIGHT. Для реализации списка, имеющего различные размеры по вертикали для каждого элемента, создается объект класса CListBox, у которого реализованы две функции: DrawItem и MeasureItem. После части исходных текстов приведен пример окна приложения (рис. 23.3).

Рис. 23.3. Окно приложения ListBox

Обратите внимание на то, что функция MeasureItem "ничего не делает" из-за того, что настройка высоты элемента осуществляется непосредственно перед выводом элемента списка на экран. Заранее получить высоту элементов невозможно, т. к. окно приложения и сам список могут изменять размеры по желанию пользователя.

```
//Файл LBDlg.cpp
BOOL CLBDlg::OnInitDialog()
{
  CDialog::OnInitDialog();
  CRect rect;
  GetClientRect(&rect);
  m_List.MoveWindow(&rect);
  // Настраиваем начальные размеры для всех элементов списка
  // (в нашем примере это делать необязательно)
```

```
  m_List.SetItemHeight(0, 36);
  // Добавляем три элемента в список
  m_List.AddString(_T("Короткий текст"));
  m_List.AddString(_T("Средний по длине текст\в две строки"));
  m_List.AddString(_T("Длинная строка, для отображения которой
↳необходимо несколько строк"));
return TRUE;  // return TRUE  unless you set
                the focus to a control
}
void CLBDlg::OnSize(UINT nType, int cx, int cy)
{
  CDialog::OnSize(nType, cx, cy);
  // Настраиваем размеры списка по
  // размерам окна тогда, когда
  // объект-список уже есть в системе
  if (::IsWindow(m_List.m_hWnd))
  {
    CRect rect(0, 0, cx, cy);
    m_List.MoveWindow(&rect);
    m_List.InvalidateRect(&rect);
  }
}
//Файл VListBox.cpp
// Минимальная толщина обрамления значка в элементе списка
#define FRAME  2
// CVListBox
CVListBox::CVListBox(){}
CVListBox::~CVListBox(){}
void CVListBox::DrawItem(LPDRAWITEMSTRUCT lpDrawItemStruct)
{
  CDC dc;
  dc.Attach(lpDrawItemStruct->hDC);
  CRect rect = lpDrawItemStruct->rcItem;
  // Вычисляем смещение для вывода текста (32 пикселя — ширина значка)
  rect.left += 32 + FRAME + FRAME;
  // Вычисляем требуемую высоту области вывода текста
  int nHeight = dc.DrawText((LPCTSTR)lpDrawItemStruct->itemData,
    rect, DT_CALCRECT | DT_LEFT | DT_WORDBREAK) + FRAME + FRAME;
  rect.bottom = rect.top + nHeight;
```

```
  if (nHeight < 32 + FRAME + FRAME)
    rect.bottom = rect.top + 32 + FRAME + FRAME;
  // Изменяем высоту элемента управления
  SetItemHeight(lpDrawItemStruct->itemID, rect.Height());
  // Поле структуры необходимо изменить самостоятельно
  lpDrawItemStruct->rcItem.bottom = rect.bottom;
  rect = lpDrawItemStruct->rcItem;
  // Определяем состояние элемента списка
  if (lpDrawItemStruct->itemState & ODS_SELECTED)
  {
    // Рисуем "утопленную" кнопку
    dc.DrawFrameControl(&rect, DFC_BUTTON,
      DFCS_BUTTONPUSH | DFCS_PUSHED);
  }
  else
  {
    // Рисуем "выпуклую" кнопку
    dc.DrawFrameControl(&rect, DFC_BUTTON, DFCS_BUTTONPUSH);
  }
  // Рисуем значок, центрируя его по вертикали
  rect.top += (rect.Height() — 32) / 2;
  dc.DrawIcon(rect.left+FRAME, rect.top,
    AfxGetApp()->LoadIcon(IDR_MAINFRAME));
  rect = lpDrawItemStruct->rcItem;
  rect.left += 32 + FRAME;
  rect.DeflateRect(FRAME, FRAME);
  rect.top += (rect.Height() — (nHeight — FRAME)) / 2;
  // Выводим текст элемента управления
  dc.SetBkMode(TRANSPARENT);
  dc.DrawText((LPCTSTR)lpDrawItemStruct->itemData, rect,
    DT_LEFT | DT_WORDBREAK);

  dc.Detach();
}
// Эту функцию необходимо было переопределить, но
// она вызывается только при добавлении элемента в список,
// а в нашем примере высота элемента настраивается при каждом
// отображении. Следовательно, функция ничего не делает
void CVListBox::MeasureItem(
  LPMEASUREITEMSTRUCT lpMeasureItemStruct){}
```

Комбинированный список

Ранее были рассмотрены классы CEdit и CListBox, позволяющие создавать элементы управления "текстовое поле" и "список", но они настолько часто применяются в паре друг с другом, что в Windows был создан элемент управления, сочетающий свойства обоих этих элементов. Он представляет собой комбинацию окна списка с окном текстового поля и называется "комбинированным списком" (COMBO BOX). Часть комбинированного списка — непосредственно список — может выводиться на экран все время (стиль CBS_SIMPLE) или может как бы раскрываться из текстового поля (стили CBS_DROPDOWN и CBS_DROPDOWNLIST). Текущая выбранная запись списка выводится в прямоугольном окне, которое представляет собой окно простейшего текстового поля, и может быть как доступна (стили CBS_SIMPLE и CBS_DROPDOWN), так и недоступна (стиль CBS_DROPDOWNLIST) для редактирования. Если к тому же комбинированный список определен как раскрывающийся, то при нажатии символьной клавиши на клавиатуре строка, начинающаяся с данного символа, будет выделена.

Если вы хотите обрабатывать уведомляющие сообщения Windows, посылаемые комбинированным списком своему родительскому окну (обычно это объект класса, производного от CDialog), то необходимо добавить в карту сообщений этого окна соответствующий обработчик. Далее перечислены макросы, с помощью которых и производится назначение функций для обработки сообщений:

ON_CBN_CLOSEUP	Окно списка в комбинированном списке закрыто; это сообщение не посылается для комбинированного списка, имеющего стиль CBS_SIMPLE
ON_CBN_DBLCLK	Пользователь сделал двойной щелчок на строке списка в комбинированном списке; это сообщение посылается только для комбинированного списка, имеющего стиль CBS_SIMPLE; для комбинированного списка со стилями CBS_DROPDOWN или CBS_DROPDOWNLIST двойной щелчок не имеет смысла, т. к. уже одиночный щелчок приводит к сворачиванию списка
ON_CBN_DROPDOWN	список комбинированного списка развернут; это сообщение может быть послано комбинированным списком, имеющим стиль CBS_DROPDOWN или CBS_DROPDOWNLIST
ON_CBN_EDITCHANGE	Пользователь совершил действия, которые привели к изменению текста в текстовом поле комбинированного списка; в отличие от сообщения CBN_EDITUPDATE это сообщение посылается после того, как Windows обновила экран; это сообщение не посылается, если комбинированный список имеет стиль CBS_DROPDOWNLIST

ON_CBN_EDITUPDATE	Текстовое поле комбинированного блока "собирается" вывести на экран измененный текст; сообщение посылается после того, как текст был отформатирован, но перед тем, как он будет выведен на экран; сообщение не посылается, если комбинированный список имеет стиль CBS_DROPDOWNLIST
ON_CBN_ERRSPACE	Комбинированный список не может получить достаточно памяти для выполнения запроса
ON_CBN_SELENDCANCEL	Показывает, что выбор, сделанный пользователем, должен быть отменен; пользователь щелкнул на записи блока и затем на другом окне или элементе управления; сообщение посылается до сообщения CBN_CLOSEUP для уведомления, что выбор пользователя должен быть игнорирован
ON_CBN_SELENDOK	Пользователь выбрал строку списка и затем нажал клавишу <Enter> или нажал кнопку комбинированного списка (со стрелкой вниз), чтобы свернуть список; сообщение посылается до сообщения CBN_CLOSEUP для того, чтобы выбор строк пользователем был учтен; сообщения CBN_SELENDCANCEL и CBN_SELENDOK посылаются даже в случае, если сообщение CBN_CLOSEUP не посылается
ON_CBN_SELCHANGE	Выбор в списке комбинированного списка будет изменен в результате щелчка на списке или нажатия клавиш перемещения курсора; при обработке этого сообщения текст из комбинированного списка может быть получен только с помощью функции CComboBox::GetLBTex; функцию GetWindowText использовать нельзя
ON_CBN_SETFOCUS	Комбинированный список получил фокус ввода
ON_CBN_ERRSPACE	Элемент управления не может получить достаточно памяти для размещения текста или элементов списка

Инициализация

Создание объекта CComboBox и собственно комбинированного списка Windows в коде программы требует вызова конструктора класса и функции Create:

BOOL CComboBox::**Create**(
 DWORD dwStyle,
 const RECT & rect,
 CWnd* pParentWnd,
 UINT nID) —

создает комбинированный список и связывает его с объектом CComboBox. Функция возвращает значение TRUE в случае успешного завершения. Параметр dwStyle задает стиль комбинированного списка, который может принимать комбинацию следующих значений:

CBS_AUTOHSCROLL	Текст в текстовом поле автоматически прокручивается по горизонтали вправо и влево, если пользователь набирает символы в конце строки; если этот стиль не задан, то невозможно ввести текст, длина которого превышает размеры окна текстового поля
CBS_SIMPLE	Список изображается на экране все время, текущий выделенный элемент списка отображается в текстовом поле; редактирование разрешено
CBS_DROPDOWN	Подобен стилю CBS_SIMPLE за исключением того, что список не изображается до тех пор, пока пользователь не нажмет кнопку со стрелкой вниз, расположенную справа от текстового поля комбинированного списка. При этом список раскрывается из текстового поля (раскрывающийся список); редактирование разрешено
CBS_DROPDOWNLIST	Подобен CBS_DROPDOWN, но вместо строки текстового поля имеется строка со статическим текстом, который отображает текущий выбор, но редактировать его нельзя
CBS_HASSTRINGS	Комбинированный список, определяемый пользователем; на него накладывается ограничение: строки, помещаемые в список, должны быть текстовыми; комбинированный список выделяет память и хранит указатели на строки; приложение может использовать функцию-член CComboBox::GetText для получения текста текущего выбора
CBS_OEMCONVERT	Набор символов текста, вводимого в текстовом поле, конвертируется из ANSI в OEM и затем, при передаче данных из комбинированного списка, — обратно в ANSI; это обеспечивает правильное преобразование символов, когда приложение вызывает функцию Win32 API ::AnsiToOem для преобразования строки символов ANSI в строку символов OEM; эта функция может быть полезна для комбинированных списков, содержащих имена файлов, и применима только в случае, если используются стили CBS_SIMPLE или CBS_DROPDOWN
CBS_OWNERDRAWFIXED	Разработчик приложения сам отвечает за перерисовку содержимого элемента управления, при этом все элементы списка имеют одинаковую высоту
CBS_OWNERDRAWVARIABLE	Разработчик приложения сам отвечает за перерисовку содержимого элемента управления, при этом все элементы списка могут иметь различную высоту
CBS_SORT	Строки, вводимые в список, сортируются автоматически
CBS_DISABLENOSCROLL	При установке этого стиля список выводит на экран полосу вертикальной прокрутки даже если нет достаточного количества элементов для прокрутки в окне (в этом случае полоса прокрутки видна, но недоступна для использования)
CBS_NOINTEGRALHEIGHT	Задание этого стиля требует, чтобы размер комбинированного списка точно соответствовал заданному в приложении при его создании; обычно Windows изменяет размер комбинированного списка так, чтобы в его окне были видны элементы списка целиком

Параметр rect указывает расположение и размер комбинированного списка. Может быть структурой RECT или объектом класса CRect. Параметр pParentWnd задает родительское окно (обычно объект CDialog), и *не может* быть равен NULL, параметр nID задает идентификатор комбинированного списка.

Основные операции с комбинированным списком

Поскольку комбинированный список представляет собой комбинацию списка и текстового поля, он имеет как функции, аналогичные функциям его составных частей (списка и текстового поля), так и собственные.

Многие функции целиком аналогичны по параметрам и действию соответствующим функциям классов CEdit и CListBox, во избежание повторения приведем здесь только их список и краткое описание.

В классе определены функции, аналогичные функциям классов CEdit и CListBox. Кроме того, комбинированный список имеет и свои специфические функции.

Для управления раскрывающимся списком предназначена следующая группа функций:

void CComboBox::**ShowDropDown**(BOOL bShowIt = TRUE) —

делает видимым или невидимым список в комбинированном списке, имеющем стиль CBS_DROPDOWN или CBS_DROPDOWNLIST.

BOOL CComboBox::**GetDroppedState**( ) —

определяет, видим или нет раскрывающийся список в комбинированном списке.

Следующие две функции позволяют управлять шириной раскрывающегося списка при помощи получения/задания ее минимального допустимого значения. По умолчанию минимальная допустимая ширина списка равна 0. Ширина списка определяется максимальным значением из минимальной допустимой и заданной шириной комбинированного списка. Обе функции применимы лишь для комбинированных списков со стилями CBS_DROPDOWN или CBS_DROPDOWNLIST.

int CComboBox::**GetDroppedWidth**( ) —

возвращает минимальную допустимую ширину раскрывающегося списка в комбинированном списке.

int CComboBox::**SetDroppedWidth**(UINT nWidth) —

устанавливает минимальную допустимую ширину раскрывающегося списка в комбинированном списке.

void CComboBox::**GetDroppedControlRect**(LPRECT lprect) —

возвращает экранные координаты видимого раскрывающегося списка. lprect — указатель на структуру RECT, которая получит запрашиваемые координаты

Для выделения символов в окне редактирования комбинированного списка предназначена функция:

BOOL CComboBox::**SetEditSel**(
int nStartChar,
int nEndChar)

nStartChar — задает начальную позицию (если задано значение –1, то любое существующее выделение будет снято); nEndChar — задает конечную позицию (если задано значение –1, то весь текст от начальной позиции до конца будет выделен; конечная позиция задает положение первого невыделенного символа после выделения); в случае неудачного завершения функция возвращает CB_ERR, значение CB_ERR будет также возвращено, если комбинированный список имеет стиль CBS_DROPDOWNLIST или не имеет элементов.

int CComboBox::**GetLBText**(
int nIndex,
LPCTSTR lpszText)

и

void CComboBox::**GetLBText**(
int nIndex,
CString & rString) —

возвращают строку из списка комбинированного списка. Если параметр nIndex задает несуществующий номер элемента, то возвращается CB_ERR; lpszText — указатель на буфер, получающий строку (буфер должен иметь достаточно места для размещения строки и символа конца строки); rString — ссылка на объект CString (вторая версия функции заполняет этот объект текстом полученной строки).

Для определения длины строки (в байтах) из комбинированного списка используется функция:

int CComboBox::**GetLBTextLen**(int nIndex)

Для раскрывающегося списка могут использоваться два типа интерфейса: стандартный и расширенный.

BOOL CComboBox::**GetExtendedUI**( ) —

определяет, какой интерфейс используется (стандартный или расширенный). Значение TRUE означает расширенный интерфейс.

Расширенный интерфейс имеет следующие отличия:

- Щелчок на статическом тексте списка раскрывает список (только для комбинированных списков, имеющих стиль CBS_DROPDOWNLIST)
- Нажатие клавиши на клавиатуре раскрывает список

int CComboBox::**SetExtendedUI**(BOOL bExtended = TRUE) —

задает, должен ли использоваться интерфейс по умолчанию или расширенный интерфейс пользователя для комбинированных списков со стилями CBS_DROPDOWN или CBS_DROPDOWNLIST, параметр bExtended задает интерфейс: TRUE определяет расширенный интерфейс, FALSE — стандартный.

Операции со строками и виртуальные функции

Функции работы со строками в списке комбинированного списка и виртуальные функции целиком аналогичны соответствующим функциям класса CListBox (см. описание класса), поэтому ниже приводится только их список.

int CComboBox::**AddString**(LPCTSTR lpszString) —

добавляет строку в список.

int CComboBox::**DeleteString**(UINT nIndex) —

удаляет строку из списка.

int CComboBox::**InsertString**(
int nIndex, LPCTSTR lpszString) —

вставляет строку в список.

void CComboBox::**ResetContent**( ) —

удаляет все элементы списка.

int CComboBox::**FindString**(
int nStartAfter, LPCTSTR lpszString) —

осуществляет поиск строки.

int CComboBox::**FindStringExact**(
int nIndexStart, LPCTSTR lpszFind) —

ищет строку, целиком совпадающую с заданной.

int CComboBox::**SelectString**(
int nStartAfter, LPCTSTR lpszString) —

выбирает строку.

virtual void CComboBox::**DrawItem**(LPDRAWITEMSTRUCT
lpDrawItemStruct) —

вызывается при выводе на экран комбинированного списка, внешний вид которого определяется пользователем.

virtual void CComboBox::**MeasureItem**(
LPMEASUREITEMSTRUCT lpMeasureItemStruct) —

вызывается для определения размеров строки в комбинированном списке, внешний вид которого определяется пользователем.

Полоса прокрутки

Рассмотрим еще один элемент управления — полосу прокрутки (SCROLL BAR). В Windows есть два варианта использования полос прокрутки. Первый — когда полоса прокрутки функционирует как часть обычного окна или блока диалога. Второй — когда она может действовать, как любой другой элемент управления Windows. И хотя в обоих случаях управление полосой прокрутки происходит одинаково, все же использование ее как самостоятельного элемента управления требует применения некоторых специальных приемов.

Полосы прокрутки бывают вертикальные и горизонтальные (рис. 23.4), каждая полоса прокрутки имеет кнопки со стрелками, нажимая на которые можно управлять полосой прокрутки, а также ползунок — передвигая его с помощью мыши также можно управлять полосой прокрутки.

Рис. 23.4. Полосы прокрутки

В отличие от других элементов управления, полосы прокрутки посылают уведомления о различных событиях и сменах состояния в рамках одного сообщения. Точнее, таких сообщений два — по одному для горизонтальной и вертикальной полос прокрутки. Чтобы назначить свои собственные обработчики сообщений полос прокрутки, следует при помощи средства ClassWizard создать функции OnHScroll и OnVScroll для класса, определяющего родительское окно. Мы приводим описание этих функций для класса CWnd, а не для некоторого пользовательского класса окна. Дело в том, что, действительно, класс CWnd, как и некоторые другие оконные классы библиотеки MFC, содержит соответствующие обработчики. Однако эти функции — не виртуальные. Сделано так для того, чтобы исключить влияние со стороны разработчика приложения в случае, если возникнет необходимость вызова этих функций базового класса для обработки именно тех данных, которые передает элемент управления при уведомлении.

Реализовать это можно, повторно запросив параметры сообщения при помощи функции CWnd::GetCurrentMessage.

Для обработки сообщений от горизонтальной полосы прокрутки вызывается функция OnHScroll:

void CWnd::**OnHScroll**(
UINT nSBCode, UINT nPos, CScrollBar* pScrollBar) —

параметр nSBCode содержит код события и может принимать одно из следующих значений:

SB_LEFT	Прокрутка до самой левой границы
SB_ENDSCROLL	Завершение прокрутки, т. е. взаимодействия пользователя с ползунком
SB_LINELEFT	Прокрутка на одну позицию влево
SB_LINERIGHT	Прокрутка на одну позицию вправо
SB_PAGELEFT	Поблочная прокрутка влево
SB_PAGERIGHT	Поблочная прокрутка вправо
SB_RIGHT	Прокрутка до самой правой границы
SB_THUMBPOSITION	Прокрутка до позиции, определяемой параметром nPos
SB_THUMBTRACK	Прокрутка осуществляется ползунком, текущее положение которого определяется параметром nPos

Параметр nPos определяет текущее положение ползунка для кодов прокрутки SB_THUMBPOSITION или SB_THUMBTRACK, иначе не используется. Так как положение ползунка определяется заданными пределами, параметр nPos может содержать отрицательные значения, следовательно, при необходимости следует осуществлять преобразование типов от UINT (беззнаковое целое) к int (знаковое целое).

Параметр pScrollBar содержит указатель на (возможно) временный объект, в случае, если сообщение пришло не от оконной полосы прокрутки, наличие которой определяется стилями окна. В этом случае параметр равен NULL.

Тип прокрутки определяется установками полосы прокрутки и методом, каким осуществляется прокрутка. Поблочная прокрутка осуществляется при нажатии кнопки мыши в момент, когда курсор мыши находится в области полосы вне кнопок прокрутки или ползунка. Прокрутка на одну позицию осуществляется при нажатии одной из кнопок прокрутки.

Примечание

Дословный перевод этих терминов следующий: позиция называется *строкой*, а блок — страницей, что обычно действительно соответствует реализации. Однако для горизонтальной прокрутки сложно говорить о строках, да и о страницах тоже.

В случае обработки сообщений от полосы прокрутки без вызова обработчика по умолчанию (CWnd::Default) или той же функции базового класса, можно самостоятельно установить новое положение при помощи функции CWnd::SetScrollPos или CScrollBar::SetScrollBar.

Для обработки сообщений от вертикальной полосы прокрутки вызывается функция OnVScroll:

void CWnd::**OnVScroll**(
UINT nSBCode, UINT nPos, CScrollBar* pScrollBar) —

параметр nSBCode содержит код события и может принимать одно из следующих значений:

SB_BOTTOM	Прокрутка до самой нижней границы
SB_ENDSCROLL	Завершение прокрутки, т. е. взаимодействия пользователя с ползунком
SB_LINEDOWN	Прокрутка на одну позицию вниз
SB_LINEUP	Прокрутка на одну позицию вверх
SB_PAGEDOWN	Поблочная прокрутка вниз
SB_PAGEUP	Поблочная прокрутка вверх
SB_TOP	Прокрутка до самой верхней границы
SB_THUMBTRACK	Прокрутка до позиции, определяемой параметром nPos
SB_THUMBPOSITION	Прокрутка осуществляется ползунком, текущее положение которого определяется параметром nPos

Остальные параметры и требования к использованию этой функции аналогичны описанию функции CWnd::OnHScroll.

Помимо собственно прокрутки (которая может и не производиться), приложения могут обрабатывать код сообщения SB_THUMBTRACK для выполнения каких-либо дополнительных действий. Например Microsoft Word версии 7.0 при перемещении ползунка вертикальной полосы прокрутки выводит номер страницы, а в версии 8.0 дополнительно к номеру страницы также выводится текст текущего (для этой страницы) заголовка.

Входящий в состав библиотеки MFC класс CScrollBar позволяет создать полосу прокрутки и имеет в своем составе необходимые функции для управления полосой прокрутки.

Создание

Создание объекта CScrollBar и собственно полосы прокрутки Windows в коде программы требует вызова конструктора класса и функции Create:

```
BOOL CScrollBar::Create(
    DWORD dwStyle,
    const RECT& rect,
    CWnd* pParentWnd,
    UINT nID) —
```

параметр dwStyle задает стиль полосы прокрутки, который может содержать комбинацию следующих значений:

SBS_BOTTOMALIGN	Используется со стилем SBS_HORZ; нижний край полосы прокрутки выравнивается по нижнему краю прямоугольника, указанного в параметре rect
SBS_HORZ	Назначает горизонтальную полосу прокрутки; если не задан ни один из стилей SBS_BOTTOMALIGN или SBS_TOPALIGN, то полоса прокрутки имеет высоту, ширину и расположение, передаваемые в параметре rect

SBS_LEFTALIGN	Используется со стилем SBS_VERT; правый край полосы прокрутки выравнивается по правому краю прямоугольника, заданного в параметре rect; полоса прокрутки имеет ширину, заданную по умолчанию для системных полос прокрутки
SBS_SIZEBOX	Определяет размер полосы прокрутки; если не задан ни один из стилей SBS_SIZEBOXBOTTOMRIGHTALIGN или SBS_SIZEBOXTOPLEFTALIGN, то полоса прокрутки имеет ширину, высоту и расположение, заданные в параметре rect
SBS_SIZEBOXBOTTOMRIGHTALIGN	Используется со стилем SBS_SIZEBOX; правый нижний угол выравнивается по нижнему правому углу прямоугольника, заданного в параметре rect
SBS_SIZEBOXTOPLEFTALIGN	Используется со стилем SBS_SIZEBOX; верхний левый угол полосы прокрутки выравнивается по нижнему правому углу прямоугольника, заданного в параметре rect
SBS_TOPALIGN	Используется со стилем SBS_HORZ; верхний край полосы прокрутки выравнивается по верхнему краю прямоугольника, заданного в параметре rect; высота полосы прокрутки равняется высоте полосы прокрутки, установленной в системе по умолчанию
SBS_VERT	Определяет вертикальную полосу прокрутки; если не задан ни один из стилей SBS_RIGHTALIGN или SB_LEFTALIGN, то полоса прокрутки имеет высоту, ширину и расположение, передаваемые в параметре rect

Кроме вышеперечисленных, следует использовать следующие стили:

WS_CHILD	Следует указывать всегда при создании полосы прокрутки
WS_VISIBLE	Следует указывать, если полоса прокрутки должна быть изначально видимой
WS_DISABLED	Следует указывать, если полоса прокрутки должна быть изначально недоступной
WS_TABSTOP	Следует указывать, если полоса прокрутки должна входить в группу элементов управления, которым по очереди передается фокус ввода при нажатии клавиши табуляции

Параметр rect указывает расположение и размер полосы прокрутки. Может быть структурой RECT или объектом класса CRect. Параметр pParentWnd задает родительское окно (обычно объект CDialog), и *не может* быть равен NULL, а параметр nID задает идентификатор полосы прокрутки.

Работа с полосой прокрутки

После вызова функции Create, а также после создания полосы прокрутки, заданной в качестве ресурсов, минимальное и максимальное значения гра-

ниц прокрутки равны 0, также равен нулю и размер страницы (см. описание структуры SCROLLINFO далее в тексте).

Для того чтобы получить значения границ прокрутки, предназначена пара следующих функций:

void CScrollBar::**GetScrollRange**(
LPINT lpMinPos,
LPINT lpMaxPos) —

позволяет получить значения для границ прокрутки. Параметр lpMinPos — указатель на переменную целого типа, в которую записывается значение нижней границы полосы прокрутки, а параметр lpMaxPos — указатель на переменную целого типа, в которую записывается значение верхней границы полосы прокрутки.

Функция SetScrollRange, напротив, позволяет установить новые значения для границ прокрутки:

void CScrollBar::**SetScrollRange**(
int nMinPos, int nMaxPos,
BOOL bRedraw = TRUE) —

параметр *bRedraw* задает, должна ли полоса прокрутки быть перерисована для отражения произведенных изменений. Если bRedraw = TRUE, то полоса прокрутки должна быть перерисована, если bRedraw = FALSE — то нет. Установка обеих границ равными нулю приводит к тому, что полоса прокрутки окна, в соответствии с заданным стилем, становится невидимой.

Если за вызовом функции SetScrollRange следует вызов функции SetScrollPos, то параметр bRedraw нужно установить в FALSE во избежание перерисовки полосы прокрутки дважды. Разница значений между параметрами, задаваемыми в nMinPos и nMaxPos не должна превышать 32 767.

Для того чтобы определить или установить текущее положение ползунка на полосе прокрутки, предназначены следующие функции:

int CScrollBar::**GetScrollPos**( ) —

возвращает текущее положение ползунка полосы прокрутки. (В случае ошибки возвращается 0.)

Примечание

Текущее положение ползунка полосы прокрутки представляет собой относительное значение, которое зависит от текущего диапазона прокрутки. Например, если значения для прокрутки лежат в диапазоне от 100 до 200 и ползунок полосы прокрутки находится в ее середине, то текущая позиция будет равна 150.

int CScrollBar::**SetScrollPos**(
int nPos,
BOOL bRedraw = TRUE) —

устанавливает текущее положение ползунка на полосе прокрутки, задаваемое параметром nPos, которое должно лежать в заданном диапазоне, и при bRedraw = TRUE производит перерисовку полосы прокрутки для отражения нового положения.

```
// Задаем пределы прокрутки от 50 до 150,
// и во избежание повторной перерисовки
// задаем параметр bRedraw = FALSE
m_ScrollBar.SetScrollRange(50, 150, FALSE);
// Устанавливаем ползунок в среднее положение
// на полосе прокрутки и производим ее перерисовку
m_ScrollBar.SetScrollPos(100, TRUE)
```

Если необходимо сделать полосу прокрутки видимой или невидимой, то следует воспользоваться следующей функцией:

void CScrollBar::**ShowScrollBar**(BOOL bShow = TRUE) —

показывает или скрывает полосу прокрутки, причем приложение не должно вызывать эту функцию для скрытия полосы прокрутки в момент обработки сообщения от нее.

Для того чтобы сделать полосу прокрутки полностью или частично недоступной, предназначена следующая функция:

BOOL CScrollBar::**EnableScrollBar**(
UINT nArrowFlags = ESB _ENABLE_BOTH) —

делает доступными или недоступными одну или обе кнопки со стрелками на полосе прокрутки. Функция возвращает значение TRUE, если произведены запрашиваемые изменения, и FALSE — если стрелки уже находятся в запрашиваемом состоянии или произошла ошибка. Параметр nArrowFlags задает, какие кнопки со стрелками должны быть сделаны доступными или недоступными. Параметр может принимать следующие значения:

ESB_ENABLE_BOTH	Сделать доступными обе кнопки со стрелками на полосе прокрутки
ESB_DISABLE_LTUP	Сделать недоступными левую кнопку со стрелкой на горизонтальной или верхнюю на вертикальной полосе прокрутки
ESB_DISABLE_RTDN	Сделать недоступными правую кнопку со стрелкой на горизонтальной или нижнюю на вертикальной полосе прокрутки
ESB_DISABLE_BOTH	Сделать недоступными обе кнопки со стрелками на полосе прокрутки

Флаги имеют значения, при которых запрещение одной стрелки ведет к *автоматическому* разрешению другой.

Для получения или изменения параметров полосы прокрутки предназначена следующая пара функций:

BOOL CScrollBar::**GetScrollInfo**(
LPSCROLLINFO lpScrollInfo, UINT nMask) —

возвращает структуру, содержащую параметры полосы прокрутки.

BOOL CScrollBar::**SetScrollInfo**(
LPSCROLLINFO lpScrollInfo, BOOL bRedraw = TRUE) —

устанавливает параметры полосы прокрутки.

Параметр lpScrollInfo в обеих функциях является указателем на структуру SCROLLINFO, а параметр nMask первой функции указывает, какие параметры полосы прокрутки должны быть переданы. Обычно используются в качестве маски флаги SIF_ALL и комбинации из SIF_PAGE, SIF_POS, SIF_TRACKPOS и SIF_RANGE (см. ниже описание структуры SCROLLINFO).

Параметр bRedraw позволяет управлять режимом перерисовки полосы прокрутки в случае изменения ее свойств.

Структура SCROLLINFO содержит информацию о полосе прокрутки, включая минимальное и максимальное значения границ прокрутки, размер страницы и положение ползунка на полосе прокрутки. Она имеет следующее описание:

```
typedef struct tagSCROLLINFO {
  UINT cbSize;
  UINT fMask;
  int nMin;
  int nMax;
  UINT nPage;
  int nPos;
  int nTrackPos;
} SCROLLINFO;
typedef SCROLLINFO, FAR *LPSCROLLINFO;
```

cbSize — задает размер структуры (в байтах); fMask — задает параметры полосы прокрутки и может быть комбинацией следующих значений:

SIF_ALL	Комбинация значений SIF_PAGE, SIF_POS и SIF_RANGE
SIF_DISABLENOSCROLL	Используется только при установке параметров полосы прокрутки; если новые параметры полосы прокрутки делают ее ненужной, то это значение приводит к тому, что полоса прокрутки становится недоступной вместо того, чтобы стать невидимой
SIF_PAGE	Параметр nPage содержит размер страницы
SIF_POS	Параметр nPos содержит положение ползунка на полосе прокрутки
SIF_RANGE	Параметры nMin и nMax содержат информацию о минимальном и максимальном значениях границ прокрутки

nMin — задает минимальное значение границы для прокрутки; nMax — задает максимальное значение границы для прокрутки; nPage — задает размер страницы для автоматического определения текущего положения ползунка при прокрутке на одну страницу (по умолчанию размер страницы равен 0); nPos — задает положение ползунка на полосе прокрутки; nTrackPos — задает непосредственное положение ползунка на полосе прокрутки в момент его перемещения при помощи мыши. Приложение не может изменять значение этого параметра, и функция SetScrollInfo игнорирует это поле.

При запросе параметров полосы прокрутки структуру SCROLLINFO *нет необходимости* инициализировать какими-либо значениями, включая поле, определяющее размер структуры, и поле маски.

Следующая функция избавляет от необходимости создавать дополнительные переменные для определения границ прокрутки, в случае если минимальное значение границы равно 0. Для этого можно воспользоваться вызовом функции GetScrollLimit.

int CScrollBar::**GetScrollLimit**( ) —

возвращает максимальный предел для прокрутки, в случае ошибки возвращает 0. К сожалению, эта функция возвращает значение *большее на единицу* задаваемого при вызове функции CScrollBar::SetScrollRange.

Для полос прокрутки, созданных в окнах в соответствии с заданным стилем окон, невозможно получить доступ к объектам класса CScrollBar (см. описание функции CWnd::OnHScroll в начале раздела). Однако сам класс CWnd имеет в своем составе функции, аналогичные по назначению и наименованию функциям класса CScrollBar.

Глава 24

Классы элементов управления с расширенными возможностями

Библиотека классов предоставляет разработчикам четыре дополнительных класса с расширенными возможностями, производных от классов, инкапсулирующих возможности стандартных элементов управления Windows.

Класс	Базовый класс	Элемент управления
CBitmapButton	CButton	Кнопки с заданными рисунками для всех возможных состояний
CComboBoxEx	CComboBox	Комбинированный список с возможностью непосредственной работы со списками изображений
CCheckListBox	CListBox	Список элементов, реализованных в виде флажков
CDragListBox	CListBox	Список с возможностью переноса его элементов при помощи мыши

Далее мы подробно остановимся на каждом из этих четырех классов.

Кнопки с нанесенным битовым изображением

Класс CBitmapButton предназначен для создания кнопок, на которых вместо текста будет размещено битовое изображение. Кнопка может иметь до четырех растровых изображений, вывод которых в зависимости от состояния кнопки и поддерживает класс CBitmapButton. Суть реализации этого класса в том, что он позволяет создавать командные кнопки — объекты Windows, отображение которых берет на себя приложение. В данном случае, отображением кнопки во всевозможных состояниях занимаются функции класса, исполь-

зующие созданные разработчиком изображения. Состояния, определенные для кнопок: Нормальное (доступное, ненажатое), Нажатое, С фокусом ввода, Недоступное.

В принципе, достаточно определить только один битовый массив для первого состояния, чтобы создать объект рассматриваемого класса. Рамка кнопки должна содержаться в самом изображении. Размер битовых массивов может быть любым, но размер собственно объекта-кнопки определяется первым заданным битовым массивом, т. е. массивом для нормального состояния кнопки. Если битовые массивы для других состояний имеют размеры, отличные от размера ненажатой кнопки, следует привести размер кнопки в соответствие с размером конкретного битового массива.

Различные виды применения требуют различное количество изображений для кнопки:

Состояние кнопки				
Нормальное	**Нажатое**	**С фокусом ввода**	**Недоступное**	**Примечание**
X				Вывод битового массива
X	X			Кнопка без стиля WS_TABSTOP
X	X	X	X	Кнопка со всеми состояниями в блоке диалога
X	X	X		Кнопка со стилем WS_TABSTOP в блоке диалога

При создании кнопки в шаблоне блока диалога необходимо обязательно установить стиль BS_OWNERDRAW. При создании кнопки чисто программным путем, все, что требуется, сделают функции класса CBitmapButton.

Кнопки в блоке диалога

Для включения кнопки в блок диалога необходимо выполнить следующие шаги:

1. Создать от одного до четырех битовых массивов (по мере надобности), которые будут отображать состояния кнопки.

2. Создать шаблон блока диалога, определив в нем самоотображающуюся (owner redraw) кнопку. Размер кнопки не имеет значения, так как при размещении ее размер будет автоматически изменен в соответствии с размером битового массива.

Примечание

Обратите внимание на то, что поиск и загрузка битовых массивов для кнопки осуществляются на основе строки, которая задана для размещения на кнопке (хотя в действительности она на кнопку помещена не будет, поиск битовых изображений осуществляется с ее помощью). Поэтому следует задать строку текста для помещения на кнопке (например, как "IMAGE") и идентификатор кнопки (IDC_IMAGE). Для каждого битового изображения необходимо задать строго определенное имя: оно образуется путем добавления к имени, использованному при создании кнопки (в нашем примере — "IMAGE"), букв "U", "D", "F" или "X" ("U" — для нормального состояния кнопки (up), "D" — для нажатого (down), "F" — для кнопки, имеющей фокус ввода (focused), и "X" — когда кнопка недоступна). В данном случае битовые изображения получат идентификаторы "IMAGEU", "IMAGED", "IMAGEF" и "IMAGEX". Следует еще раз особо подчеркнуть, что битовые изображения имеют не числовые, а *строковые* идентификаторы. В противном случае битовые изображения не будут найдены и, соответственно, отображены.

3. В класс, производный от CDialog, необходимо добавить объект CBitmapButton.
4. Из функции CDialog::OnInitDialog вызвать функцию CBitmapButton ::AutoLoad, указав в качестве параметров идентификатор кнопки, в нашем случае это IDC_IMAGE, и указатель на родительское окно — this.

Кнопки, созданные программным путем

При создании кнопки внутри окна необходимо выполнить следующие шаги:

1. Создать от одного до четырех битовых массивов (по мере необходимости), которые будут отображать состояния кнопки.
2. Создать объект класса CBitmapButton.
3. Вызвать функцию Create для создания элемента управления Windows "кнопка" и связи его с объектом класса CBitmapButton.
4. Вызвать функцию CBitmapButton::LoadBitmaps для загрузки ресурсов (битовых массивов) после того, как кнопка уже создана.

Функции класса

Сообщения, посылаемые объектом класса CBitmapButton имеют тот же смысл и обрабатываются так же, как и при работе с обычной кнопкой.

Как обычно, конструктор класса создает лишь объект CBitmapButton, никак не воздействуя на Windows. Программное создание объекта Windows, как обычно, требует вызова функции Create. Но в отличие от ранее описанных классов, CBitmapButton не имеет в своем составе такой функции — создание объекта будет осуществляться при помощи функции базового класса, т. е. CButton::Create. Это скорее недоработка, чем попытка дать возможность известной гибкости разработчикам приложений. Поэтому важно при таком создании рассматриваемых кнопок не забыть указать стиль BS_OWNERREDRAW.

Для загрузки битовых изображений предназначены функции:

BOOL CBitmapButton::**LoadBitmaps**(
LPCTSTR lpszBitmapResource,
LPCTSTR lpszBitmapResourceSel = NULL,
LPCTSTR lpszBitmapResourceFocus = NULL,
LPCTSTR lpszBitmapResourceDisabled = NULL)

и

BOOL CBitmapButton::**LoadBitmaps**(
UINT nIDBitmapResource,
UINT nIDBitmapResourceSel = 0,
UINT nIDBitmapResourceFocus = 0,
UINT nIDBitmapResourceDisabled = 0) —

позволяют загружать ресурсы, имеющие как строковые, так и числовые идентификаторы.

Данную функцию можно использовать, если нужно задать битовые массивы для помещения их на кнопку по именам или идентификаторам (для каждого индивидуально) или тогда, когда невозможно использование функции CBitmapButton::AutoLoad, например, при создании кнопки вне блока диалога.

При создании кнопки в блоке диалога целесообразнее использовать функцию:

BOOL CBitmapButton::**AutoLoad**(
UINT nID,
CWnd* pParent) —

параметр nID задает идентификатор кнопки, а параметр pParent является указателем на объект-владелец кнопки, т. е. блок диалога.

При использовании данной функции следует строго соблюдать порядок создания кнопки и битовых массивов, предназначенных для размещения на ней (см. раздел "Кнопки в блоке диалога").

Следующая функция позволяет изменять размер кнопки в соответствии с размером уже загруженных битовых массивов:

void CBitmapButton::**SizeToContent**( ) —

эту функцию следует использовать в случае, если битовые массивы имеют различные размеры, и тогда, когда происходит смена состояний кнопки.

Список, имеющий флажки

Класс CCheckListBox позволяет создать элемент управления, сочетающий свойства списка и флажков (кнопок выбора): он представляет собой обычный список (LIST BOX), в котором к каждой строке добавлен слева флажок (CHECK BOX), устанавливая или снимая который пользователь может выбирать строки (элементы) списка.

Создаваемый таким образом список всегда является списком, внешний вид которого определяется пользователем, так как каждый его элемент содержит не только текстовую строку. В простейшем случае список может содержать текстовые строки и флажки в каждой строке, но это совсем не обязательно, можно определить список, в котором будут выводиться флажки и небольшие растровые изображения.

Если создается список, содержащий только текст и флажки, то для вывода строк списка можно использовать функцию по умолчанию CCheckListBox ::DrawItem. Но в любом случае необходимо переопределить функции CListBox::CompareItem, CCheckListBox::DrawItem и CCheckListBox ::MeasureItem.

Инициализация

Создание объекта CCheckListBox и собственно списка Windows в коде программы требует вызова конструктора класса и функции Create:

BOOL CCheckListBox::**Create**(
DWORD dwStyle,
const RECT & rect,
CWnd* pParentWnd,
UINT nID) —

создает элемент управления Windows и связывает его с объектом класса CCheckListBox. Параметр dwStyle задает стиль списка. Он должен содержать стиль LBS_OWNERDRAWFIXED (все элементы в списке одинаковой высоты) или LBS_OWNERDRAWVARIABLE (элементы в списке имеют разную высоту). Эти стили можно комбинировать со стилями обычного списка (см. описание класса CListBox). Параметр rect задает размер и расположение списка на экране; параметр pParentWnd, который *не может* быть равен NULL, задает родительское окно (обычно объект CDialog); параметр nID задает идентификатор списка.

Функции для работы с расширенным списком

Следующая пара функций позволяет определить и установить состояние флажка для элемента списка:

int CCheckListBox::**GetCheck**(int nIndex) —

возвращает состояние флажка для элемента списка.

void CCheckListBox::**SetCheck**(
int nIndex,
int nCheck) —

устанавливает состояние флажка. Параметр nIndex является номером элемента в списке, а параметр nCheck определяет состояние флажка: 0 — снят, 1 — установлен, 2 — неопределенное состояние.

Для того чтобы сделать доступным/недоступным элемент списка и определить его состояние, предназначены следующие функции:

BOOL CCheckListBox::**IsEnabled**(int nIndex) —

определяет, доступен ли элемент списка

void CCheckListBox::**Enable**(
int nIndex,
BOOL bEnabled = TRUE) —

делает доступным или недоступным элемент списка.

Для получения и установления стиля флажков списка предназначены функции:

UINT CCheckListBox::**GetCheckStyle**() —

возвращает стиль флажков.

void CCheckListBox::**SetCheckStyle**(UINT nStyle) —

параметр nStyle задает стиль флажков. Флажки могут иметь следующие стили: BS_CHECKBOX, BS_AUTOCHECKBOX, BS_AUTO3STATE или BS_3STATE.

Виртуальные функции

virtual CRect CCheckListBox::**OnGetCheckPosition**(
CRect rectItem,
CRect rectCheckBox) —

вызывается библиотекой для определения положения флажка записи списка, где rectItem определяет расположение и размер элемента списка, а rectCheckBox — размер и расположение (по умолчанию) флажка строки списка.

Реализация по умолчанию этой функции возвращает размер и расположение флажка, созданного с параметрами по умолчанию (выровнен по левому верхнему углу записи и имеет стандартный размер). Однако флажок может располагаться в правой части записи списка и иметь размер больше или меньше размера, установленного по умолчанию. В этих случаях необходимо переопределить эту функцию для того, чтобы библиотека могла получить правильные значения.

В следующем примере функция переопределяет функцию по умолчанию, помещает флажок в правой части элемента списка и переопределяет его размер:

```
CRect CMercuryCheckListBox::OnGetCheckPosition(
  CRect rectItem, CRect rectCheckBox)
{
  CRect rectMyCheckBox;
  rectMyCheckBox.top = rectItem.top -1;
  rectMyCheckBox.bottom = rectItem.bottom -1;
  rectMyCheckBox.right = rectItem.right -1;
  rectMyCheckBox.left =
    rectItem.right -1 — rectCheckBox.Width();
  return rectMyCheckBox;
}
```

virtual void CCheckListBox::**DrawItem**(
LPDRAWITEMSTRUCT lpDrawItemStruct) —

вызывается библиотекой при необходимости перерисовки элемента списка, внешний вид которого определяется пользователем.

virtual void CCheckListBox::**MeasureItem**(
LPMEASUREITEMSTRUCT lpMeasureItemStruct) —

вызывается библиотекой при необходимости определения высоты элемента списка, внешний вид которого определяется пользователем.

Следующий пример иллюстрирует создание списка, содержащего флажки, с использованием класса CChecklistBox.

```
//Файл CheckBoxDlg.h
#include "ColorBox.h"
class CCheckBoxDlg : public CDialog
{
public:
  CCheckBoxDlg(CWnd* pParent = NULL);  // standard constructor
// Dialog Data
  //{{AFX_DATA(CCheckBoxDlg)
     enum { IDD = IDD_CHECKBOX_DIALOG };
    // NOTE: the ClassWizard will add data members here
  //}}AFX_DATA
  //{{AFX_VIRTUAL(CCheckBoxDlg)
  protected:
  virtual void DoDataExchange(CDataExchange* pDX);
  //}}AFX_VIRTUAL
protected:
  HICON m_hIcon;
```

```
  CColorBox m_CheckBox;
  //{{AFX_MSG(CCheckBoxDlg)
  virtual BOOL OnInitDialog();
  afx_msg void OnPaint();
  afx_msg HCURSOR OnQueryDragIcon();
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()
};
//Файл CheckBoxDlg.cpp
CCheckBoxDlg::CCheckBoxDlg(CWnd* pParent /*=NULL*/)
  : CDialog(CCheckBoxDlg::IDD, pParent)
{
  m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}
void CCheckBoxDlg::DoDataExchange(CDataExchange* pDX)
{
  CDialog::DoDataExchange(pDX);
  //{{AFX_DATA_MAP(CCheckBoxDlg)
    // NOTE: the ClassWizard will add DDX and DDV calls here
  //}}AFX_DATA_MAP
}
BEGIN_MESSAGE_MAP(CCheckBoxDlg, CDialog)
  //{{AFX_MSG_MAP(CCheckBoxDlg)
  ON_WM_PAINT()
  ON_WM_QUERYDRAGICON()
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()
BOOL CCheckBoxDlg::OnInitDialog()
{
  CDialog::OnInitDialog();
  SetIcon(m_hIcon, TRUE);     // Set big icon
  SetIcon(m_hIcon, FALSE);    // Set small icon
  // Связываем объект класса CColorBox с элементом
  // управления по идентификатору
  VERIFY(m_CheckBox.SubclassDlgItem(IDC_CHECKBOX, this));
  // Заполняем список строками, в которых выводятся различные цвета -
  // цветов всего 8, и их определяет наличие или отсутствие сооставляющей
  for (int red = 0; red <= 255; red += 255)
    for (int green = 0; green <= 255; green += 255)
```

```
      for (int blue = 0; blue <= 255; blue += 255)
        m_CheckBox.AddColorItem(RGB(red, green, blue));

  return TRUE;  // return TRUE  unless you set
                // the focus to a control
}
//Файл ColorBox.h
class CColorBox : public CCheckListBox
{
public:
  CColorBox();
public:
  void AddColorItem(COLORREF color);
//{{AFX_VIRTUAL(CColorBox)
  public:
  virtual int CompareItem(LPCOMPAREITEMSTRUCT lpCompareItemStruct);
  virtual void DrawItem(LPDRAWITEMSTRUCT lpDrawItemStruct);
  virtual void MeasureItem(LPMEASUREITEMSTRUCT lpMeasureItemStruct);
  //}}AFX_VIRTUAL
public:
  virtual ~CColorBox();
protected:
  //{{AFX_MSG(CColorBox)
    // NOTE — the ClassWizard will add and
    //        remove member functions here.
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()
};
//Файл ColorBox.cpp
CColorBox::CColorBox(){}
CColorBox::~CColorBox(){}
BEGIN_MESSAGE_MAP(CColorBox, CCheckListBox)
  //{{AFX_MSG_MAP(CColorBox)
    // NOTE — the ClassWizard will add and
    //        remove mapping macros here.
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()
void CColorBox::AddColorItem(COLORREF color)
{
```

```
  AddString((LPCTSTR) color); // добавляем строку с список
}
// Переопределяем функцию для сравнения строк списка
int CColorBox::CompareItem(
  LPCOMPAREITEMSTRUCT lpCompareItemStruct)
{
  // return -1 = item 1 sorts before item 2
  // return 0 = item 1 and item 2 sort the same
  // return 1 = item 1 sorts after item 2
  COLORREF cr1 = (COLORREF)lpCompareItemStruct->itemData1;
  COLORREF cr2 = (COLORREF)lpCompareItemStruct->itemData2;
  if (cr1 == cr2)
   return 0; // строки одинаковы
  // Сначала сортируем строки по интенсивности
  int intensity1 = GetRValue(cr1) +
                   GetGValue(cr1) +
                   GetBValue(cr1);
  int intensity2 = GetRValue(cr2) +
                   GetGValue(cr2) +
                   GetBValue(cr2);
  if (intensity1 < intensity2)
    return -1; // Сначала идут строки с меньшей яркостью
  else if (intensity1 > intensity2)
    return 1; // Более яркие строки идут вторыми
  // Если строки имеют одинаковую интенсивность,
  // то сортируем по цветам:
  //   первый — синий, последний — красный
  if (GetBValue(cr1) > GetBValue(cr2))
    return -1;
  else if (GetGValue(cr1) > GetGValue(cr2))
    return -1;
  else if (GetRValue(cr1) > GetRValue(cr2))
    return -1;
  else
    return 1;
  return 0;
}
// Переопределяем функцию для рисования строк списка,
// причем рисуем только содержимое каждого из
```

```
// элементов - квадрат и знак в нем (установлен/снят)
// рисуются без нашего участия
void CColorBox::DrawItem(
  LPDRAWITEMSTRUCT lpDrawItemStruct)
{
  CDC* pDC = CDC::FromHandle(lpDrawItemStruct->hDC);
  COLORREF cr = (COLORREF)lpDrawItemStruct->itemData;
  if (lpDrawItemStruct->itemAction & ODA_DRAWENTIRE)
  {
    CBrush br(cr);
    pDC->FillRect(&lpDrawItemStruct->rcItem, &br);
  }
  if ((lpDrawItemStruct->itemState & ODS_SELECTED) &&
    (lpDrawItemStruct->itemAction &
    (ODA_SELECT | ODA_DRAWENTIRE)))
  {
    // Выделяем рамку, если запись выделена
    COLORREF crHilite = RGB(255-GetRValue(cr),
                            255-GetGValue(cr),
                            255-GetBValue(cr));
    CBrush br(crHilite);
    pDC->FrameRect(&lpDrawItemStruct->rcItem, &br);
  }
  if (!(lpDrawItemStruct->itemState & ODS_SELECTED) &&
    (lpDrawItemStruct->itemAction & ODA_SELECT))
  {
    // Удаляем рамку, если выделение с записи списка снято
    CBrush br(cr);
    pDC->FrameRect(&lpDrawItemStruct->rcItem, &br);
  }
}
// Переопределяем функцию для определения размеров строк списка
void CColorBox::MeasureItem(
  LPMEASUREITEMSTRUCT lpMeasureItemStruct)
{
  lpMeasureItemStruct->itemHeight = COLOR_ITEM_HEIGHT;
}
```

После запуска приложения на экран будет выведен блок диалога (рис. 24.1), содержащий список, в котором строки будут иметь различные цвета:

Рис. 24.1. Блок диалога после осуществления выбора

Обратите внимание на то, что связывание члена CCheckBoxDlg::m_CheckBox происходит без механизма динамического обмена данными (DDX, Dynamic Data Exchange).

Комбинированный список с поддержкой списка образов

Класс CComboBoxEx позволяет создать комбинированный список, который значительно упрощает работу с изображениями в комбинированном списке. В отличие от стандартного комбинированного списка, при использовании этого класса нет необходимости самому реализовывать рисование образов, поскольку из него теперь есть прямой доступ к рисункам из списка образов. Элементы этого класса поддерживают до трех образов на элемент: для выбранного, невыбранного и недоступного состояний.

Создание комбинированного списка

Как обычно, прежде всего нужно создать объект класса с помощью конструктора CComboBoxEx::CComboBoxEx. Затем, следуя известной схеме, необходимо создать объект Windows и связать его с объектом класса. Для этого в классе реализована функция:

BOOL CComboBoxEx::**Create**(
    DWORD dwStyle,
    const RECT& rect,
    CWnd* pParentWnd,
    UINT nID) —

открывает доступ к расширенному комбинированному списку Windows. Функция имеет следующие параметры: dwStyle — задает комбинацию стилей; может принимать только следующие значения — все другие игнорируются:

CBS_SIMPLE	Список изображается на экране все время, текущий выделенный элемент списка отображается в текстовом поле; редактирование разрешено

CBS_DROPDOWN	Подобен стилю CBS_SIMPLE за исключением того, что список не изображается до тех пор, пока пользователь не нажмет кнопку со стрелкой вниз, расположенную справа от текстового поля комбинированного списка. При этом список раскрывается из текстового поля (раскрывающийся список); редактирование разрешено
CBS_DROPDOWNLIST	Подобен CBS_DROPDOWN, но вместо текстового поля имеется строка со статическим текстом, который отображает текущий выбор, но редактировать его нельзя
WS_CHILD	Всегда является дочерним окном

rect — определяет размер и положение элемента управления; pParentWnd — указатель на родительское окно (не должен быть NULL); nID — идентификатор элемента управления.

Основные операции

К основным операциям относятся добавление и удаление элемента из расширенного комбинированного списка, а также установка и получение подробной информации об элементе. Но обо всем по порядку.

int CComboBoxEx::**InsertItem**(const COMBOBOXEXITEM* pCBItem) —

вставляет новый элемент в расширенный комбинированный список. В случае успешного завершения функция возвращает индекс вставленного элемента, и –1 в противном случае. При вызове этой функции формируется сообщение WM_NOTIFY с уведомлением CBEN_INSERTINEM родительскому окну.

В качестве параметра функция принимает указатель на структуру COMBOBOXEXITEM, которая содержит информацию об элементе и имеет следующие поля:

```
typedef struct {
    UINT    mask;
    int     iItem;
    LPTSTR  pszText;
    int     cchTextMax;
    int     iImage;
    int     iSelectedImage;
    int     iOverlay;
    int     iIndent;
    LPARAM  lParam;
   } COMBOBOXEXITEM, *PCOMBOBOXEXITEM;
```

mask — набор флагов, которые определяют доступные поля структуры; это может быть комбинация следующих значений:

CBEIF_DI_SETITEM	Элемент уже хранится в расширенном комбинированном списке и не надо вставлять его снова. Этот флаг используется только с уведомлением CBEN_GETDISPINFO.
CBEIF_IMAGE	Доступно поле iImage

CBEIF_INDENT	Доступно поле iIndent
CBEIF_LPARAM	Доступно поле lParam
CBEIF_OVERLAY	Доступно поле iOverlay
CBEIF_SELECTEDIMAGE	Доступно поле iSelectedImage
CBEIF_TEXT	Доступно поле pszText

iItem — индекс элемента; pszText — указатель на буфер, который содержит и/или получает текст, ассоциированный с элементом; размер буфера задается в поле cchTextMax. Если параметр получает значение LPSTR_TEXTCALLBACK, то элемент управления будет получать соответствующую информацию, используя уведомление CBEN_GETDISPINFO. iImage — индекс (начиная с 0) рисунка в списке образов, который отображает невыбранное состояние элемента; iSelectedImage — индекс (начиная с 0) рисунка в списке образов, который отображает выбранное состояние элемента; iOverlay — индекс (начиная с 1) рисунка в списке образов, который отображает перекрытое состояние элемента. Если три последних параметра принимают значение I_IMAGECALLBACK, то элемент управления будет получать соответствующую информацию, используя уведомление CBEN_GETDISPINFO. iIndent — число отступов для элемента (каждый отступ равен 10 пикселям). Если параметр получает значение I_INDENTCALLBACK, то элемент управления будет получать соответствующую информацию, используя уведомление CBEN_GETDISPINFO. lParam — 32-битное значение, ассоциированное с элементом.

Эта же структура используется и в двух других функциях:

BOOL CComboBoxEx::**GetItem**(COMBOBOXEXITEM* pCBItem)

и

BOOL CComboBoxEx::**SetItem**(const COMBOBOXEXITEM* pCBItem) —

соответственно, получают или устанавливают информацию об элементе расширенного комбинированного списка.

Для удаления элемента из расширенного комбинированного списка реализована функция:

int CComboBoxEx::**DeleteItem**(int iIndex) —

возвращает число элементов, оставшихся в расширенном комбинированном списке, и CD_ERR в случае неудачного выполнения. Параметр содержит номер удаляемого элемента.

Функции работы с атрибутами

Таких функций в классе немного, поскольку основное поведение он наследует от своего базового класса CComboBox. Тем не менее, рассмотрим их.

BOOL CComboBoxEx::**HasEditChanged**( ) —

позволяет узнать, изменил ли пользователь содержимое текстового поля, ассоциированного с расширенным комбинированным списком. Она возвращает ненулевое значение, если это произошло, и 0 в противном случае.

Примечание

К этому моменту наблюдательные читатели наверняка обратили внимание на то, что для функций, возвращающих BOOL, мы используем два стиля коммен-

тариев: либо *ненулевое* значение, либо *TRUE*. Делаем мы так, чтобы соблюсти стиль фирмы Microsoft. От себя же хотим добавить, что значение FALSE всегда одно и то же — 0. К сожалению, этого нельзя сказать о значении TRUE. Точно известно только одно — это значение отлично от нуля. Поэтому в программах не рекомендуются проверки вида:

```
if (HasEditChanged( ) == TRUE)
```

а только

```
if (HasEditChanged( ) != FALSE)
```

Следующие две функции позволяют получить и установить дополнительные стили расширенного комбинированного списка

DWORD CComboBoxEx::**GetExtendedStyle**( ) const

и

DWORD CComboBoxEx::**SetExtendedStyle**(
DWORD dwExMask,
DWORD dwExStyles) —

в качестве дополнительных стилей для рассматриваемого элемента управления определены следующие (параметр dwExStyles):

CBES_EX_CASESENSITIVE	Строки в расширенном комбинированном списке чувствительны к регистру
CBES_EX_NOEDITIMAGE	В текстовом поле не изображаются элементы образов
CBES_EX_NOEDITIMAGEINDENT	В текстовом поле нет отступа текста, освобождающего пространство для элемента образа
CBES_EX_NOSIZELIMIT	Вертикальный размер расширенного комбинированного списка может уменьшаться до размеров, меньших чем элемент управления. В этом случае элемент управления "комбинированный список" отсекается
CBES_EX_PATHWORDBREAKPROC	В качестве разделителей слов в поле редактирования могут использоваться слэш (/), обратный слэш (\), и точка (.), что позволит перемещаться по комбинированному списку с помощью клавиш <Ctrl>+<стрелка>. Этот стиль применим только в Windows NT

Параметр dwMask показывает, какие дополнительные стили будут изменяться. Если его значение равно 0, то воздействие оказывается на все дополнительные стили. Функции возвращают, соответственно, текущие или предыдущие дополнительные стили.

Доступ к списку образов, ассоциированному с элементом управления, можно получить с помощью функций:

CImageList* CComboBoxEx::**GetImageList**( ) const

и

CImageList* CComboBoxEx::**SetImageList**(CImageList* pImageList) —

возвращают указатель на *временный* объект класса CImageList. Если в результате выполнения функции SetImageList изменилась высота элемента управления, то необходимо после нее вызвать функцию SetWindowPos, чтобы отобразить изменения на экране.

Две оставшиеся функции позволяют получить доступ непосредственно к элементам управления Windows, естественно посредством соответствующих классов:

CEdit* CComboBoxEx::**GetEditCtrl**( ) —

к редактору текста

и

CComboBox* CComboBoxEx::**GetComboBoxCtrl**( ) —

к комбинированному списку.

Список с возможностью перетаскивания элементов

Класс CDragListBox позволяет создать список, который в дополнение к свойствам обычных списков позволяет пользователю перетаскивать элементы списка внутри него. Списки, обладающие этим свойством, позволяют пользователю упорядочивать элементы и могут оказаться в некоторых случаях очень удобными, например, для перемещения имен файлов в списке. По умолчанию, при перемещении элемента список помещает его в новое место, однако можно настроить параметры так, что элемент будет не перемещаться, а копироваться.

Ограничения на использование

По умолчанию список не может иметь стили LBS_MULTIPLESEL и LBS_SORT. Исключение последнего стиля, определяющего необходимость автоматической сортировки, вполне естественно. Множественный выбор в принципе может пригодиться, хотя тогда не совсем ясно, как реализовывать механизм перетаскивания или копирования, т. к. эти операции производятся только внутри последовательного списка. Кроме того, собственно операция перетаскивания может быть осуществлена только для списков, содержащих строки, т. е. объект-список должен иметь стиль LBS_HASSTRINGS или не иметь стили LBS_OWNERDRAWFIXED или LBS_OWNERDRAWVARIABLE. (Способ, позволяющий отказаться от последнего из перечисленных ограничений, изложен в конце раздела.)

Создание списка

Для создания объекта класса CDragListBox необходимо использовать конструктор. Для создания списка с возможностью перетаскивания элементов на основе шаблона блока диалога, следует создать обычный элемент управления "список" (LIST BOX) и связать с ним объект класса CDragListBox или производного от него. Естественно, что список не должен иметь стили, перечисленные выше.

Функции, предназначенные для перемещения элементов

Класс CDragListBox является производным от класса CListBox и обладает всеми его свойствами, но кроме того он имеет и свои собственные функции, обеспечивающие использование его новых возможностей.

int CDragListBox::**ItemFromPt**(
CPoint pt,
BOOL bAutoScroll = TRUE) —

возвращает индекс перетаскиваемого элемента. Параметр pt — объект класса CPoint, содержащий координаты точки внутри списка, его следует использовать для получения индекса элемента списка. Параметр bAutoScroll определяет, будет ли производиться автоматически прокрутка списка.

virtual void CDragListBox::**DrawInsert**(int nItem) —

вызывается библиотекой для прорисовки указателя вставки элемента до того, как сам элемент будет вставлен. Параметр nItem содержит индекс элемента, в который будет производиться вставка. Значение –1 сбрасывает указатель вставки. Переопределите эту функцию, если требуется задать свой указатель вставки элемента. Функция использует член класса m_nLast, сохраняющий номер элемента списка, после которого находился указатель вставки.

Собственно рисованием указателей нового места для вставки или копирования занимается недокументированная функция класса — DrawSingle. Эта функция вызывается дважды — для предыдущего положения и для текущего положения указателя вставки. Связано это с тем, что указатель рисуется в режиме PATINVERT, когда двойной вывод по одному и тому же месту восстанавливает исходное изображение.

Вызов функции BeginDrag сообщает о начале операции перемещения элемента списка:

virtual BOOL CDragListBox::**BeginDrag**(CPoint pt) —

возвращает значение TRUE, если операция перемещения разрешена; параметр pt определяет точку внутри окна списка.

Функция вызывается библиотекой тогда, когда, возможно, началась операция перемещения строки, например, когда пользователь нажал левую кнопку мыши. Эту функцию стоит переопределить, если необходимо контролировать, что происходит в начале операции перемещения записи списка. Реализация по умолчанию данной функции захватывает мышь, и строка остается в режиме перетаскивания, пока пользователь не отпустит левую или нажмет правую кнопку мыши или клавишу <Esc.> В последних двух случаях операция перемещения отменяется, и происходит вызов функции CancelDrag:

virtual void CDragListBox::**CancelDrag**(CPoint pt) —

параметр pt определяет точку внутри окна списка.

Во время перетаскивания обработчик уведомлений вызывает функцию Dragging:

virtual UINT CDragListBox::**Dragging**(CPoint pt) —

параметр pt передает экранные координаты текущего положения указателя мыши, который может находиться и вне окна списка. Функция возвращает идентификатор изображаемого курсора:

DL_COPYCURSOR	Обозначает режим копирования элемента (курсор со знаком "+")
DL_MOVECURSOR	Обозначает режим перемещения элемента (обычный курсор)
DL_STOPCURSOR	Обозначает, что элемент не может быть перемещен в указанное место (курсор в виде перечеркнутой окружности)

Реализация по умолчанию функции возвращает DL_MOVECURSOR.

И, наконец, последняя из виртуальных функций — функция Dropped:

virtual void CDragListBox::**Dropped**(
int nSrcIndex, CPoint pt) —

вызывается библиотекой, когда запись перемещена (скопирована) и отпущена. Параметр nSrcIndex задает исходные координаты перемещаемого элемента, параметр pt содержит новые координаты.

Реализация по умолчанию данной функции копирует элемент списка и его данные в новое место и удаляет затем исходный элемент. Если необходимо реализовать копирование элементов при перемещении, то следует переопределить данную функцию.

Методика обхода ограничений

При желании можно допустить работу со списками, не имеющими стиля LBS_HASSTRINGS при наличии стилей LBS_OWNERDRAWFIXED или LBS_OWNERDRAWVARIABLE.

Для этого нужно не только написать свою версию функции DrawItem, но и полностью переписать функцию Dropped, в которой следует отказаться лишь от использования функций GetText и SetText.

Реализация копирования элементов списка

Данный пример иллюстрирует реализацию списка, позволяющего копировать элементы списка в указанное место. Элементами списка являются имена файлов текущего каталога, включая имена подкаталогов. Обратите внимание, что на рис. 24.2 отображен результат предыдущего копирования строки, содержащей имя файла readme.txt.

```
//Файл DragListDlg.cpp
// Связываем член m_DragList с элементом управления
void CDragListDlg::DoDataExchange(CDataExchange* pDX)
{
  CDialog::DoDataExchange(pDX);
```

```
  //{{AFX_DATA_MAP(CDragListDlg)
  DDX_Control(pDX, IDC_DRAGLIST, m_DragList);
  //}}AFX_DATA_MAP
}
BOOL CDragListDlg::OnInitDialog()
{
  CDialog::OnInitDialog();
  // Заполняем элемент управления именами файлов текущего каталога
  m_DragList.Dir(0xFF, "*.*");
  ...
}
// Файл DragBox.h
class CDragBox : public CDragListBox
{
...
public:
  // ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CDragBox)
  //}}AFX_VIRTUAL
  virtual void DrawInsert(int nItem);
  virtual UINT Dragging(CPoint pt);
  virtual void Dropped(int nSrcIndex, CPoint pt);
};
// Файл DragBox.cpp
// Пример реализации функции рисования
// указателя перемещения
void CDragBox::DrawInsert(int nItem)
{
  // Проверяем, было ли перемещение в новое место
  if (m_nLast != nIndex)  {
    DrawSingle(m_nLast);
    DrawSingle(nIndex);
    m_nLast = nIndex;
  }
}
// Основную работу пусть делает функция базового класса -
// мы только изменим вид курсора
UINT CDragBox::Dragging(CPoint pt)
{
  if (CDragListBox::Dragging(pt) == DL_MOVECURSOR)
```

```
    return DL_COPYCURSOR;
  return DL_STOPCURSOR;
}
void CDragBox::Dropped(int nSrcIndex, CPoint pt)
{
  // Проверяем допустимость стилей списка
  ASSERT(!(GetStyle() & LBS_OWNERDRAWFIXED | LBS_OWNERDRAWVARIABLE)) ||
    (GetStyle() & LBS_HASSTRINGS));
  // Удаляем указатель перемещения
  DrawInsert(-1);
  // Определяем новое место для вставки
  int nDestIndex = ItemFromPt(pt);
  // Проверяем необходимость вставки
  if (nSrcIndex == -1 || nDestIndex == -1)
     return;
  if (nDestIndex == nSrcIndex || nDestIndex == nSrcIndex+1)
    return;
  // Получаем информацию об элементе
  CString str;
  DWORD dwData;
  GetText(nSrcIndex, str);
  dwData = GetItemData(nSrcIndex);
  // Вот теперь можно текущий элемент удалить -
  // тогда функция будет производить перемещение
  // Вставляем новый элемент и связываем с ним данные
  if (nSrcIndex < nDestIndex)
    nDestIndex--;
  nDestIndex = InsertString(nDestIndex, str);
  SetItemData(nDestIndex, dwData);
  // Назначаем новый элемент текущим
  SetCurSel(nDestIndex);
}
```

Рис. 24.2. Вид окна приложения во время выбора места для вставки копии имени файла

Глава 25

Вспомогательные элементы управления

Прошло уже больше четырех лет после выхода Windows 95, где появились новые элементы управления, которые существенно расширили небольшой набор так называемых "стандартных" элементов управления. Рассмотрим их более подробно.

Просмотр видеоклипов

Класс CAnimateCtrl поддерживает функциональные возможности элемента управления Windows, который реализует просмотр видеоклипов (элемент управления "анимация").

ОС Windows поддерживает просмотр видеоинформации, записанной в формате AVI (Audio Video Interleaved) — стандартном формате Windows. Такие видеоклипы представляют собой совокупность растровых изображений — *кадров*. Сам просмотр осуществляется в прямоугольной области, размер которой определяется либо заданными размерами элемента управления, либо размерами изображения в зависимости от стиля элемента (см. ниже).

Для возможности просмотра видеоинформация должна соответствовать следующим требованиям (можно просматривать только простейшие видеоклипы):

- Клип должен иметь ровно один видеопоток, содержащий хотя бы один кадр
- Клип может иметь максимум два потока, один из которых обычно представляет звук и игнорируется
- Содержимое клипа может быть либо не сжато, либо сжато с использованием алгоритма RLE8
- Изменение палитры не допускается

Примечание

Для применения в приложениях более совершенных средств мультимедиа следует использовать окна класса MCIWnd (имеется в виду класс объектов Windows, а не класс языка C++).

Видеоклип может быть добавлен к приложению в качестве ресурса типа AVI, например, вот так:

```
IDR_VISUAL  AVI  DISCARDABLE  "res\\Visual.avi"
```

Иными словами, для просмотра любого видеофайла (соответствующего требованиям) достаточно его имени.

Элемент управления "анимация"

Microsoft рекомендует использовать этот элемент управления для визуализации длительных операций. Сама система использует, прежде всего, видеоклипы для визуализации копирования, перемещения и удаления файлов и папок, а также при поиске объектов, например, компьютеров, по сети.

Разработчики программного обеспечения идут дальше. Система Remove-IT 95 (мониторинг использования различных компонентов системы, в основном, приложений и отдельных файлов) компании Vertisoft Systems, Inc. широко использует анимацию в диалоге выбора команд, а также для визуализации работы монитора, собирающего информацию и т. д. (рис. 25.1).

Рис. 25.1. Команды и соответствующие им кнопки с видеоклипами

Класс CAnimateCtrl

Как и для других элементов управления, для просмотра видеоклипов достаточно или создать соответствующий объект в качестве дочернего в любом

окне, или добавить этот элемент управления в шаблон блока диалога. Оба способа позволяют задавать дополнительные стили, свойственные просмотру видеоклипов.

Конструктор создает только объект класса. Для создания непосредственно объекта Windows без использования шаблона блока диалога служит функция Create:

BOOL CAnimateCtrl::**Create**(
DWORD dwStyle,
const RECT & rect,
CWnd* pParentWnd,
UINT nID) —

назначение параметров очевидно. Стоит только отметить, что параметр *dwStyle* может принимать любые комбинации стилей элементов управления плюс следующие, свойственные рассматриваемому:

ACS_CENTER	Обеспечивает центрирование видеоклипа в рамках, заданных при создании элемента управления; при этом можно исключить области клипа, выходящие за рамки элемента управления; в противном случае размер элемента просмотра будет подстроен под размеры клипа; при этом левый верхний угол кадров клипа совпадет с верхним левым углом элемента управления
ACS_TRANSPARENT	Обеспечивает прозрачный режим отображения видеоклипа, т. е. произойдет замена цвета фона кадров цветом фона родительского окна элемента просмотра
ACS_AUTOPLAY	Обеспечивает немедленное и автоматическое начало "проигрывания" видеоклипа в цикле, т. е. продолжение просмотра с начала после завершения

Собственно сам элемент управления (класс CAnimateCtrl) не предусматривает большого числа возможностей для манипулирования видеоклипом. Существует всего пять функций и два уведомляющих сообщения, которые можно обрабатывать.

После создания объекта необходимо загрузить видеоклип из ресурсов или AVI-файла (функция Open). Вызов этой функции с параметром, равным NULL, приведет к освобождению ресурса, что также может быть достигнуто вызовом функции Close. Важной особенностью функции Open является то, что, не начиная "проигрывать" клип, тем не менее, она отображает его первый кадр.

Примечание

Стиль ACS_AUTOPLAY снимает это ограничение — задание этого стиля приведет к немедленному началу "проигрывания".

Для начала "проигрывания" класс предусматривает использование функции Play:

BOOL CAnimateCtrl::**Play**(
UINT nFrom,
UINT nTo,
UINT nRep) —

параметры nFrom и nTo определяют начальный и конечный кадры, считая с нулевого, для "проигрывания". Оба значения должны лежать в пределах от 0 и до 65 535 включительно. Значение параметра nTo, равное –1, указывает, что "проигрывание" необходимо проводить до конца клипа, таким образом, для "проигрывания" клипа полностью нет необходимости знать количество составляющих его кадров. Третий параметр определяет число "проигрываний" клипа. Значение –1 задает бесконечное "проигрывание".

Для остановки "проигрывания" видеоклипа в любой момент времени необходимо вызвать функцию Stop.

Для перемещения к определенному кадру, но без инициализации "проигрывания" служит функция Seek:

BOOL CAnimateCtrl::**Seek**(UINT nTo) —

единственный параметр определяет номер кадра, начиная с нуля, который необходимо отобразить. Значение параметра, равное –1, указывает на необходимость отображения последнего кадра клипа.

Уведомления

Описываемый элемент управления посылает два вида уведомлений. Причем, как исключение из правила, оба уведомления посылаются не в форме сообщения WM_NOTIFY, что обычно для общих элементов управления, а в форме сообщения WM_COMMAND.

Первое сообщение — ACN_START — посылается приложению, а, точнее, родительскому окну, в момент начала "проигрывания" видеоклипа.

Второе сообщение — ACN_STOP — посылается по окончании процесса "проигрывания".

Для того чтобы обработать эти сообщения, в рамках библиотеки классов MFC существует макрос ON_CONTROL, при помощи которого определяют элемент уведомления, тип уведомления и функцию его обработки.

Для наглядности продемонстрируем пример (приложение Animate), который создан на основе блока диалога. Последний содержит два элемента управления для просмотра одного и того же видеоклипа. Суть примера в том, что "проигрывание" осуществляется попеременно то одним элементом управления, то другим. Переключение осуществляется по завершении работы одного из элементов.

```
// Файл AnimateDlg.cpp
// Связываем объекты класса CAnimateCtrl с элементами блока диалога
void CAnimateDlg::DoDataExchange(CDataExchange* pDX)
```

```
{
  CDialog::DoDataExchange(pDX);
  //{{AFX_DATA_MAP(CAnimateDlg)
  DDX_Control(pDX, IDC_ANIMATE_1, m_Animate_1);
  DDX_Control(pDX, IDC_ANIMATE_2, m_Animate_2);
  //}}AFX_DATA_MAP
}
// Добавляем в карту сообщений функции-обработчики
// уведомлений. (Добавление осуществляется вручную,
// т. к. ClassWizard не поддерживает эти уведомления)
BEGIN_MESSAGE_MAP(CAnimateDlg, CDialog)
  //{{AFX_MSG_MAP(CAnimateDlg)
  ON_WM_PAINT()
  ON_WM_QUERYDRAGICON()
  //}}AFX_MSG_MAP
  ON_CONTROL(ACN_STOP, IDC_ANIMATE_1, AnimateStop_1)
  ON_CONTROL(ACN_STOP, IDC_ANIMATE_2, AnimateStop_2)
END_MESSAGE_MAP()
BOOL CAnimateDlg::OnInitDialog()
{
  CDialog::OnInitDialog();
  // "Открываем" клип для первого элемена управления
  m_Animate_1.Open(IDR_VISUAL);
  // ... и немедленно начинаем его просмотр
  m_Animate_1.Play(0, (UINT)-1, 1);
  // "Открываем" клип для второго элемена управления
  m_Animate_2.Open(IDR_VISUAL);
...
}
// Функции обработки уведомления о завершении просмотра
// от первого элемента управления - начинаем просмотр
// во втором элементе управления
void CAnimateDlg::AnimateStop_1()
{
  m_Animate_2.Play(0, (UINT)-1, 1);
}
// Функции обработки уведомления о завершении просмотра
// от второго элемента управления - начинаем просмотр
// в первом элементе управления
```

```
void CAnimateDlg::AnimateStop_2()
{
  m_Animate_1.Play(0, (UINT)-1, 1);
}
```

Рис. 25.2. Окно приложения Animate: слева — активен верхний элемент управления; справа — активен нижний элемент управления

Создание видеоклипов

Просматривать видеоклипы вы научились. А как же все-таки их создавать? Для этого необходимо создать набор файлов формата BMP (кадров) и при помощи таких средств, как Video for Windows фирмы Microsoft, VideoStudio фирмы Ulead Systems, Inc. или аналогичных создать AVI-файл на основе подготовленных кадров. При этом можно создать клип "одним махом" или добавлять по кадру вручную.

В первом случае необходимо, чтобы файлы имели названия следующего формата:

```
<имя><номер>.BMP
```

Нумерация файлов должна соответствовать порядку кадров: frm001.bmp, frm002.bmp, frm003.bmp и т. д. Для преобразования набора файлов-кадров в видеоклип необходимо выполнить команду Image Sequence.

"Горячая клавиша"

Клавиша вызова или элемент управления "горячая клавиша" (hot key control) позволяет пользователю создавать клавиши вызова, т. е. такие комбинации клавиш, нажатие которых приведет к немедленному выполнению заданной команды. Данный элемент управления отображает пользовательский выбор и проверяет его допустимость. Класс библиотеки MFC CHotKeyCtrl предоставляет интерфейс с описываемым элементом управления.

Собственно комбинация клавиш определяется, во-первых, виртуальным кодом одной из алфавитно-цифровых клавиш, клавиш перемещения курсора и любых других, за исключением только трех клавиш — <Alt>, <Shift> и <Ctrl>, которые являются модификаторами и представляют собой вторую составляющую комбинации. Причем в комбинацию могут быть включены только одна из этих клавиш, любые две или все три.

Для использования описываемого элемента управления необходимо либо включить соответствующий элемент в шаблон блока диалога, либо создать с использованием функции Create непосредственно дочернее окно в любом другом. (Класс CHotKeyCtrl имеет конструктор без параметров, который просто создает объект класса, ничего более не предпринимая, т. е. никак не взаимодействует с операционной системой.)

BOOL CHotKeyCtrl::**Create**(
  DWORD dwStyle,
  const RECT & rect,
  CWnd* pParentWnd,
  UINT nID) —

возвращает TRUE в случае успешного создания и FALSE в противном случае. Параметр dwStyle содержит любую комбинацию стилей элементов управления. Параметр rect определяет местоположение и размер элемента управления. Параметр pParentWnd должен указывать на родительское окно, причем этот параметр не должен быть равен NULL, что разрешено для некоторых окон в ряде случаев. Параметр nID обеспечивает окно идентификатором, использование которого позволяет взаимодействовать с окном.

Примечание

Ряд классов не требует обязательного указания родительского окна создаваемого объекта. Например, это относится к классам блоков диалога, т. к. при выполнении соответствующая функция Create самостоятельно определяет родительское окно, выбирая, в случае необходимости, главное окно приложения (вызов функции AfxGetMainWnd) или главное системное окно (desktop window).

После создания элемента управления его можно и обычно нужно инициализировать некоторым значением. Специфика использования настраиваемых клавиш вызова требует сохранения информации о конкретной комбинации вне исполняемого модуля или библиотеки динамической компоновки. Это может быть INI-файл, запись в реестре или файл конфигурации, формат которого определяет приложение. Однако начальные (исходные) значения могут находиться как в исполняемом модуле (или DLL), так и в специальных конфигурационных файлах. Действительно, ограничений на изменение без ведома приложения и в INI-файлах, и в реестре практически нет даже в рамках Windows NT. Поскольку механизм сохранения какой бы то ни было информации не является предметом данного раздела, в сопровождающем его примере нет кода, обеспечивающего сохранение текущих назначений. Однако во время работы приложения-примера эти значения сохраняются прямо в памяти с использованием глобального массива.

Итак, для начальной инициализации служит функция SetHotKey:

void CHotKeyCtrl::**SetHotKey**(
WORD wVirtualKeyCode,
WORD wModifiers)

Прежде чем описывать параметры, представим и обратную функцию получения информации о выбранной комбинации:

void CHotKeyCtrl::**GetHotKey**(
WORD& wVirtualKeyCode,
WORD& wModifiers) —

первый параметр определяет виртуальный код клавиш (напомним, что такой клавишей может быть любая, за исключением трех клавиш-модификаторов <Alt>, <Shift> и <Ctrl>). Второй параметр как раз и определяет режим нажатия клавиш-модификаторов.

Обратите внимание, что оба параметра функции GetHotKey являются ссылками, что вполне естественно, т. к. они используются для получения информации.

Библиотека классов MFC определяет следующие флаги, для определения нажатия клавиш-модификаторов:

HOTKEYF_ALT	Клавиша <Alt>
HOTKEYF_CONTROL	Клавиша <Ctrl>
HOTKEYF_EXT	В качестве клавиши (помимо модификатора) была нажата одна из клавиш перемещения курсора
HOTKEYF_SHIFT	Клавиша <Shift>

В классе CHotKeyCtrl есть еще одна версия функции GetHotKey:

DWORD CHotKeyCtrl::**GetHotKey**( ) —

не принимает параметры, а наоборот, возвращает виртуальный код клавиши и модификатор, причем виртуальный код находится в младшем байте младшего слова, модификатор — в старшем байте того же слова, а для использования полученного значения можно воспользоваться следующим кодом, где переменная m_HotKey является объектом класса CHotKeyCtrl, связанным с соответствующим объектом Windows:

```
DWORD dwKeyAndShift = m_HotKey.GetHotKey();
m_HotKey.SetHotKey((LOBYTE(LOWORD(dwKeyAndShift)),
                    HIBYTE(LOWORD(dwKeyAndShift)));
```

Естественно, что далеко не все комбинации клавиш разрешены. Так, при выборе части комбинаций генерируется звуковой сигнал, что, однако, не мешает их назначению и использованию в качестве клавиш вызова. Win32 API и вместе с ним класс CHotKeyCtrl дают возможность заранее отсечь часть комбинаций, для этого служит функция SetRules, которая определяет ряд правил выбора комбинаций для последующего назначения их в качестве клавиш вызова:

void CHotKeyCtrl::**SetRules**(
WORD wInvalidComb,
WORD wModifiers) —

параметр wInvalidComb определяет правила, wModifiers позволяет указать флаги, определяющие клавиши-модификаторы, которые будут использованы вместо запрещенных в случае их нажатия.

Параметр wInvalidComb может принимать любую комбинацию следующих флагов, запрещая выбор тех или иных клавиш-модификаторов:

HKCOMB_A	Запрет выбора клавиши <Alt>
HKCOMB_C	Запрет выбора клавиши <Ctrl>
HKCOMB_CA	Запрет выбора клавиш <Ctrl> и <Alt> одновременно
HKCOMB_NONE	Запрет отсутствия выбора клавиш-модификаторов
HKCOMB_S	Запрет выбора клавиши <Shift>
HKCOMB_SA	Запрет выбора клавиш <Shift>+<Alt> одновременно
HKCOMB_SC	Запрет выбора клавиш <Shift>+<Ctrl> одновременно
HKCOMB_SCA	Запрет выбора клавиш <Shift>+<Ctrl>+<Alt> одновременно

К сожалению, ни элемент управления, ни класс CHotKeyCtrl не имеют средств к связыванию определенных действий с заданной комбинацией клавиш. Для этого придется воспользоваться или посылкой специального сообщения WM_SETHOTKEY или функцией Win32 API RegisterHotKey. Однако выбор того или иного способа обуславливается отнюдь не удобством — каждый из способов регистрирует в системе определенные типы сообщений для комбинаций клавиш.

Microsoft предлагает следующие наименования для типов клавиш вызова — *глобальные* и *поточно-ориентированные*. Рассмотрим эти типы более подробно.

Глобальные "горячие клавиши"

Глобальные клавиши вызова ассоциируются только с окнами верхнего уровня и позволяют их активизировать в любой момент. Для регистрации горячей клавиши такого типа необходимо послать сообщение WM_SETHOTKEY нужному окну. Активизация приложения происходит следующим образом — окну посылается сообщение WM_SYSCOMMAND с идентификатором сообщения SC_HOTKEY. Больше никакой дополнительной информации оконная процедура, обслуживающая сообщение WM_SYSCOMMAND, не получает. Таким образом, имеет смысл определить только одну комбинацию клавиш для создания горячей клавиши этого типа. Отметим, что система отслеживает назначенную горячую клавишу до тех пор, пока приложение, пославшее сообщение WM_SETHOTKEY, активно.

Поточно-ориентированные "горячие клавиши"

В отличие от глобальных, поточно-ориентированные клавиши вызова могут быть назначены для каждого интерактивного потока приложения, т. е. такого потока, который представлен окном. Для назначения горячей клавиши

необходимо воспользоваться функцией Win32 API RegisterHotKey. При нажатии заданной комбинации клавиш Windows помещает в начало очереди сообщений соответствующего потока сообщение WM_HOTKEY, которое также содержит полную информацию о кодах нажатия, и, что более важно, определенный пользователем идентификатор.

Важность функции RegisterHotKey для изложения материала требует приведения ее описания в этом разделе:

BOOL **RegisterHotKey**(
HWND hWnd,
int id,
UINT fsModifiers,
UINT vk) —

параметр hWnd определяет дескриптор окна, которому будет отправлено сообщение WM_HOTKEY, при значении этого параметра, равном NULL, сообщение будет направляться в очередь вызвавшего функцию потока. Остальные три параметра задают идентификатор и коды, определяющие комбинацию клавиш.

На значение идентификатора накладываются определенные ограничения: для исполняемых модулей значения идентификаторов должны лежать в пределах от 0x0000 и до 0xBFFF, а код, находящийся в динамически подключаемых библиотеках (DLL), должен определять значения идентификаторов в пределах от 0xC000 и до 0xFFFF.

Примечание

Это ограничение связано с тем, что для исключения конфликтов значение идентификатора следует определять при помощи функции Win32 API GlobalAddAtom, которая как раз и возвращает значения в этих пределах.

В случае, если задаваемая комбинация клавиш имеет тот же идентификатор, что и заданная ранее, последнее значение заменяется новым. Для отмены горячей клавиши необходимо воспользоваться функцией UnregisterHotKey.

Функция RegisterHotKey принимает в качестве флагов, определяющих клавиши-модификаторы, значения, отличные от тех, что получает приложение в результате выполнения функции CHotKeyCtrl::GetHotKey. Поэтому перед регистрацией горячей клавиши необходимо провести переопределение. Флаги, используемые в функции RegisterHotKey:

MOD_ALT	Клавиша <Alt>
MOD_CONTROL	Клавиша <Ctrl>
MOD_SHIFT	Клавиша <Shift>

Необходимо отметить, что и глобальные и поточно-ориентированные клавиши вызова имеют приоритет перед комбинациями, заданными в качестве сокращений (акселераторов, accelerators) команд.

Пример использования класса CHotKeyCtrl

В данном примере есть возможность назначать пять клавиш вызова, из которых четыре являются поточно-зависимыми, а одна — глобальной. Все назначения происходят в функциях-членах класса CHotKeyDlg, создающего блок диалога. Последний выводится на экран по команде IDM_HOTKEY, которая добавлена в системное меню, т. к. само приложение меню не имеет. Обратите внимание, что код этой команды должен лежать в определенных пределах.

Для хранения начальных значений комбинаций клавиш и соответствующих команд, а также для хранения новых устанавливаемых значений комбинаций клавиш используется глобальный массив gHotKey. При каждом создании блока диалога (функция CHotKeyDlg::OnInitDialog) все пять элементов управления инициализируются последними значениями комбинаций клавиш, а при завершении диалога, при подтверждении введенных значений (функция CHotKeyDlg::OnOK) будет обновлено содержимое массива, а также зарегистрированы все пять клавиш вызова.

Как для создания блока диалога в ответ на команду, так и для обработки команды SC_HOTKEY от глобальной горячей клавиши, класс окна CHotKeyFrame содержит обработчик сообщения WM_SYSCOMMAND. В коде, обрабатывающем данную горячую клавишу, предусмотрен вывод текущего времени по центру окна.

Для обработки команд, связанных с поточно-ориентированными клавишами вызова, класс окна имеет функцию CHotKeyFrame::OnHotKey), в которой в соответствии с полученным идентификатором производятся манипуляции с главным окном приложения — оно может быть свернуто (минимизировано), развернуто (максимизировано), восстановлено и спрятано.

```
// Файл HotKeyFrm.cpp
// Код команды вызова диалога для назначения клавиш вызова
#define IDM_HOTKEYBOX  0x0010
// Массив структур для хранения комбинаций клавиш
// и идентификаторов команд, связанных с ними. Первые четыре
// клавиши вызова являются поточно-ориентированными,
// а последняя — глобальной и соответственно не имеет
// связанного с ней идентификатора
static struct {
  WORD m_wVK;
  WORD m_wMod;
  UINT m_nCmd;
} gHotKey[5] =
{{VK_F1, HOTKEYF_SHIFT | HOTKEYF_CONTROL, SW_MINIMIZE},
```

```
 {VK_F2, HOTKEYF_SHIFT | HOTKEYF_CONTROL, SW_MAXIMIZE},
 {VK_F3, HOTKEYF_SHIFT | HOTKEYF_CONTROL, SW_SHOWNORMAL},
 {VK_F4, HOTKEYF_SHIFT | HOTKEYF_CONTROL, SW_HIDE},
 {VK_F5, HOTKEYF_SHIFT | HOTKEYF_CONTROL}
};
// функция перевода кодов флагов для использования
// в функции RegisterHotKey
UINT TransModifiers(WORD wMod)
{
  UINT uMod = 0;
  if (wMod & HOTKEYF_ALT)        uMod |= MOD_ALT;
  if (wMod & HOTKEYF_SHIFT)      uMod |= MOD_SHIFT;
  if (wMod & HOTKEYF_CONTROL)    uMod |= MOD_CONTROL;
  return uMod;
}
// CHotKeyDlg — блок диалога "Клавиши вызова"
class CHotKeyDlg : public CDialog
{
public:
  CHotKeyDlg();
  //{{AFX_DATA(CHotKeyDlg)
    enum { IDD = IDD_HOTKEYBOX };
    CHotKeyCtrl m_Minimize;
    CHotKeyCtrl m_Maximize;
    CHotKeyCtrl m_ShowNormal;
    CHotKeyCtrl m_Hide;
    CHotKeyCtrl m_DrawTime;
    //}}AFX_DATA
    //{{AFX_VIRTUAL(CHotKeyDlg)
    protected:
    // DDX/DDV support
    virtual void DoDataExchange(CDataExchange* pDX);
    //}}AFX_VIRTUAL
protected:
  //{{AFX_MSG(CHotKeyDlg)
  virtual void OnOK();
  virtual BOOL OnInitDialog();
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()
```

```
};
CHotKeyDlg::CHotKeyDlg() : CDialog(CHotKeyDlg::IDD){}
void CHotKeyDlg::DoDataExchange(CDataExchange* pDX)
{
  CDialog::DoDataExchange(pDX);
  //{{AFX_DATA_MAP(CHotKeyDlg)
  DDX_Control(pDX, IDC_MINIMIZE, m_Minimize);
  DDX_Control(pDX, IDC_MAXIMIZE, m_Maximize);
  DDX_Control(pDX, IDC_SHOWNORMAL, m_ShowNormal);
  DDX_Control(pDX, IDC_HIDE, m_Hide);
  DDX_Control(pDX, IDC_DRAWTIME, m_DrawTime);
  //}}AFX_DATA_MAP
}
BEGIN_MESSAGE_MAP(CHotKeyDlg, CDialog)
  //{{AFX_MSG_MAP(CHotKeyDlg)
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()
//  Функция начальной инициализации элементов управления
BOOL CHotKeyDlg::OnInitDialog()
{
  CDialog::OnInitDialog();
  m_Minimize.SetHotKey(gHotKey[0].m_wVK, gHotKey[0].m_wMod);
  m_Maximize.SetHotKey(gHotKey[1].m_wVK, gHotKey[1].m_wMod);
  m_ShowNormal.SetHotKey(gHotKey[2].m_wVK, gHotKey[2].m_wMod);
  m_Hide.SetHotKey(gHotKey[3].m_wVK, gHotKey[3].m_wMod);
  m_DrawTime.SetHotKey(gHotKey[4].m_wVK, gHotKey[4].m_wMod);
  return TRUE;
}
// В случае подтверждения выбора запоминаем новые значения для кодов
// комбинаций клавиш и регистрируем клавиши вызова
void CHotKeyDlg::OnOK()
{
  m_Minimize.GetHotKey(gHotKey[0].m_wVK, gHotKey[0].m_wMod);
  RegisterHotKey(AfxGetMainWnd()->m_hWnd, 0,
    TransModifiers(gHotKey[0].m_wMod), gHotKey[0].m_wVK);
  m_Maximize.GetHotKey(gHotKey[1].m_wVK, gHotKey[1].m_wMod);
  RegisterHotKey(AfxGetMainWnd()->m_hWnd, 1,
    TransModifiers(gHotKey[1].m_wMod), gHotKey[1].m_wVK);
  m_ShowNormal.GetHotKey(gHotKey[2].m_wVK, gHotKey[2].m_wMod);
```

```
  RegisterHotKey(AfxGetMainWnd()->m_hWnd, 2,
    TransModifiers(gHotKey[2].m_wMod), gHotKey[2].m_wVK);
  m_Hide.GetHotKey(gHotKey[3].m_wVK, gHotKey[3].m_wMod);
  RegisterHotKey(AfxGetMainWnd()->m_hWnd, 3,
    TransModifiers(gHotKey[3].m_wMod), gHotKey[3].m_wVK);
  m_DrawTime.GetHotKey(gHotKey[4].m_wVK, gHotKey[4].m_wMod);
  DWORD dwKeyAndShift = m_DrawTime.GetHotKey();
  AfxGetMainWnd()->SendMessage(WM_SETHOTKEY, dwKeyAndShift);
  CDialog::OnOK();
}
// CHotKeyFrame
BEGIN_MESSAGE_MAP(CHotKeyFrame, CFrameWnd)
  //{{AFX_MSG_MAP(CHotKeyFrame)
  ON_WM_PAINT()
  ON_WM_SYSCOMMAND()
  ON_WM_CREATE()
  ON_MESSAGE(WM_HOTKEY, OnHotKey)
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()
void CHotKeyFrame::OnPaint()
{
  CPaintDC dc(this);
}
int CHotKeyFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
  if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
    return -1;
  // Добавляем пункт в системное меню
  CMenu* pSysMenu = GetSystemMenu(FALSE);
  // IDM_HOTKEYBOX должен находиться в заданных пределах
  ASSERT((IDM_HOTKEYBOX & 0xFFF0) == IDM_HOTKEYBOX);
  ASSERT(IDM_HOTKEYBOX < 0xF000);
  pSysMenu->AppendMenu(MF_STRING, IDM_HOTKEYBOX, _T("Клавиши вызова"));
  return 0;
}
// Функция обработки сообщения WM_SYSCOMMAND
void CHotKeyFrame::OnSysCommand(UINT nID, LPARAM lParam)
{
  if ((nID & 0xFFF0) == IDM_HOTKEYBOX)  {
```

```
    CHotKeyDlg dlgHotKey;
    dlgHotKey.DoModal();
  }  else if (nID == SC_HOTKEY)  {
    if (!IsIconic())    {
      InvalidateRect(NULL);
      UpdateWindow();
      CClientDC dc(this);
      CRect rect;
      GetClientRect(rect);
      dc.SetBkMode(TRANSPARENT);
      dc.DrawText(CTime::GetCurrentTime().Format(_T("%H:%M:%S")),
        &rect, DT_CENTER | DT_VCENTER | DT_SINGLELINE);
    }
  }  else
    CWnd::OnSysCommand(nID, lParam);
}
// Функция обработки сообщения WM_HOTKEY
LONG CHotKeyFrame::OnHotKey(UINT nID, LPARAM lParam)
{
  if (0 <= nID && nID < 5)
    ShowWindow(gHotKey[nID].m_nCmd);
  return 0;
}
```

Рис. 25.3. Блок диалога настройки "горячих клавиш" в главном окне приложения HotKey

Индикатор

Индикатор — это дочернее окно, которое можно использовать в приложении для индикации процесса выполнения некоторой длительной операции. Такое окно представляет собой вытянутую и расположенную горизонтально, а теперь и вертикально, прямоугольную область (полосу). По мере выполнения операции содержимое области заполняется слева направо прямоугольниками, цвет которых соответствует системному цвету, используемому для подсвечивания надписей или иных объектов.

В отличие от большинства других элементов управления, цвет которых можно задать в ответ на соответствующий запрос, цвет индикатора определяется системными установками и изменению не подлежит.

Большинство программ индикацию реализуют двумя способами: добавляют элемент индикации в блок диалога, либо используют его в специально созданных панелях диалога, которые были описаны в главе 11. И тот и другой способ подразумевает использование шаблонов блока диалога, что исключает необходимость функции создания окна индикатора. Библиотека классов MFC имеет в своем составе класс CProgressCtrl, который обеспечивает интерфейс между приложением и объектом Windows "индикатор". Этот класс имеет конструктор, задача которого — создание объекта класса, собственно же элемент индикации может быть создан при помощи функции Create (использование которой не актуально при задании элемента в шаблоне блока диалога):

BOOL CProgressCtrl::**Create**(
DWORD dwStyle,
const RECT & rect,
CWnd* pParentWnd,
UINT nID) —

параметры этой функции полностью совпадают с рассмотренными ранее для функции CHotKeyCtrl::Create, за исключением двух новых, специфичных для данного элемента управления, стилей:

PBS_VERTICAL	Индикатор располагается вертикально — "сверху вниз". Без этого флага расположение стандартное — "слева направо"
PBS_SMOOTH	Область индикации заполняется непрерывно. Без этого флага заполнение осуществляется блоками

Реализация процесса и его взаимосвязи с индикацией — полностью задача разработчика приложения. Собственно индикатор является лишь средством отображения отношения двух чисел. Первое определяется величиной некоторого интервала, а второе — смещением от начала интервала. Границы интервала лежат в пределах от –217 483 648 до 217 483 647 включительно.

А смещение определяется заданным текущим абсолютным значением и нижней границей интервала. Возможность задания интервала и смещения его нижней границы относительно нуля может обеспечить некоторую связь с природой отслеживаемого процесса, исключая дополнительный код на преобразование.

Для задания интервала класс CProgressCtrl теперь имеет две функции SetRange и SetRange32:

void CProgressCtrl::**SetRange**(
short nLower,
short nUpper)

и

void CProgressCtrl::**SetRange32**(
int nLower,
int nUpper) —

параметры nLower и nUpper определяют его нижнюю и верхнюю границы соответственно.

Текущее значение индикатора может быть установлено тремя различными способами. Во-первых, можно непосредственно задать текущее значение:

int CProgressCtrl::**SetPos**(int nPos) —

эта функция не только задает текущее значение (параметр nPos), но и возвращает предыдущее значение индикатора. При неверном задании текущего значения (вне интервала) новое значение будет приведено к значению ближайшей границы интервала.

Во-вторых, можно осуществить ранее заданное смещение (шаг) относительно текущего значения при помощи функции StepIt:

int CProgressCtrl::**StepIt**( ) —

эта функция также возвращает предыдущее значение индикатора. Причем в случае, если в результате добавления к текущему значению величины шага значение индикатора выйдет за пределы интервала, новое значение индикатора будет возвращено в заданные пределы.

Шаг задается функцией SetStep:

int CProgressCtrl::**SetStep**(int nStep) —

шаг задается параметром nStep, сама функция возвращает предыдущее значение шага. Значение шага может быть отрицательным, что позволяет не только увеличивать значение индикатора, но и уменьшать его.

В-третьих, текущее значение устанавливается непосредственно заданным смещением при помощи функции OffsetPos:

int CProgressCtrl::**OffsetPos**(int nPos) —

параметр nPos определяет смещение относительно текущего значения. Сама функция, как и все остальные, изменяющие текущее значение, возвращает его предыдущее значение. При выходе за пределы интервала, в отличие от функции StepIt, выполнение

функции OffsetPos приведет к тому, что текущим значением интервала будет либо нижняя, либо верхняя границы в зависимости от значения смещения.

При выполнении любой функции, в результате которой может измениться отношение величины интервала и текущего значения (SetRange, SetPos, StepIt, OffsetPos) происходит автоматическая перерисовка индикатора.

Однако уже при создании (с использованием функции Create или во время создания блока диалога) индикатор инициализируется следующими начальными значениями (по умолчанию):

- Текущее значение 0
- Интервал в пределах от 0 до 100
- Шаг приращения 10

Узнать текущий диапазон и положение индикатора можно при помощи функций GetRange и GetPos соответственно.

Использование индикатора в панели диалога

Суть следующего примера в том, что индикатор размещен в панели диалога, созданной при помощи класса CDialogBar, о котором было рассказано в главе 11. При создании объекта CInProgFrame кроме панели диалога создается строка состояния, которая заменяется время от времени панелью с индикатором. Для этого создаются два таймера. Задача первого — имитация начала и окончания некоторой операции, второго — имитация собственно операции. Обработчик сообщений от таймера либо переключает панели, либо изменяет значение индикатора. Для наглядности в код "начала операции" добавлены функции начальной инициализации индикатора.

```
// Файл InProgFrm.cpp
// Идентификаторы таймеров
#define ID_PULSE   1
#define ID_INPROG  2
// Массив настроек строки состояния
static UINT BASED_CODE indicators[] =
{
  ID_SEPARATOR,
  ID_INDICATOR_NUM,
  ID_INDICATOR_CAPS,
  ID_INDICATOR_OVR
};
// CInProgFrame
// Переопределяем функцию PreCreateWindow для задания
// своего класса окна, задавая иной фон и отказываясь от стилей
```

```
// CS_VREDRAW/CS_HREDRAW, чтобы исключить
// перерисовку панелей элементов управления при изменении
// размеров главного окна
BOOL CInProgFrame::PreCreateWindow(CREATESTRUCT& cs)
{
  cs.lpszClass = AfxRegisterWndClass(CS_DBLCLKS,
    AfxGetApp()->LoadStandardCursor(IDC_ARROW),
    (HBRUSH)(COLOR_WINDOW+1),
    AfxGetApp()->LoadIcon(IDR_INPROG));
  return TRUE;
}
int CInProgFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
  if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
    return -1;
  // Создаем строку состояния
  if (!m_wndStatusBar.Create(this) ||
      !m_wndStatusBar.SetIndicators(indicators,
      sizeof(indicators)/sizeof(UINT)))
  {
    TRACE0("Ошибка при создании панели \n");
    return FALSE;
  }
  // Скрываем строку состояния — на ее месте будет панель диалога
  ShowControlBar(&m_wndStatusBar, FALSE, FALSE);
  // Создаем панель диалога
  if (!m_wndDlgBar.Create(this, IDD_PROGRESS,
    CBRS_BOTTOM, IDD_PROGRESS))
  {
    TRACE0("Ошибка при создании панели \n");
    return -1;
  }
  // Разрешаем привязку панелей элементов управления
  EnableDocking(CBRS_ALIGN_BOTTOM);
  // Получаем указатель на объект класса CProgressCtrl
  // по его идентификатору в шаблоне блока диалога
  CProgressCtrl *pProgress =
    (CProgressCtrl *)m_wndDlgBar.GetDlgItem(IDC_PROGRESS);
  // Инициализируем индикатор
```

```
  pProgress->SetRange(0, 1000);
  pProgress->SetPos(0);
  pProgress->SetStep(10);
  // Создаем два таймера для имитации длительной операции
  SetTimer(ID_PULSE, 10000, NULL);
  SetTimer(ID_INPROG,  100, NULL);
  return 0;
}
// Функция-обработчик сообщений от таймера
void CInProgFrame::OnTimer(UINT nIDEvent)
{
  // Статическая переменная-переключатель или флаг режима
  static int nCheck = 1;
  CProgressCtrl *pProgress =
    (CProgressCtrl *)m_wndDlgBar.GetDlgItem(IDC_PROGRESS);
  if (nIDEvent == ID_PULSE)
  {
    if (nCheck = !nCheck)
    {
      // Начало "операции"
      pProgress->SetRange(0, 1000);
      pProgress->SetPos(0);
      pProgress->SetStep(10);
      ShowControlBar(&m_wndStatusBar, FALSE, FALSE);
      ShowControlBar(&m_wndDlgBar, TRUE, FALSE);
    }
    else
    {
      // Завершение "операции"
      ShowControlBar(&m_wndDlgBar, FALSE, FALSE);
      ShowControlBar(&m_wndStatusBar, TRUE, FALSE);
    }
  }
  else if (nIDEvent == ID_INPROG && nCheck)
  {
    pProgress->StepIt();// "Операция" осуществляется
  }
  CFrameWnd::OnTimer(nIDEvent);
}
```

Рис. 25.4. Окно приложения InProg с индикатором в панели диалога

Линейка с ползунком

Линейка с ползунком (SLIDER или TRACKBAR) представляет собой окно с линейкой, ползунком на ней и, возможно, метками. Кроме того, можно выделить любой диапазон в рамках размера линейки, которая может иметь как горизонтальную, так и вертикальную ориентацию. Когда пользователь перемещает ползунок при помощи мыши или используя клавиши управления курсором, этот элемент управления посылает уведомляющие сообщения, позволяющие отслеживать производимые изменения.

Этот элемент управления предназначен для того, чтобы пользователи могли выбирать дискретные значения в заданном диапазоне. Вид элемента позволяет использовать его, например, для запуска какой-либо операции, в качестве, если хотите, рубильника.

Библиотека MFC предоставляет класс CSliderCtrl, который инкапсулирует возможности линейки с ползунком Windows, упрощая управление этим элементом.

Несмотря на явно аналоговое представление значений, ползунок может принимать значения из заданного целочисленного диапазона. После его создания, в рамках блока диалога или в другом окне, можно задать ряд параметров, для чего класс предоставляет большое количество функций. Но об этом позже. Сначала опишем, какие уведомления и в каких случаях посылает данный элемент управления.

Уведомления

Независимо от того, как ориентирован элемент управления — горизонтально или вертикально, — уведомления он посылает родительскому окну в

рамках сообщения WM_HSCROLL. Для обработки сообщений от линейки с ползунком следует перегрузить функцию класса родительского окна OnHScroll. Эта функция получает код уведомления и указатель на класс CSliderCtrl. Следует иметь в виду, что соответствующий параметр функции имеет тип указателя на класс CScrollCtrl, поэтому для вызова функций класса CSliderCtrl следует произвести преобразование типа.

Все уведомляющие сообщения можно разделить на следующие группы. Сообщения TB_BOTTOM, TB_LINEDOWN, TB_LINEUP и TB_TOP посылаются только при взаимодействии пользователя с элементом управления с помощью клавиатуры, т. е. клавиш перемещения курсора. Сообщения TB_THUMBPOSITION и TB_THUMBTRACK посылаются при действиях мышью. А сообщения TB_ENDTRACK, TB_PAGEDOWN и TB_PAGEUP посылаются в обоих случаях.

В таблице перечислены все уведомления и соответствующие им виртуальные клавиши, вызывающие эти сообщения:

TB_BOTTOM	VK_END — ползунок переводится в самое правое или самое нижнее положение
TB_ENDTRACK	WM_KEYUP — завершение перемещения ползунка
TB_LINEDOWN	VK_RIGHT или VK_DOWN — перемещение ползунка на одну позицию вправо (вниз). По умолчанию линейка имеет 101 позицию — от 0 до 100
TB_LINEUP	VK_LEFT или VK_UP — перемещение на одну позицию влево (вверх)
TB_PAGEDOWN	VK_NEXT — перемещение на одну "страницу" вправо (вниз) с помощью клавиши <Page Down> или мыши. По умолчанию линейка разбита на 5 "страниц" шириной в 20 позиций
TB_PAGEUP	VK_PRIOR — перемещение на одну "страницу" влево (вверх) с помощью клавиши <Page Up> или мыши
TB_THUMBPOSITION	WM_LBUTTONUP — текущее положение ползунка после завершения его перемещения мышью; следует за сообщением TB_THUMBTRACK
TB_THUMBTRACK	Ползунок перемещался мышью
TB_TOP	VK_HOME — ползунок переводится в самое левое (самое верхнее) положение

Создание элемента управления

Как обычно, конструктор класса создает только объект CSliderCtrl, и для создания элемента управления следует вызвать функцию Create:

BOOL CSliderCtrl::**Create**(
DWORD dwStyle,
const RECT& rect,
CWnd* pParentWnd,
UINT nID) —

параметр dwStyle может содержать любые комбинации стилей элементов управления, включая следующие, свойственные рассматриваемому:

TBS_HORZ	Линейка имеет горизонтальную ориентацию (принята по умолчанию)
TBS_VERT	Линейка имеет вертикальную ориентацию
TBS_AUTOTICKS	Линейка имеет метки для всех значений в заданном диапазоне значений; без этого стиля линейка может иметь метки только для начального и конечного положения ползунка или там, где они заданы функциями SetTic и SetTicFreq; метки добавляются автоматически, поэтому при наличии данного стиля нельзя использовать функции SetTic и SetTicFreq, для удаления меток можно воспользоваться функцией ClearTics; метки устанавливаются сразу же после вызова функции SetRange
TBS_NOTICKS	Исключает отображение каких-либо меток, в том числе и для начального и конечного положений ползунка
TBS_BOTTOM	Линейка имеет метку для конечного положения ползунка
TBS_TOP	Линейка имеет метку для начального положения ползунка
TBS_RIGHT	Элемент отображает метки справа (снизу) от линейки, ползунок направлен в ту же сторону
TBS_LEFT	Элемент отображает метки слева (сверху) от линейки, ползунок направлен в ту же сторону
TBS_BOTH	Элемент отображает метки с обеих сторон, ползунок имеет прямоугольную форму, т. е. он фактически не направлен ни в одну из сторон
TBS_ENABLESELRANGE	Элемент отображает "допустимый" диапазон выбора; при этом линейка имеет внутреннюю область большего, чем обычно, размера, в которой заданный диапазон имеет цвет, отличный от цвета линейки; начало и конец диапазона ограничены метками треугольной формы
TBS_FIXEDLENGTH	Размер ползунка элемента управления может быть изменен
TBS_NOTHUMB	Ползунок элемента управления не отображается
TBS_TOOLTIPS	Элемент управления поддерживает всплывающие подсказки. При создании линейки с этим стилем автоматически создается элемент управления "всплывающая подсказка", который по умолчанию отображает текущую позицию ползунка

Параметр rect задает размер и расположение линейки. Может быть объектом класса CRect или структурой RECT, в случае, если заданный размер меньше, чем требуется для отображения линейки и ползунка, они не масштабируются — часть элемента управления будет скрыта. Параметр pParentWnd задает родительское окно (обычно это CDialog). Значение этого параметра не должно быть равно NULL. Параметр nID задает идентификатор линейки.

Параметры элемента управления

Кроме задания стилей, можно получать текущие значения установок, а также изменять (настраивать) следующие значения (напомним, что по умолчанию линейка имеет 101 позицию и пять "страниц"):

- Общий диапазон (максимальное и минимальное положение ползунка). Для этого можно воспользоваться функциями GetRangeMax, SetRangeMax, GetRangeMin, SetRangeMin, GetRange и SetRange
- Величину единичных перемещений, уведомления о которых производится сообщениями TB_LINEUP и TB_LINEDOWN (функции GetLineSize и SetLineSize)
- Величину страничных перемещений, уведомления о которых производится сообщениями TB_PAGEUP и TB_PAGEDOWN (функции GetPageSize и SetPageSize)
- "Допустимый" диапазон (его максимальное и минимальное значения). Для этого можно воспользоваться функциями GetSelection, SetSelection. Для того чтобы отказаться от отображения "допустимого" диапазона, следует вызвать функцию ClearSel
- Размер элемента управления (функция GetChannelRect) и размер ползунка (функция GetThumbRect)
- Текст всплывающей подсказки (функции GetToolTips и SetToolTips)

Примечание

Некоторые из функций настроек по умолчанию после изменения параметров не допускают перерисовки линейки в соответствии с новыми значениями. Это может привести к неожиданным результатам, т. к. даже изменение положения ползунка приводит лишь к перерисовке области, непосредственно располагающейся под ползунком. Таким образом, отказ от перерисовки при смене сразу нескольких параметров следует завершать сообщением о необходимости обновления элемента управления (например, вызовом функции InvalidateRect).

Отдельно рассмотрим функции, управляющие отображением меток и положением ползунка.

Положение меток

Для получения количества меток следует воспользоваться функцией GetNumTics.

Положение метки лежит в общем диапазоне значений. Если стиль элемента управления TBS_AUTOTICKS, количество меток определяется общим диапазоном, несмотря на то что будущее количество видимых меток определяется функцией SetTicFreq.

Вообще задание этого стиля приводит к тому, что для определения положения следует не пользоваться следующими функциями, а вычислять его.

Для получения положения всех меток класс имеет функцию:

DWORD* CSliderCtrl::**GetTicArray**( ) —

возвращает указатель на массив, содержащий положения всех меток элемента управления.

Для получения положения конкретной метки по ее номеру, считая слева или сверху, служит функция:

int CSliderCtrl::**GetTic**(int nTic) —

возвращает значение –1 в случае, если параметр nTic содержит несуществующий номер.

А установить положение метки позволяет функция SetTic.

Для получения позиции метки в единицах измерения клиентской области родительского окна следует пользоваться функцией GetTicPos.

При задании стиля TBS_AUTOTICKS частота отображения меток определяется функцией:

void CSliderCtrl::**SetTicFreq**(int nFreq) —

параметр nFreq сообщает, как часто следует отображать метку. Значение 10, например, говорит о том, что отображению подлежит каждая десятая метка, считая с нулевой. (Крайняя нижняя или правая метка также будет отображена).

Для удаления меток служит функция ClearTics.

Положение ползунка

Для получения текущего положения ползунка, а также программной установки его положения следует использовать функции GetPos и SetPos соответственно.

Класс CSliderCtrl имеет в своем составе очень полезную функцию, позволяющую проверять допустимость положения ползунка, ранее установленного при помощи функции SetPos:

void CSliderCtrl::**VerifyPos**( ) —

не возвращает какое-либо значение, позволяющее определить допустимость заданного положения (причем, заданного отнюдь не параметром этой функции!). Работа функции заключается в том, что в случае, когда текущее положение лежит вне общего диапазона, она устанавливает ползунок в крайнее левое (верхнее) или правое (нижнее) положение, в зависимости от того, к какому положению ближе заданное. Этой функцией ни в коем случае *нельзя* пользоваться в обработчике уведомляющих сообщений!

Счетчик

Элемент управления "счетчик" (SPIN BUTTON или UP-DOWN) представляет собой пару кнопок со стрелками, назначение которых — в смене значения, например, уменьшение либо увеличение числового значения. Счетчик похож на вырожденную полосу

прокрутки (SCROLL BAR) и действует примерно так же. Кроме того, этот элемент управления можно ассоциировать с другим элементом управления так, что его содержимое будет меняться под управлением спина. Такие ассоциированные элементы управления называются *окнами сопровождения* (BUDDY WINDOW).

Если счетчик ассоциирован с другим окном (элементом управления), для пользователя они выглядят как единый элемент управления. Действительно, счетчик может автоматически позиционировать себя с другим элементом управления, располагающимся прямо перед ним, например, в шаблоне блока диалога, и самостоятельно устанавливать заголовок этого элемента.

В случае, когда счетчик меняет числовые значения, нажатие верхней кнопки изменяет значение в сторону максимума, а нижней — в сторону минимума. Максимумом по умолчанию является значение 0, минимумом — значение 100.

Не имея ассоциированного с ним окна, счетчик функционирует как упрощенный вариант полосы прокрутки. Например, элемент управления, реализующий вкладки (TAB), использует счетчик для осуществления доступа к дополнительным вкладкам.

Уведомления

Счетчик может посылать до четырех различных сообщений для одной ориентации. Первые из которых — уведомляющие. Все они посылаются родительскому окну и никак не воздействуют на ассоциированное со счетчиком окно. Эти три сообщения взаимосвязаны.

Первое сообщение, UDN_DELTAPOS, посылается родительскому окну при нажатии одной из кнопок со стрелками для того, чтобы дать возможность приложению до осуществления модификации содержимого ассоциированного со счетчиком окна предпринять какие-либо действия. Например, проверить допустимость изменений, отменить или разрешить их. Для того чтобы отменить изменения в ассоциированном окне, следует в обработчике этого сообщения установить по указателю pResult значение TRUE. И наоборот — значение FALSE разрешит произвести изменения:

```
BEGIN_MESSAGE_MAP(CSunDlg, CDialog)
  //{{AFX_MSG_MAP(CSunDlg)
  ...
  ON_NOTIFY(UDN_DELTAPOS, IDC_SPIN, OnDeltaPos)
  ...
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()
void CSunDlg::OnDeltaPos(NMHDR* pNMHDR, LRESULT* pResult)
{
```

```
  NM_UPDOWN* pNMUpDown = (NM_UPDOWN*)pNMHDR;
  // Меняем значения в два раза быстрее, чем задано
  pNMUpDown->iDelta *= 2;
  // В случае, если ожидаемое значение
  // будет больше 10, - запрещаем изменение
  if (pNMUpDown->iPos + pNMUpDown->iDelta > 10)
    *pResult = TRUE;  // Запрет изменения
  else
    *pResult = FALSE; // Разрешение изменения
}
```

Как видно из примера, обработчик получает указатель на структуру NM_UPDOWN, которая содержит необходимую информацию о возможных изменениях:

```
typedef struct _NM_UPDOWN {
  NMHDR hdr;
  int iPos;
  int iDelta;
} NM_UPDOWNW
```

Первый параметр структуры содержит заголовок. Он требуется для всех уведомляющих сообщений. Заголовок содержит дескриптор и идентификатор окна, пославшего сообщение, а также код самого сообщения. Параметр iPos содержит текущее значение счетчика, а параметр iDelta — возможное изменение текущего положения. Оба параметра числовые и могут быть отрицательными.

Возникает вопрос: что делать, если ассоциированное окно содержит нечисловые значения, или ассоциированного окна нет, ведь изменение значения счетчика должно быть связано с выполнением некоторых действий?

Для ответа посмотрим, что же происходит после обработки сообщения UDN_DELTAPOS. В случае, если разрешение изменения получено (а по умолчанию именно так и происходит), счетчик посылает одно из сообщений, аналогичных сообщениям от полосы прокрутки: WM_HSCROLL или WM_VSCROLL. Тип сообщения зависит от того, как ориентирован счетчик, вертикально или горизонтально. В обоих случаях обработчики получают всю необходимую информацию.

```
void CSunDlg::OnVScroll(
  UINT nSBCode, UINT nPos, CScrollBar* pScrollBar)
{
  // Обрабатываем только сообщения от счетчика
  if (pScrollBar->m_hWnd == GetDlgItem(IDC_SPIN)->m_hWnd)
  {
    // Нас интересует завершение "прокрутки" счетчика
```

```
    if (nSBCode == SB_ENDSCROLL)
    {
      // Здесь выполняются необходимые действия,
      // связанные с изменением значения счетчика.
      // Текущее значение счетчика содержит
      // параметр nPos
    }
  }
  CDialog::OnVScroll(nSBCode, nPos, pScrollBar);
}
```

Автоматическое изменение

При наличии ассоциированного окна счетчик может самостоятельно изменять его содержимое только в одном случае. Когда счетчик создан для изменения целочисленных значений, при нажатии одной из кнопок со стрелкой счетчик посылает ассоциированному с ним окну сообщения WM_GETTEXT и WM_SETTEXT.

Первое сообщение используется для того, чтобы определить текущее значение содержимого окна, второе — для его изменения. В качестве значения нового содержимого окна выступает строка, содержащая символьное представление числа — текущего значения счетчика. Такой подход позволяет синхронизировать содержимое окна и значения счетчика, т. е. позволяет изменять содержимое ассоциированного окна и другими средствами помимо счетчика — программно или вручную пользователем.

А что будет, если пользователь ввел строку текста, которая вовсе не является представлением числа?

В этом случае в качестве текущего значения, которое будет увеличено или уменьшено, выступает *текущее значение* счетчика — окно ведь не содержит числа. Реализация, пожалуй, довольно логична.

Из вышесказанного следует, что связывать счетчик можно только с такими типами окон, которые изменяют свое содержимое при получении сообщения WM_SETTEXT. Однако это не совсем так — любому окну Windows можно посылать любые сообщения, следовательно, и любое окно может быть связано со счетчиком. Другое дело, что содержимое его может не меняться. Например, кнопки, текстовые поля, статические элементы откликаются на воздействие со стороны счетчика, а другие, такие как индикатор (PROGRESS BAR), никак не реагируют.

Создание счетчика

Конструктор класса создает только объект CSpinButtonCtrl, для создания элемента управления следует вызвать функцию:

BOOL CSpinButtonCtrl::**Create**(
DWORD dwStyle,
const RECT & rect,
CWnd* pParentWnd,
UINT nID) —

параметр *dwStyle* может принимать любые комбинации стилей элементов управления плюс следующие, свойственные рассматриваемому элементу управления:

UDS_HORZ	Горизонтальная ориентация счетчика (по умолчанию — вертикальная)
UDS_WRAP	Поддерживается переход через граничные значения; в этом случае после максимального значения текущим становится минимальное и наоборот
UDS_ARROWKEYS	Поддерживается управление счетчиком с помощью клавиш перемещения курсора (<↑> и <↓>, независимо от ориентации элемента управления)
UDS_SETBUDDYINT	В качестве значений содержимого ассоциированного окна используются целочисленные значения — изменение содержимого ассоциированного окна происходит автоматически средствами счетчика (используются сообщения WM_GETTEXT и WM_SETTEXT)
UDS_NOTHOUSANDS	Содержимое ассоциированного окна не имеет разделителя тысяч между каждыми тремя цифрами
UDS_AUTOBUDDY	Автоматический выбор в качестве ассоциированного окна ближайшего предыдущего в списке созданных, например, в шаблоне блока диалога
UDS_ALIGNRIGHT	Счетчик располагается в правой части ассоциированного окна, уменьшая тем самым его клиентскую область
UDS_ALIGNLEFT	Счетчик располагается в левой части ассоциированного окна, уменьшая тем самым его клиентскую область

Параметр rect задает размер и расположение счетчика. Может быть объектом класса CRect или структурой RECT. Параметр pParentWnd задает родительское окно (обычно это CDialog). Значение этого параметра не должно быть равно NULL. Параметр nID задает идентификатор текстового поля.

Параметры элемента управления

В случае, если счетчик не имеет стиля UDS_AUTOBUDDY, класс предоставляет возможность связывания с ним окна (функция SetBuddy возвращает указатель, а объект класса CWnd — окно, которое было ранее ассоциировано со счетчиком). В то же время можно получить указатель на объект класса CWnd — окно, которое в настоящий момент ассоциировано со счетчиком

(функция GetBuddy). Указатель может указывать на временный объект, и, следовательно, не может быть сохранен для дальнейшего использования.

Для установки значения счетчика и получения текущего значения используются функции SetPos и GetPos соответственно. Тип текущего значения — целое число, следовательно для изменения содержимого ассоциированного окна, которое не является числом, следует использовать некоторое преобразование. В этом случае значение счетчика может использоваться в качестве индекса массива строк или объектов другого типа.

При получении текущего значения счетчик запрашивает содержимое ассоциированного с ним окна.

Обе функции проверяют заданное значение или результат преобразования содержимого ассоциированного окна. Это значение должно лежать в пределах заданного диапазона. Напомним, что по умолчанию для счетчика задан диапазон от 100 до 0 (именно так: 100 — минимальное значение, а 0 — максимальное). Для изменения диапазона следует использовать одну из функций:

void CSpinButtonCtrl::**SetRange**(
  int nLower,
  int nUpper)

или

void CSpinButtonCtrl::**SetRange32**(
  int nLower,
  int nUpper) —

параметры nLower и nUpper содержат значения границ диапазона. Значения и того и другого параметра должны лежать в пределах от UD_MINVAL и до UD_MAXVAL. Кроме того, разница между максимальным и минимальным значениями границ не должна быть больше UD_MAXVAL. В настоящий момент верны следующие значения:

UD_MAXVAL	0x7FFF (32 767)
UD_MINVAL	–UD_MAXVAL (–32 767)

Примечание

Естественно, что в программе для проверки допустимости значений следует использовать не числа, а мнемонические обозначения границ.

Кроме назначения новых границ диапазона, можно, воспользовавшись функцией GetRange, получить текущие значения границ диапазона изменения счетчика:

void CSpinButtonCtrl::**GetRange**(
  int & lower,
  int & upper)

и

void CSpinButtonCtrl::**GetRange32**(
int & lower,
int & upper) —

возвращают 32-разрядные числа, содержащие нижнюю и верхнюю границы диапазона допустимых значений.

Если требуется существенно изменить значение счетчика, а вместе с ним и содержимое ассоциированного окна именно с помощью средств счетчика, т. е. его кнопок со стрелками, приходится тратить довольно много времени. Значение счетчика можно изменить только на одну единицу. Однако счетчик предоставляет возможность изменять величину приращения в зависимости от времени непрерывного нажатия кнопок. Для этого служат функции SetAccel и GetAccel.

И, наконец, последняя группа функций, позволяющая задавать основание системы счисления для отображения числовых значений счетчика в ассоциированном окне. Всего можно использовать две системы счисления — десятичную и шестнадцатеричную. Для этого используется, как обычно, пара функций SetBase и GetBase. Первая позволяет назначать систему счисления, а вторая — получать информацию о текущей установленной:

int CSpinButtonCtrl::**SetBase**(int nBase) —

возвращает основание ранее установленной системы счисления, либо 0, если была задана система, не поддерживаемая счетчиком, т. е. не десятичная и не шестнадцатеричная. Параметр nBase может принимать значения 10 или 16. Следует добавить, что при установке шестнадцатеричной системы счисления значения счетчика *не могут быть отрицательными*.

Всплывающая подсказка

Элемент управления "всплывающая подсказка" представляет собой небольшое всплывающее окно, отображающее однострочный текст в качестве краткого описания или комментария к связанному с ним объекту приложения. Таким объектом может быть любое окно или даже прямоугольная область клиентской части окна. Большую часть времени подсказка скрыта и появляется лишь тогда, когда пользователь устанавливает курсор мыши на объект, имеющий подсказку, и оставляет его в границах объекта примерно полсекунды. Естественно, что подсказка появляется в непосредственной близости от объекта. Подсказка исчезает при нажатии кнопки мыши или при смещении курсора с объекта.

Текст подсказки может содержать любую необходимую информацию, т. к. перед ее выводом элемент управления получает извещение и может сгенерировать текст "на ходу". Например, в случае, если в области уведомления полосы задач задано отображение текущего времени, помещение на часы курсора приведет (через полсекунды) к появлению подсказки с текущей датой.

Следует добавить, что один элемент управления может обеспечить подсказкой несколько объектов.

Класс CToolTipCtrl и соответствующие функции ряда оконных классов библиотеки MFC обеспечивают полную поддержку всплывающих подсказок.

Интересным представляется то, как можно реализовать поддержку всплывающих подсказок в приложении. При реализации одного из способов класс CToolTipCtrl в явном виде приложению, в принципе, не нужен — класс окна имеет средства, предоставляющие все необходимое; достаточно перегрузить одну или несколько виртуальных функций. Другой способ, напротив, в полной мере может использовать только объект (объекты) класса CToolTipCtrl и при этом никак не задействовать специальные возможности класса CWnd и производных от него. Наконец, третий способ — ничего не делать (почти ничего). Правда, при этом подсказками будут снабжены только элементы панели (панелей) инструментов (TOOLBAR) — и все. Кстати сказать, такие средства автоматической генерации кода, как Component Gallery и AppWizard обеспечивают 90% всего необходимого для реализации второго и третьего способов соответственно.

Оконная поддержка

Для использования всплывающих подсказок необходимо проделать целый ряд вспомогательных действий. Поэтому прежде чем описывать непосредственно класс CToolTipCtrl, рассмотрим механизм, позволяющий обойтись без него. Он довольно сложен, однако часть классов библиотеки MFC поддерживают подсказки автоматически — класс CFrameWnd и производные от него (тот самый третий способ). Связано это с тем, что данный класс имеет в своем составе обработчик уведомляющего сообщения TTN_NEEDTEXT. Однако все остальные классы окон, не являющиеся производными от класса CFrameWnd, такой поддержки не имеют. Класс окна CWnd предоставляет разработчикам следующие функции поддержки подсказок: EnableToolTips, CancelToolTips, FilterToolTipMessage и OnToolHitTest. Рассмотрим часть из них более подробно.

BOOL CWnd::**EnableToolTips**(BOOL bEnable) —

> возвращает TRUE, если подсказки разрешены, и FALSE в противном случае. При этом параметр bEnable разрешает или запрещает вывод подсказок. Однако только вызова этой функции для разрешения вывода подсказок недостаточно.

Еще одна, скорее вспомогательная, функция OnToolHitTest, которую можно перегрузить для настройки ряда параметров, прежде всего, текста будущей подсказки. Нет смысла углубляться в представление этой функции — скажем только, что по умолчанию она:

- Подтверждает или опровергает наличие некоторого объекта в точке текущего положения курсора

- Устанавливает флаг, сообщающий системе о необходимости запроса текста для подсказки перед его выводом (LPSTR_TEXTCALLBACK)
- Устанавливает флаг центрирования для объектов, которые не являются кнопками (BUTTON)

У этой функции есть, тем не менее, одно интересное свойство, воспользовавшись которым, можно, по сути, реализовать поддержку всплывающих подсказок еще одним, четвертым способом — для этого следует задать в явном виде строчку текста для вывода подсказки. В этом случае никаких обработчиков сообщений не нужно! Смотрите сами:

```
int CNeptuneWnd::OnToolHitTest(CPoint point, TOOLINFO* pTI) const
{
  // Получаем идентификатор элемента управления
  int nHit = CWnd::OnToolHitTest(point, pTI);
  if (pTI != NULL)
  {
    // Загружаем текст, определяемый тем же идентификатором, что и
    // элемент управления, но заданный в строковых ресурсах приложения
    CString csText;
    csText.LoadString(nHit);
    if (!csText.IsEmpty())    {
      pTI->lpszText = new TCHAR[csText.GetLength()+1];
      lstrcpy(pTI->lpszText, (LPCTSTR)csText);
    }
  }
  return nHit;
}
```

Обратите внимание на выделение памяти для хранения текста. Дело в том, что впоследствии код библиотеки MFC самостоятельно освободит ее в случае, если значение элемента lpszText структуры TOOLINFO не равно LPSTR_TEXTCALLBACK. (При перегрузке функции не забудьте добавить слово "const" после имени и параметров функции как при описании, так и при ее определении.)

Наконец, поговорим об обработчике сообщения TTN_NEEDTEXT. Такой обработчик нужен как раз в случае, если функция OnToolHitText не определена описанным выше способом.

Приведем методику создания обработчика этого сообщения.

- В определении класса функция описывается так:

  ```
  BOOL memberFxn( UINT id, NMHDR * pTTTStruct, LRESULT * pResult);
  ```

- В карту сообщений добавляется строка:

```
ON_NOTIFY_EX(TTN_NEEDTEXT, 0, memberFxn)
```

- Определение функции добавляется так:

```
BOOL CClassWnd::OnToolTipNotify(UINT id, NMHDR* pNMHDR,
  LRESULT* pResult)
{
  TOOLTIPTEXT *pTTT = (TOOLTIPTEXT *)pNMHDR;
  if (pTTT->uFlags & TTF_IDISHWND)  {
    // idFrom - фактически дескриптор объекта
    UINT nID = ::GetDlgCtrlID((HWND)pNMHDR->idFrom);
    if (nID)    {
      pTTT->lpszText = MAKEINTRESOURCE(nID);
      pTTT->hinst = AfxGetResourceHandle();
      return TRUE;
    }
  }
  return FALSE;
}
```

Какой бы из двух способов ни был выбран, необходимо вызвать функцию CWnd::EnableToolTips со значением параметра TRUE.

Возвращаемся к классу CToolTipCtrl

Итак, зачем же все-таки нужен класс CToolTipCtrl? На самом деле, все способы использования подсказок, рассмотренные ранее, базируются как раз на применении этого класса. Вообще, сама библиотека классов MFC тем и хороша, что является как бы надстройкой над Win32 API, а не просто упрощает программирование. Наряду с функциями классов, инкапсулирующих соответствующие объекты Windows, большинство классов библиотеки содержат целый ряд функций, обеспечивающих взаимодействие объектов приложения, связывая их в единое целое, скрывая или упрощая действия, необходимые для программирования под Windows.

Перед описанием класса приведем пример его использования.

```
BOOL CVenusDlg::OnInitDialog()
{
  ...
  m_ToolTip.Create(this); // Создаем объект-подсказку
  // Связываем тексты подсказки с двумя объектами
  m_ToolTip.AddTool(GetDlgItem(IDOK), _T("Подтверждение"));
  m_ToolTip.AddTool(GetDlgItem(IDCANCEL), _T("Отмена"));
```

```
  // Активизируем (разрешаем) вывод подсказок
  m_ToolTip.Activate(TRUE);
  ...
}
BOOL CVenusDlg::PreTranslateMessage(MSG* pMsg)
{
  m_ToolTip.RelayEvent(pMsg);
  return CDialog::PreTranslateMessage(pMsg);
}
```

Вот и все, что необходимо для обеспечения вывода подсказок к элементам диалога. Обратите внимание, что один и тот же объект класса CToolTipCtrl и вместе с ним объект Windows — подсказка, обрабатывает вывод подсказок для более чем одного объекта.

Кроме конструктора, который только создает объект класса, для создания объекта Windows необходимо воспользоваться функцией Create:

BOOL CToolTip::**Create**(
CWnd* pParentWnd,
DWORD dwStyle = 0) —

обязательный параметр pParentWnd должен указывать на окно, элементам которого и будут сопоставлены подсказки. Второй параметр может содержать комбинацию любых стилей элементов управления. Для рассматриваемого класса предусмотрены два дополнительных:

TTS_ALWAYSTIP	Обеспечивает вывод текста подсказки независимо от того, активно или нет родительское окно элемента управления
TTS_NOPREFIX	Отменяет обработку символа амперсанда (&) в качестве управляющего (этот символ указывает на необходимость подчеркивания следующего за ним знака); указание этого стиля, таким образом, позволяет использовать текст пунктов меню для подсказок

Функция Create кроме заданных при ее вызове стилей добавляет стили WS_POPUP и WS_EX_TOOLWINDOW.

Основная функция назначения подсказок объектам окна:

BOOL CToolTipCtrl::**AddTool**(
CWnd* pWnd,
UINT nIDText,
LPCRECT lpRectTool = NULL,
UINT nIDTool = 0)

и

BOOL CToolTipCtrl::**AddTool**(
CWnd* pWnd,
LPCTSTR lpszText = LPSTR_TEXTCALLBACK,

LPCRECT lpRectTool = NULL,
UINT nIDTool=0) —

параметр pWnd является указателем на объект класса, связанного с окном Windows, — чаще всего этим объектом является элемент управления, список которых теперь существенно расширен.

Отличие версий заключается в том, каким образом задается текст подсказки. В первом случае это идентификатор nIDText, а во втором — указатель на строку текста lpszText (или значение LPSTR_TEXTCALLBACK), сообщающий о необходимости запросить текст подсказки перед ее выводом. Для этого следует создать обработчик сообщения TTN_NEEDTEXT родительского окна (см. раздел "Оконная поддержка").

Если в качестве объекта используются элементы управления, допускающие воздействие со стороны пользователя, то при связывании с ними текста подсказок нет необходимости использовать остальные два параметра функции. С другой стороны, если в качестве первого параметра задан указатель на родительское окно, использование параметров lpRectTool и nIDTool позволит выводить подсказку в любом месте окна, например, там, где находится статический элемент. Область назначения подсказки задается в координатах относительно левого верхнего угла клиентской области окна, заданного параметром pWnd.

```
// IDC_TEXT_PROMPT - идентификатор статического элемента блока диалога
CRect rect;
GetDlgItem(IDC_TEXT_PROMPT)->GetWindowRect(&rect);
ScreenToClient(&rect);
// Связываем текст подсказки с областью окна там,
// где расположен статический текст
m_ToolTip.AddTool(this, _T("Текст приглашения"),
  &rect, IDC_TECT_PROMPT);
```

Функция обратного вызова (callback function) DelTool отменяет вывод подсказки для заданного указателем окна.

Даже после создания объекта-подсказки сами подсказки выводиться не будут. Для этого необходимо, во-первых, активизировать, другими словами, разрешить вывод подсказок, и, во-вторых, обеспечить поддержку получения соответствующего сообщения о необходимости вывода подсказки объекту подсказки. В примере все это продемонстрировано. Рассмотрим используемые для этого функции.

void CToolTipCtrl::**Activate**(
BOOL bActivate) —

разрешает или запрещает bActivate вывод подсказок.

Для вывода подсказки необходимо передать объекту CToolBarCtrl информацию о местоположении указателя мыши, а также о нажатии клавиш, но уже

для того, чтобы спрятать текст подсказки. Для этого в перегруженной функции родительского окна необходимо вызвать функцию:

void CToolTipCtrl::**RelayEvent**(LPMSG lpMsg) —

обрабатывает только сообщения WM_LBUTTONDOWN, WM_MOUSEMOVE, WM_LBUTTONUP, WM_RBUTTONDOWN, WM_MBUTTONDOWN, WM_RBUTTONUP и WM_MBUTTONUP.

Дополнительно, пожалуй, стоит выделить функцию HitTest, позволяющую получить информацию о наличии подсказки по заданным координатам, и функцию SetDelayTime, задающую интервал времени, после которого будет выведена подсказка.

Кроме вышеперечисленных, класс CToolTipCtrl имеет в своем составе функции, позволяющие манипулировать подсказками: получить и назначить параметры подсказок (GetToolInfo и SetToolInfo), получить копию текста подсказки, связанной с конкретным окном (GetText), или назначить новый текст (UpdateTipText), изменять цвета текста и фона (SetTipTextColor и SetTipBkColor), переустанавливать размеры и положение окна подсказки (SetMargin и SetMaxTipWidth) и некоторые другие.

Примечание

Для всех перечисленных функций в классе также реализованы "противоположные", с помощью которых можно получить соответствующую информацию.

Глава 26

Комплексные элементы управления

В этом разделе мы рассмотрим общие элементы управления, которые в отличие от рассмотренных в предыдущей главе отличаются более широкими функциональными возможностями. Их использование существенно облегчает создание современных приложений.

Но начнем мы, как это ни покажется странно, с описания класса CImageList, существенно облегчающего работу со списками изображений, которые широко используются многими элементами управления (и не только ими).

Список изображений

Список изображений — мощное средство поддержки работы с большим количеством битовых массивов и значков, доступ к которым осуществляется по номеру, а не по идентификатору. Единственное требование к изображениям списка заключается в том, что все они должны иметь один размер. Все изображения хранятся в одном битовом массиве, в формате, поддерживаемом монитором. Ширина битового массива определяется количеством и шириной хранимых изображений, а их высота определяет высоту массива. Список также может дополнительно хранить монохромный битовый массив, используемый в качестве маски, для реализации "прозрачного" (transparent) вывода. Список поддерживает функции рисования, создания и удаления всего списка, добавления и уничтожения составляющих его элементов, их объединения и переноса (dragging).

В библиотеке MFC есть класс CImageList, инкапсулирующий возможности списка изображений. Собственно список не является в полном смысле слова элементом управления, т. к. он не является окном, однако при этом список — это объект Windows, и класс CImageList, по сути, предоставляет удобный интерфейс с этим объектом. Кроме использования списка для любых

нужд разработчика, он в ряде случаев требуется некоторым общим элементам управления, таким как "просмотр списка" (LIST VIEW), "просмотр дерева" (TREE VIEW), "вкладка" (TAB) и т. д.

Существуют списки двух типов: содержащие и не содержащие маски. Списки первого типа содержат два битовых массива одного размера, один — для хранения собственно изображений, другой — для хранения монохромного битового массива, который содержит маски для каждого изображения.

Создание списка

Создание списка при помощи класса CImageList осуществляется в два этапа. Во-первых, следует создать объект класса, а затем вызвать одну из версий функции Create для создания списка-объекта Windows. Для хранения дескриптора последнего класс имеет в своем составе член m_hImageList:

HIMAGELIST CImageList::**m_hImageList** —

доступ к нему при необходимости рекомендуется осуществлять при помощи функций GetSafeHandle, Attach и Detach.

Функции создания существенно отличаются по производимым ими действиям. Рассмотрим их подробнее.

BOOL CImageList::**Create**(
int cx, int cy, BOOL bMask, int nInitial, int nGrow) —

создает пустой список, с задаваемыми параметрами cx и cy размерами битовых массивов, которые в дальнейшем можно будет добавить в список. При задании ненулевого значения параметра nInitial, сразу при создании списка создается битовый массив, который в дальнейшем будет содержать изображения. Параметр nGrow позволяет более эффективно выделять память для хранения изображений, он задает число, которое определяет, на сколько изображений следует увеличивать битовый массив при необходимости. Параметр bMask определяет, какого типа следует создавать список — содержащий маски или нет. Все остальные версии функции создают списки, в обязательном порядке содержащие маски.

Следующие две версии функции Create позволяют создавать уже заполненные списки, содержимое которых, впрочем, можно изменять во время работы:

BOOL CImageList::**Create**(
UINT nBitmapID, int cx, int nGrow, COLORREF crMask)

и

BOOL CImageList::**Create**(LPCTSTR lpszBitmapID, int cx, int nGrow,
COLORREF crMask) —

параметры nBitmapID и lpszBitmapID задают дескрипторы битовых массивов, уже содержащих изображения, количество которых определяется их шириной (параметр cx); crMask позволяет указать цвет для создания масок и реализации прозрачного вывода изображений. Цвет каждой точки изображения, совпадающий с цветом, заданным параметром crMask, заменяется на черный, и устанавливается соответствующий бит маски.

Последняя версия функции позволяет создать список на основе двух других:

BOOL CImageList::**Create**(CImageList& imagelist1, int nImage1, CImageList& imagelist2, int nImage2, int dx, int dy) —

объединяет содержимое двух списков таким образом, что изображения второго списка накладываются на изображения первого, а маска создается в результате выполнения логической операции OR для каждого бита масок обоих списков. Параметры dx и dy определяют размеры изображений.

Изменение содержимого списка

Для добавления, удаления и замены изображений используются функции Add, Remove и Replace соответственно.

int CImageList::**Add**(CBitmap* pbmImage, CBitmap* pbmMask),
int CImageList::**Add**(CBitmap* pbmImage, COLORREF crMask)

и

int CImageList::**Add**(HICON hIcon) —

позволяют не только добавлять сразу несколько битовых массивов, количество которых определяется заданной при создании шириной изображений списка, но и задавать маску для изображений. Последняя версия функции позволяет добавлять в список значок, который также содержит маску.

BOOL CImageList::**Replace**(int nImage, CBitmap* pbmImage, CBitmap* pbmMask)

и

int CImageList::**Replace**(int nImage, HICON hIcon) —

позволяют заменить изображение по его номеру как битовым массивом (параметр pbmImage), так и значком.

BOOL CImageList::**Remove**(int nImage) —

удаляет элемент с индексом nImage из списка изображений.

Значки, в отличие от битовых массивов, уже содержат маску, позволяющую производить рисование в прозрачном режиме. Следующая функция создает значок на основе битового массива и маски (если она есть):

HICON CImageList::**ExtractIcon**(int nImage) —

возвращает дескриптор созданного значка для изображения по номеру, задаваемому параметром nImage.

Для заполнения и сохранения всех изображений списка реализованы функции:

BOOL CImageList::**Read**(CArchive* pArchive)

и

BOOL CImageList::**Write**(CArchive* pArchive) —

используют указатели на объекты класса CArchive. Эти объекты следует создать самостоятельно.

Так как объект класса CImageList лишь представляет в программе объект Windows, путем связывания с ним последнего можно обеспечить доступ к ранее созданному другими средствами списку. К функциям данной группы относятся функции Attach и Detach.

Рисование изображений

Для отображения в прозрачном режиме, когда все точки области устройства вывода изображения имеют один и тот же цвет, следует задать цвет фона изображения таким же, что и цвет области. В этом случае существенно ускоряется процесс вывода, т. к. не требуется дополнительное использование маски. Для этого класс CImageList имеет в своем составе функцию:

COLORREF CImageList::**SetBkColor**(COLORREF cr) —

возвращает ранее установленный цвет фона. По умолчанию цвет имеет значение CLR_NONE, в этом случае прозрачный режим реализуется с использованием маски.

Для рисования изображений следует использовать функцию Draw:

BOOL CImageList::**Draw**(CDC* pdc, int nImage, POINT pt, UINT nStyle) —

осуществляет вывод на устройство, определяемое указателем (параметр pdc) на объект класса CDC, по координатам, задаваемым параметром pt. Изображение определяется индексом (параметр nImage). Параметр nStyle задает режим вывода изображения и может содержать комбинацию следующих значений.

ILD_NORMAL	Рисование осуществляется на основе цвета фона, заданного вызовом функции SetBkColor, что позволяет реализовать прозрачный режим и при отсутствии маски
ILD_TRANSPARENT	Рисование осуществляется с использованием маски вместо цвета фона
ILD_BLEND50, ILD_SELECTED, ILD_BLEND	При наличии маски происходит вывод изображения, "смешанного" с 50% цвета "подсвечивания", для индикации выбора, режим игнорируется при отсутствии маски
ILD_BLEND25, ILD_FOCUS	При наличии маски происходит вывод изображения, "смешанного" с 25% цвета "подсвечивания", для индикации выбора, режим игнорируется при отсутствии маски
ILD_OVERLAYMASK	Происходит вывод изображения с участием маски перекрытия; совместно с флагом следует указать индекс маски перекрытия, заданного при помощи макроса INDEXTOOVERLAYMASK

Примечание

Каждый список изображений может иметь до четырех масок перекрытия, задаваемых функцией SetOverlayImage. В качестве масок перекрытия используются изображения, содержащиеся в списке.

Перемещение изображений

Как ни странно, класс списка изображений поддерживает набор функций для осуществления процесса "перетаскивания" его элементов. Достоинство реализации в том, что перемещение изображения происходит плавно, изображение при этом остается полноцветным и исключается мигание курсора мыши.

Для того чтобы начать процесс перемещения изображения, следует вызвать функцию BeginDrag:

BOOL CImageList::**BeginDrag**(int nImage, CPoint ptHotSpot) —

параметр nImage задает порядковый номер изображения, а параметр ptHotSpot — координаты *точки привязки* (hot spot) на изображении, которая и будет перемещаться, и относительно нее будет производиться отображение изображения.

Совместно с функцией BeginDrag следует вызвать функцию:

static BOOL CImageList::**DragEnter**(CWnd* pWndLock, CPoint point) —

блокирует перерисовку окна в процессе перемещения изображения. Первый параметр указывает на окно, изменения в котором будут заблокированы. Значение этого параметра, равное NULL, позволяет перемещать изображение и вне окна, по всему экрану, при этом окном, где будет происходить перемещение, будет являться рабочий стол (desktop). Второй параметр задает точку в окне, которой будет соответствовать точки привязки. Координаты точки задаются в области всего окна, а не только в его клиентской части, следовательно, требуется произвести соответствующие преобразования.

В процессе перемещения (обычно в обработчике сообщения WM_MOUSEMOVE) для отображения текущего положения изображения служит функция DragMove:

static BOOL CImageList::**DragMove**(CPoint pt) —

принимает координаты нового положения точки привязки в области окна.

По завершении операции перемещения следует сообщить системе о необходимости разблокирования перерисовки окна (функция DragLeave) и завершить собственно перемещение (функция EndDrag).

Если возникает необходимость рисования в окне во время перемещения изображения, следует либо по мере необходимости разблокировать окно при помощи функции DragLeave (с последующей повторной блокировкой), либо использовать функцию DragShowNolock.

Кроме вышеперечисленных, полезной может оказаться функция SetDragCursorImage для временного изменения изображения на момент перемещения путем комбинирования перемещаемого изображения с некоторым другим, например, с изображением курсора.

Далее приведем пример программы (приложение ImageDrag), в которой реализован описанный выше механизм — из начального положения изображение при помощи мыши может быть перенесено в любую часть окна (рис. 26.1).

```
// Файл ImageDragFrm.cpp
// CImageDragFrame
BEGIN_MESSAGE_MAP(CImageDragFrame, CFrameWnd)
  //{{AFX_MSG_MAP(CImageDragFrame)
  ON_WM_PAINT()
  ON_WM_CREATE()
  ON_WM_LBUTTONDOWN()
  ON_WM_MOUSEMOVE()
  ON_WM_LBUTTONUP()
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()
// CImageDragFrame message handlers
void CImageDragFrame::OnPaint()
{
  CPaintDC dc(this);
  // Отображение изображения
  m_List.Draw(&dc, 0, m_pt, ILD_TRANSPARENT);
}
int CImageDragFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
  if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
    return -1;
  // Создание списка изображений
  m_List.Create(32, 32, TRUE, 1, 1);

  // Добавление значка
  m_List.Add(AfxGetApp()->LoadIcon(IDR_IMAGEDRAG));

  // Начальная инициализация дополнительных параметров
  m_pt = CPoint(0, 0);
  m_Drag = FALSE;
  return 0;
}
void CImageDragFrame::OnLButtonDown(UINT nFlags, CPoint point)
{
  CRect rect(m_pt.x, m_pt.y, m_pt.x+32, m_pt.y+32);
  // Проверка на наличие курсора мыши в рамках значка
  if (rect.PtInRect(point))  {
    // Начинаем перемещение
    m_Drag = TRUE;
```

```
    m_List.BeginDrag(0, CPoint(0, 0));
    // Преобразование координат
    GetWindowRect(&rect);
    CPoint pt = point;
    ClientToScreen(&pt);
    m_List.DragEnter(this,
      CPoint(pt.x — rect.left, pt.y — rect.top));
  }
  else
    m_Drag = FALSE;
  CFrameWnd::OnLButtonDown(nFlags, point);
}
void CImageDragFrame::OnMouseMove(UINT nFlags, CPoint point)
{
  if (m_Drag)  {
    // Преобразование координат
    CRect rect;
    GetWindowRect(&rect);

    CPoint pt = m_pt = point;
    ClientToScreen(&pt);

    // Перемещение
    m_List.DragMove(CPoint(pt.x — rect.left, pt.y — rect.top));
  }
  CFrameWnd::OnMouseMove(nFlags, point);
}
void CImageDragFrame::OnLButtonUp(UINT nFlags, CPoint point)
{
  if (m_Drag)  {
    // Завершение перемещения
    m_List.DragLeave(this);
    m_List.EndDrag();
    m_Drag = FALSE;
    // Инициируем перерисовку окна, т.к. после вызова
    // функции DragLeave изображение было скрыто
    InvalidateRect(NULL);
  }
  CFrameWnd::OnLButtonUp(nFlags, point);
}
```

Рис. 26.1. Окно приложения ImageList в момент перемещения изображения

Информация о списке

Для получения информации о параметрах списка следует пользоваться функциями GetImageCount, возвращающей количество элементов (изображений) списка, и GetImageInfo, заполняющей структуру IMAGEINFO информацией об изображении по его порядковому номеру в списке:

BOOL CImageList::**GetImageInfo**(int nImage, IMAGEINFO* pImageInfo)

Для получения информации об изображении следует передать указатель на структуру IMAGEINFO, первые два поля которой используются для хранения дескрипторов битового массива и маски, поля Unused1 и Unused2 перестали использоваться и должны быть установлены в 0 (раньше они содержали количество плоскостей и битов, соответственно, требуемых для хранения цвета), а поле rcImage — размеры битового массива.

Теперь можно переходить непосредственно к элементам управления.

Заголовок

В этом разделе мы рассмотрим еще один элемент управления — "заголовок" и соответствующий ему класс библиотеки MFC CHeaderCtrl.

Заголовок полезен для использования вместе со списками, создаваемыми с помощью класса CListCtrl, но это не значит, что его нельзя использовать в других случаях, например, при создании электронных таблиц. Заголовок обычно разбит на части, называемые *элементами заголовка*. Каждая из них является собственно заголовком для соответствующего столбца. Разделители можно передвигать, меняя ширину столбцов, в случае, если элемент заголовка создан с соответствующим стилем.

На рис. 26.2 показано, как данный элемент управления может быть использован для создания окна, в котором выводится информация о файлах на диске.

Рис. 26.2. Использование элемента управления "заголовок"

Можно создать элемент заголовка, по внешнему виду похожий и действующий при щелчке на нем мышью так же, как кнопки, срабатывающие по нажатию. Для этого нужно задать стиль HDS_BUTTONS. В этом случае можно реализовать, например, сортировку по алфавиту столбца записей, осуществляемую при нажатии на элемент заголовка соответствующего столбца.

Каждый элемент заголовка может содержать текстовую строку или битовый массив и 32-битное значение, ассоциированное с ним, которое определяется приложением.

В последней версии реализации этого элемента управления добавлены следующие возможности:

- Настройка порядка расположения элементов
- Поддержка режима drag-and-drop для переупорядочивания элементов заголовка
- Изображение текста столбца при изменении размеров
- Подсвечивание элемента заголовка, когда указатель мыши "задерживается" над ним
- Поддержка вывода изображений, хранящихся в списке

Но обо всем по порядку.

Создание элемента заголовка

Элемент заголовка по-прежнему невозможно непосредственно создать в редакторе ресурсов. Однако теперь средство ClassWizard поддерживает создание объектов производных от класса CHeaderCtrl.

Для того чтобы вставить заголовок в блок диалога, следует:

1. Вручную создать объект класса CHeaderCtrl в классе блока диалога.
2. В функции OnInitDialog создать собственно объект-заголовок, используя функцию Create.
3. Добавить в заголовок элементы заголовка

4. С помощью средства ClassWizard добавить необходимые обработчики сообщений в классе блока диалога.

Для того чтобы вставить заголовок в окно просмотра, не базирующееся на классе CListView, следует:

1. Вручную создать объект класса CHeaderCtrl в классе представления.
2. В функции OnInitialUpdate создать собственно объект-заголовок, используя функцию Create.
3. Добавить в заголовок элементы заголовка.
4. С помощью средства ClassWizard добавить необходимые обработчики сообщений в класс представления.

Как вы можете видеть, отсутствие поддержки со стороны средства разработки приложений приводит к тому, что создание заголовка и в блоках диалога и в окнах-представлениях практически идентично. В обоих случаях встроенный объект создается тогда, когда создается блок диалога или окно-представление при помощи вызова функции Create:

BOOL CHeaderCtrl::**Create**(DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID) —

параметр dwStyle может содержать любые комбинации стилей элементов управления, включая следующие, свойственные рассматриваему:

HDS_BUTTONS	Элементы заголовка выглядят как кнопки, срабатывающие по нажатию
HDS_HORZ	Заголовок имеет горизонтальную ориентацию
HDS_VERT	Заголовок имеет вертикальную ориентацию (к сожалению, в настоящий момент этот стиль не поддерживается)
HDS_HIDDEN	Изначально заголовок не виден
HDS_HOTTRACK	Элемент заголовка "подсвечивается", когда над ним задерживается указатель мыши
HDS_DRAGDROP	Элементы заголовка можно переупорядочивать с помощью механизма drag-and-drop
HDS_FULLDRAG	Элемент управления продолжает отображать содержимое столбцов во время изменения их размеров

Следующая группа стилей определяет, где элемент заголовка располагается и как изменяет свои размеры:

CCS_BOTTOM	Заголовок располагается в нижней части клиентской области родительского окна и имеет ширину, равную ширине родительского окна
CCS_NOMOVEY	Заставляет заголовок в ответ на сообщение WM_SIZE изменять размер и перемещаться по горизонтали, но не по вертикали; если уже использован стиль CCS_NORESIZE, то этот стиль не применим; элемент заголовка по умолчанию имеет данный стиль

CCS_NOPARENTALIGN	Предотвращает автоматическое перемещение заголовка в верхнюю или нижнюю часть родительского окна; элемент заголовка сохраняет свое расположение внутри родительского окна, несмотря на изменение его размеров; если использованы стили CCS_TOP или CCS_BOTTOM, то высота устанавливается по умолчанию, а расположение и ширина не изменяются
CCS_NORESIZE	Заголовок не использует ширину и высоту, заданные по умолчанию; он должен использовать данные, заданные при создании или изменении размеров
CCS_TOP	Заголовок располагается в верхней части клиентской области родительского окна и имеет ширину, равную ширине родительского окна

rect задает размер и расположение заголовка; pParentWnd — родительское окно (значение этого параметра не должно быть равно NULL); nID задает идентификатор заголовка.

После того как заголовок создан, можно добавлять в него элементы. Обычно устанавливают один элемент на столбец. Для этого нужно выполнить следующие шаги для каждого добавляемого элемента заголовка:

1. Подготовить структуру HDITEM.
2. Вызвать функцию InsertItem, передав ей указатель на эту заполненную структуру.

Структура HDITEM с информацией об элементе заголовка имеет следующий вид:

```
typedef struct _HDITEM {
    UINT mask;
    int cxy;
    LPSTR pszText;
    HBITMAP hbm;
    int cchTextMax;
    int fmt;
    LPARAM lParam;
#if (_WIN32_IE >= 0x0300)
    int     iImage;
    int     iOrder;
#endif
} HDITEM;
```

Поле mask определяет, какие поля структуры содержат достоверные значения для осуществления вывода, и может содержать комбинацию следующих значений:

HDI_BITMAP	Доступно поле hbm
HDI_FORMAT	Доступно поле fmt
HDI_HEIGHT	Доступно поле cxy, и оно содержит высоту элемента

HDI_IMAGE	Доступно поле iImage
HDI_LPARAM	Доступно поле lParam
HDI_ORDER	Доступно поле iOrder
HDI_TEXT	Доступно поле pszText и cchTextMax
HDI_WIDTH	Доступно поле cxy, и оно содержит ширину элемента

cxy определяет ширину или высоту элемента в зависимости от ориентации заголовка; pszText — указатель на тестовую строку, длина которой определяется значением поля cchTextMax; hbm — дескриптор битового массива; lParam задает 32-битное значение для элемента заголовка; fmt задает способ форматирования содержимого элемента заголовка и может содержать комбинацию следующих значений:

HDF_CENTER	Содержимое элемента выравнивается по центру
HDF_LEFT	Содержимое элемента выравнивается по левому краю
HDF_RIGHT	Содержимое элемента выравнивается по правому краю
HDF_RTLREADING	Элемент содержит текстовую строку, которая располагается справа-налево, как принято в арабском языке
HDF_BITMAP	Элемент содержит битовый массив
HDF_BITMAP_ON_RIGHT	Элемент содержит битовый массив, который располагается справа от текста
HDF_IMAGE	Элемент отображает рисунок из списка образов
HDF_OWNERDRAW	Рисование элемента осуществляется в коде, написанном разработчиком приложения
HDF_STRING	Элемент содержит текстовую строку

Поле iImage содержит индекс рисунка из списка образов, iOrder — порядковый номер элемента в заголовке, который определяет порядок (слева-направо) его отображения; для самого левого элемента это поле содержит 0.

Итак, для того чтобы добавить новый элемент в заголовок по заданному индексу, предназначена функция InsertItem:

int CHeaderCtrl::**InsertItem**(int nPos, HDITEM* phdi) —

параметр nPos определяет индекс элемента, перед которым будет осуществлена вставка. Если значение параметра больше текущего количества элементов заголовка, то новый элемент будет добавлен в конец заголовка.

Для удаления элемента из заголовка предназначена функция

BOOL CHeaderCtrl::**DeleteItem**(int nPos) —

nPos — индекс удаляемого элемента.

Перед созданием нового элемента управления для него можно определить подходящие размеры. Для этого в классе реализована функция:

BOOL CHeaderCtrl::**Layout**(HDLAYOUT* pHeaderLayout) —

в качестве параметра принимает указатель на структуру HDLAYOUT, содержащую информацию, используемую для установки размера и положения элемента управления.

Поле prc — указатель на структуру RECT, где располагаются координаты прямоугольника, в котором предполагается разместить элемент управления. Соответствующие размеры возвращаются в структуре WINDOWPOS, адрес которой определяется в поле pwpos.

Атрибуты элементов заголовка

Следующая пара функций позволяет получить и изменить атрибуты элементов заголовка.

BOOL CHeaderCtrl::**GetItem**(int nPos, HDITEM*pHeaderItem)

и

BOOL CHeaderCtrl::**SetItem**(int nPos, HDITEM *pHeaderItem) —

для получения информации о конкретном элементе следует задавать поле mask структуры HDITEM, в противном случае информация возвращена не будет.

Для получения общего количества элементов заголовка в классе реализована функция GetItemCount, а для параметров прямоугольника, занимаемого элементом, — функция GetItemRect.

Как мы уже упоминали, теперь для отображения в этом элементе управления рисунков можно использовать список изображений Windows. Для работы с ним в классе реализованы следующие функции:

CImageList* CHeaderCtrl::**GetImageList**( )

и

CImageList* CHeaderCtrl::**SetImageList**(CImageList* pImageList) —

позволяют, соответственно, получить и определить объект класса CImageList, в котором хранятся рисунки, отображаемые в элементе управления.

Следующие две функции позволяют узнать и изменить порядок расположения элементов в заголовке:

BOOL CHeaderCtrl::**GetOrderArray**(LPINT piArray, int iCount = −1)

и

BOOL CHeaderCtrl::**SetOrderArray**(int iCount, LPINT piArray) —

параметр piArray является указателем на буфер, в который записываются значения индексов элементов в той последовательности, в какой они должны отображаться (слева-направо) в элементе управления; параметр iCount задает общее число элементов в заголовке (если в функции GetOrderArray использовать значение по умолчанию –1, то библиотека сама определит число элементов — посредством функции GetItemCount).

int CHeaderCtrl::**OrderToIndex**(int nOrder) —

позволяет узнать порядковый номер элемента заголовка по его индексу.

Можно создать специальный рисунок, который будет отображаться на экране при перемещении элемента заголовка. Для этого следует воспользоваться функцией

CImageList* CHeaderCtrl::**CreateDragImage**(int nIndex) —

создает базовый образ элемента, отображаемый при его перемещении, и помещает его в список образов. Сам элемент задается индексом nIndex. После создания базового образа его можно менять, используя для этого возможности класса CImageList, о котором речь впереди.

Обработка извещений элемента заголовка

Для осуществления управления заголовком и связанными с ним столбцами и строками, с помощью средства ClassWizard необходимо добавить в класс родительского окна или блока диалога обработчики сообщений для функции OnNotify. Сообщения посылаются в рамках сообщения WM_NOTIFY родительскому окну. Перечислим уведомляющие сообщения, которые может посылать заголовок:

HDN_ITEMCLICK	Пользователь сделал щелчок на элементе заголовка
HDN_ITEMCHANGED	Атрибуты элемента заголовка изменились
HDN_ITEMCHANGING	Атрибуты элемента заголовка изменяются
HDN_ITEMDBLCLICK	Пользователь сделал двойной щелчок на элементе заголовка
HDN_DIVIDERDBLCLICK	Пользователь сделал двойной щелчок на разделителе
HDN_BEGINTRACK	Пользователь начал перемещать разделитель
HDN_TRACK	Пользователь перемещает разделитель
HDN_ENDTRACK	Пользователь закончил перемещать разделитель

Для получения информации о конкретном элементе, в результате взаимодействия пользователя с которым сгенерировано уведомление, следует преобразовать параметр lParam к указателю на структуру HD_NOTIFY. Эта структура содержит следующую информацию об элементе:

```
typedef struct _HD_NOTIFY {
  NMHDR hdr;
  int iItem;
  int iButton;
  HDITEM FAR* pitem;
} HD_NOTIFY;
```

Поле iItem содержит индекс элемента (поле iButton), который, возможно, был нажат (0 — левая кнопка, 1 — правая, 2 — средняя), поле pitem является указателем на ранее описанную структуру HDITEM.

Чтобы проиллюстрировать вышесказанное, приведем пример (рис. 26.3) использования заголовков (приложение Header).

В функции OnInitialUpdate класса блока диалога CHeaderDlg создается заголовок из пяти элементов одинаковой ширины.

```
// Файл HeaderDlg.cpp
BOOL CHeaderDlg::OnInitDialog()
{
  CRect rect;
  GetClientRect(&rect);
  rect.bottom = 30;
  rect.DeflateRect(5, 5);
  m_Horz.Create(HDS_BUTTONS | HDS_HORZ | CCS_TOP |
    WS_CHILD | WS_VISIBLE, rect, this, 1000);
  int nWidth = rect.Width() / 5;
  TCHAR szText[2];
  HD_ITEM item = { HDI_TEXT | HDI_WIDTH, nWidth, szText, NULL, 1 };
  for (int i = 0; i < 5; i++)  {
    m_Offset[i] = nWidth;
    szText[0] = 'A'+i; szText[1] = '\0';
    m_Horz.InsertItem(i, &item);
  }
  ...
}
```

В функции OnPaint в зависимости от ширины элементов заголовка рисуется сетка, в пять столбцов и несколько строк.

```
void CHeaderDlg::OnPaint()
{
  if (IsIconic())  { ...  }
  else  {
    CPaintDC dc(this); // device context for painting
    CPen pen(PS_DOT, 1, RGB(0, 0, 0));
    CPen *pPen = dc.SelectObject(&pen);
    CRect rect;
    GetClientRect(&rect);
    rect.DeflateRect(5, 5);
    rect.top += 20;
    dc.FillSolidRect(&rect, RGB(255, 255, 255));
    int nOffset = 4;
    for (int i = 0; i < 5; i++)    {
      if ((nOffset += m_Offset[i]) >= rect.right)
        break;
      dc.MoveTo(nOffset, 20);
      dc.LineTo(nOffset, rect.bottom);
```

```
    }
    for (i = 1; i < 10; i++)    {
      dc.MoveTo(rect.left, rect.top + (rect.Height() / 10) * i);
      dc.LineTo(rect.right, rect.top + (rect.Height() / 10) * i);
    }
    dc.SelectObject(pPen);
    CDialog::OnPaint();
  }
}
```

Функция осуществляет обработку сообщения, уведомляющего об окончании изменения ширины элемента заголовка, и инициирует перерисовку содержимого окна.

```
BOOL CHeaderDlg::OnNotify(WPARAM wParam, LPARAM lParam,
  LRESULT* pResult)
{
  NMHDR *nmhdr = (NMHDR *)lParam;
  if (nmhdr->hwndFrom == m_Horz.m_hWnd)  {
    if (nmhdr->code == HDN_ENDTRACK)
    {
      HD_NOTIFY *phdn = (HD_NOTIFY *)lParam;
      m_Offset[phdn->iItem] = phdn->pitem->cxy;
      InvalidateRect(NULL);
    }
  }
```

Рис. 26.3. Окно приложения Header

Элемент заголовка, определяемый пользователем

Можно определить вывод отдельных элементов заголовка, внешний вид которых определяется пользователем. Для этого при создании элемента нужно, чтобы поле fmt структуры HD_ITEM содержало флаг HDF_OWNERDRAW. Также следует создать класс на базе класса CHeaderCtrl и переопределить

функцию DrawItem. Эта функция будет вызываться всякий раз, когда понадобится перерисовать элемент (элементы, если их несколько) заголовка:

void CHeaderCtrl::**DrawItem**(LPDRAWITEMSTRUCT lpDrawItemStruct)

Ранее мы уже рассматривали структуру DRAWITEMSTRUCT, однако в случае заголовка ее поля имеют несколько иное значение. Поле CtlType содержит значение ODT_HEADER, поле CtlID — идентификатор окна заголовка, поле itemID — порядковый номер элемента заголовка, для рисования которого вызывается функция DrawItem, поле itemAction всегда содержит значение ODA_DRAWENTIRE, т. к. требуется перерисовка всего элемента, поле itemState — значение ODS_SELECTED, если курсор мыши находится на элементе и этот элемент находится в нажатом состоянии, в противном случае значение поля равно 0, поля hwndItem и hDC содержат дескрипторы окна заголовка и контекста устройства соответственно, поле rcItem определяет границы элемента заголовка и поле itemData содержит 32-битное число, ассоциированное с элементом.

Просмотр списка

Просмотр списка — мощное средство представления упорядоченного списка элементов, каждый из которых может иметь структуру, состоящую из нескольких полей. С каждой записью могут быть ассоциированы до трех значков различного размера, отображение которых может нести дополнительную визуальную информацию.

Режимы вывода

Элемент "просмотр списка" (список просмотра) может выводить записи четырьмя различными способами, наличие которых существенно отличает этот элемент управления от всех других. При этом возможно переключение от одного режима представления к другому после создания элемента управления. Рассмотрим режимы вывода подробнее:

Режим вывода	Описание
Крупные значки (Large Icons)	Задается стилем LVS_ICON; каждая запись представляет собой значок, под которым расположена надпись; пользователь может перемещать записи внутри списка просмотра
Мелкие значки (Small Icons)	Задается стилем LVS_SMALLICON; каждая запись выводится в виде маленького значка с надписью, расположенной справа от него; пользователь может перемещать записи внутри списка просмотра

(окончание)

Режим вывода	Описание
Список (List)	Задается стилем LVS_LIST; каждая запись выводится в виде маленького значка с надписью, расположенной справа от него; в отличие от предыдущего способа вывода, записи выводятся столбцами и пользователь не может менять их расположение внутри списка
Таблица (Report)	Задается стилем LVS_REPORT; каждая запись с информацией о ней выводится в отдельной строке; значок и надпись выводятся в столбцах, расположенных слева, а в столбцах справа приложение может размещать дополнительную информацию для каждой записи; если не задан стиль LVS_NOCOLUMNHEADER, то каждый столбец имеет заголовок

Примечание

Размер больших значков — 32×32 пикселя, а малых — 16×16.

Примером применения просмотра списка может служить правая часть окна программы Windows Explorer, которая позволяет установить перечисленные выше в таблице способы вывода записей в окне.

Для задания выравнивания пиктограмм и надписей при первых двух способах вывода можно применять стили LVS_ALIGNTOP — выравнивание по верхнему краю (по умолчанию) или LVS_ALIGNLEFT — выравнивание по левому краю. Способ выравнивания также можно изменять уже после создания просмотра списка. Для отделения стилей, задающих выравнивание записей, можно применять стиль LVS_ALIGNMASK.

Дополнительные стили окна позволяют использовать другие возможности, например, автоматически выбирать элемент, когда курсор мыши "задерживается" над ним определенное время, редактировать надписи под значками непосредственно в просмотре списка, выбирать больше одной записи, в режиме таблицы изменять порядок расположения столбцов и т. д.

Создание просмотра списка

В библиотеке MFC есть класс CListCtrl, инкапсулирующий возможности рассматриваемого элемента управления. Как и большинство других элементов управления, просмотр списка создается или в блоке диалога добавлением в шаблон блока диалога соответствующего элемента или в обычном окне при помощи вызова функции Create в функциях окна OnInitDialog (для диалогов) или OnCreate:

BOOL CListCtrl::**Create**(DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID) —

параметр dwStyle может содержать любые комбинации стилей элементов управления, а также следующие, свойственные рассматриваемому:

LVS_ALIGNLEFT	Записи выравниваются по левому краю при просмотре в режиме крупных или мелких значков
LVS_ALIGNTOP	Записи выравниваются по верхнему краю при просмотре в режиме крупных или мелких значков
LVS_AUTOARRANGE	Значки автоматически выравниваются при просмотре в режиме крупных или мелких значков при добавлении, удалении или перемещении записей
LVS_EDITLABELS	Позволяет редактировать текстовые надписи непосредственно в списке просмотра, для поддержки этого режима необходимо обрабатывать уведомляющее сообщение LVN_ENDLABELEDIT
LVS_ICON	Задает режим крупных значков
LVS_LIST	Задает режим списка
LVS_NOCOLUMNHEADER	Заголовки столбцов не выводятся на экран в случае просмотра в режиме таблицы; по умолчанию заголовки выводятся на экран
LVS_NOLABELWRAP	Текст в надписях выводится в виде одной строки без переноса на следующую; по умолчанию текст может быть перемещен в случае просмотра в режиме крупных или мелких значков
LVS_NOSCROLL	Отключает прокрутку списка просмотра в окне; все записи должны вмещаться в клиентскую область окна
LVS_NOSORTHEADER	Отключает стиль имитации кнопок для элементов заголовка
LVS_OWNERDRAWFIXED	Требует написания кода для реализации отображения записей в случае просмотра в виде таблицы; список просмотра посылает сообщение WM_DRAWITEM в случае необходимости перерисовки каждой записи, но не посылает отдельные сообщения для подзаписей (см. ниже)
LVS_REPORT	Задает режим таблицы, для возможности отображения значков совместно с записями таблицы следует комбинировать с этим стилем стиль LVS_ICON
LVS_SHAREIMAGELISTS	Просмотр списка использует список изображений совместно с другими элементами управления
LVS_SHOWSELALWAYS	Позволяет всегда показывать выбранные элементы, даже когда элемент управления не имеет фокуса
LVS_SINGLESEL	Позволяет в данный момент времени выбирать только одну запись в просмотре списка
LVS_SMALLICON	Задает режим мелких значков
LVS_SORTASCENDING	Сортирует текстовые записи по возрастанию
LVS_SORTDESCENDING	Сортирует текстовые записи по убыванию

Помимо этих стилей можно также задавать дополнительные:

LVS_EX_CHECKBOXES	В элементе управления разрешено использовать флажки, которые видимы и функционируют во всех режимах просмотра списка
LVS_EX_FLATSB	В элементе управления плоские полосы прокрутки
LVS_EX_FULLROWSELECT	при выборе элемента, выделяются (подсвечиваются) также все его подэлементы. Использование этого стиля разрешено только совместно со стилем LVS_REPORT
LVS_EX_GRIDLINES	Вокруг элементов и подэлементов изображаются линии сетки. Использование этого стиля разрешено только совместно со стилем LVS_REPORT
LVS_EX_HEADERDRAGDROP	Разрешено переупорядочивание столбцов списка просмотра методом drag-and-drop. Использование этого стиля разрешено только совместно со стилем LVS_REPORT
LVS_EX_INFOTIP	Перед выводом всплывающей подсказки, элемент управления посылает своему родительскому окну уведомление LVN_GETINFOTIP. Использование этого стиля разрешено только совместно со стилем LVS_ICON
LVS_EX_MULTIWORKAREAS	Пока для списка просмотра определены одна или несколько рабочих областей, отключает действие стиля LVS_AUTOARRANGE
LVS_EX_ONECLICKACTIVATE	Когда пользователь один раз щелкает элемент списка просмотра, элемент управления посылает своему родительскому окну уведомление LVN_ITEMACTIVATE. Кроме того, при использовании этого стиля элемент, над которым расположен курсор мыши, будет отображаться подсвеченным
LVS_EX_REGIONAL	Для просмотра списка можно создавать регион, область вне которого считается не входящей в элемент. Использование этого стиля разрешено только совместно со стилем LVS_ICON
LVS_EX_SUBITEMIMAGES	Для подэлементов разрешено использовать изображения. Использование этого стиля разрешено только совместно со стилем LVS_REPORT
LVS_EX_TRACKSELECT	Задержка курсора мыши над элементом просмотра списка дольше определенного интервала времени приводит к тому, что элемент автоматически выбирается
LVS_EX_TWOCLICKACTIVATE	Если пользователь сделал двойной щелчок на элементе просмотра списка, элемент управления посылает своему родительскому окну уведомление LVN_ITEMACTIVATE. Кроме того, при использовании этого стиля элемент, над которым расположен курсор мыши, будет отображаться подсвеченным
LVS_EX_UNDERLINECOLD	Отменяет автоматическую подсветку элемента списка просмотра при установленном стиле LVS_EX_ONECLICKACTIVATE
LVS_EX_UNDERLINEHOT	Текст элемента списка просмотра изображается подчеркнутым. Этот стиль игнорируется, если не установлен стиль LVS_EX_ONECLICKACTIVATE или LVS_EX_TWOCLICKACTIVATE

rect — задает размер и расположение списка просмотра, pParentWnd — задает родительское окно (не должен быть равен NULL); nID — задает идентификатор просмотра списка.

Для добавления записи в список служит функция InsertItem. Ее первая версия использует заранее подготовленную структуру LVITEM, а последняя в качестве параметров принимает значения, аналогичные значениям полей структуры LVITEM. Вторая и третья версии функции полезны для списков, элементы которых не требуют для отображения дополнительных данных:

int CListCtrl::**InsertItem**(const LVITEM*pItem)
int CListCtrl::**InsertItem**(int nItem, LPCTSTR lpszItem)
int CListCtrl::**InsertItem**(int nItem, LPCTSTR lpszItem, int nImage)
int CListCtrl::**InsertItem**(UINT nMask, int nItem, LPCTSTR lpszItem, UINT nState, UINT nStateMask, int nImage, LPARAM lParam)

Мы подробно рассмотрим только структуру, т. к. имена и типы параметров последней версии функции говорят сами за себя:

```
typedef struct _LV_ITEM {
  UINT mask;
  int iItem;
  int iSubItem;
  UINT state;
  UINT stateMask;
  LPSTR pszText;
  int cchTextMax;
  int iImage;
  LPARAM lParam;
#if (_WIN32_IE >= 0x0300)
  int iIndent;
#endif
} LVITEM;
```

Поле mask определяет, какие из оставшихся полей несут информацию, и может содержать комбинацию следующих значений:

LVIF_TEXT	Поле pszText содержит данные
LVIF_IMAGE	Поле iImage содержит данные
LVIF_PARAM	Поле lParam содержит данные
LVIF_STATE	Поле state содержит данные
LVIF_INDENT	Поле iIndent содержит данные
LVIF_NORECOMPUTE	Для получения текстовой информации элемент управления не генерирует уведомление LVN_GETDISPINFO
LVIF_DI_SETITEM	Список затребованной информации об элементах просмотра списка хранится в операционной системе

item задает индекс записи, перед которой осуществляется вставка новой записи; iSubItem содержит порядковый номер (начиная с 1) дополнительного поля (в функции InsertItem не используется); state и stateMask задают текущее и возможные состояния для данной записи и могут содержать комбинации следующих флагов:

LVIS_ACTIVATING	Элемент активизируется в уведомлении LVN_ACTIVATE
LVIS_CUT	Запись выделена для операции вырезания или вставки (cut-and-paste)
LVIS_DROPHILITED	Запись выделена для переноса (drag-and-drop)
LVIS_FOCUSED	Запись имеет фокус ввода; естественно, что может быть выделено несколько записей одновременно, но только одна может иметь фокус ввода
LVIS_SELECTED	Запись выбрана

pszText — указатель на текстовую строку (буфер) (если параметр имеет значение LPSTR_TEXTCALLBACK, то это строка обратного вызова); cchTextMax задает размер буфера, на который указывает поле pszText, используется только для получения параметров записи в функции GetItem (в функции InsertItem значение этого параметра игнорируется); iImage задает индекс значка в списке изображений значков (если параметр имеет значение I_IMAGECALLBACK, то это запись обратного вызова); lParam задает 32-битное значение, ассоциированное с записью. Новое поле iIndent определяет величину отступа элемента (в "единицах" ширины изображения) и может использоваться только для самих элементов, но не для подэлементов.

Для удаления записи списка по индексу следует использовать функцию DeleteItem, а для единовременного удаления всех записей — DeleteAllItem.

Работа со столбцами

Если появляется необходимость реализовать поддержку режима отображения содержимого просмотра списка в виде таблицы, следует в обязательном порядке, независимо от количества полей представления записей списка, создать столбцы. Для определения требуемой ширины столбца следует воспользоваться функцией:

int CListCtrl::**GetStringWidth**(LPCTSTR lpsz) —

позволяет на основании строки текста определить минимальную ширину столбца, необходимую для ее вывода на экран целиком.

Для добавления (вставки) нового столбца служит функция:

int CListCtrl::**InsertColumn**(int nCol, const LV_COLUMN* pColumn)

и

int CListCtrl::**InsertColumn**(int nCol, LPCTSTR lpszColumnHeading,
int nFormat = LVCFMT _LEFT, int nWidth = −1, int nSubItem = −1) —

обе версии взаимозависимы. Параметр nCol определяет индекс нового столбца.

Как обычно, рассмотрим ту реализацию, которая в качестве параметров, в данном случае, для нового столбца, использует структуру.

Примечание

Использование второй версии несколько проще, и в примере, приведенном в конце раздела, используется именно вторая версия функции InsertColumn.

Эта структура называется LV_COLUMN и имеет следующее описание:

```
typedef struct _LVCOLUMN {
  UINT mask;
  int fmt;
  int cx;
  LPSTR pszText;
  int cchTextMax;
  int iSubItem;
#if (_WIN32_IE >= 0x0300)
  int iImage;
  int iOrder;
#endif
} LV_COLUMN;
```

Поле mask определяет, какие из остальных полей структуры заданы для описания столбца, и может содержать нуль или комбинацию следующих флагов:

LVCF_FMT	Поле fmt содержит данные
LVCF_SUBITEM	Поле iSubItem содержит данные
LVCF_IMAGE	Поле iImage содержит данные
LVCF_ORDER	Поле iOrder содержит данные
LVCF_TEXT	Поле pszText содержит данные
LVCF_WIDTH	Поле cx содержит данные

cx определяет ширину столбца (при значении –1 ширина не задается, т. е. содержимое столбца не будет отображаться); fmt задает режим выравнивания столбцов, причем этот режим относится как к заголовку столбца, если он задан и отображается, так и к содержимому полей всех записей просмотра списка (всего таких режимов три, и названия флагов, их определяющих, говорят сами за себя:

LVCFMT_BITMAP_ON_RIGHT	Битовый массив располагается справа от текста. Не относится к рисунку из списка образов, присвоенного элементу заголовка
LVCFMT_LEFT	Текст сдвинут влево
LVCFMT_CENTER	Текст располагается по центру
LVCFMT_RIGHT	Текст сдвинут вправо
LVCFMT_IMAGE	Для изображения элемента используется рисунок из списка изображений
LVCFMT_COL_HAS_IMAGES	Элемент заголовка имеет рисунок в списке изображений

pszText — указатель на строку, содержащую наименование заголовка столбца (при получении информации о столбце это поле должно указывать на буфер, куда будет помещено текущее название, длина буфера задается значением поля cchTextMax); iSubItem связывает столбец с полем записи; iImage определяет индекс изображения внутри списка образов, а iOrder — порядковый номер столбца, начиная с крайнего левого.

Для удаления столбца предназначена функция DeleteColumn. Для получения параметров столбца, а также для их изменения предназначены следующие функции.

BOOL CListCtrl::**GetColumn**(int nCol, LV_COLUMN* pColumn) —

возвращает параметры столбца.

BOOL CListCtrl::**SetColumn**(int nCol, const LV_COLUMN* pColumn) —

устанавливает новые параметры столбца.

int CListCtrl::**GetColumnWidth**(int nCol) —

возвращает ширину столбца.

BOOL CListCtrl::**SetColumnWidth**(int nCol, int cx) —

устанавливает ширину столбца.

В режиме просмотра списка в виде таблицы элемент управления имеет специальные заголовки столбцов, доступ к которым достигается посредством функции:

CHeaderCtrl* CListCtrl::**GetHeaderCtrl**( )

Параметры просмотра списка

Как уже упоминалось, стиль просмотра списка можно менять в любое время уже после его создания. Для этого, к сожалению, библиотека классов MFC не предусматривает соответствующие функции, поэтому следует использовать функции API — ::GetWindowLong и ::SetWindowLong. В следующем примере приведена функция, позволяющая это делать.

```
void CSaturnListCtrl::SetStyle(DWORD dwViewStyle)
{
  ASSERT(::IsWindow(m_hWnd));
  // Получаем текущие стили окна
  DWORD dwStyle = GetWindowLong(m_hWnd, GWL_STYLE);
  // Устанавливаем стиль, если только он не был задан ранее
  if ((dwStyle & LVS_TYPEMASK) != dwViewStyle)
    SetWindowLong(m_hWnd, GWL_STYLE,
    (dwStyle & ~LVS_TYPEMASK) | dwViewStyle);
}
```

Для работы с дополнительными стилями в классе CListCtrl служат функции SetExtendedStyle и SetExtendedStyle. Кроме того, класс CListCtrl предусматривает возможность установки и изменения цвета фона списка просмотра. Следующая пара функций предназначена для получения значения текущего цвета фона (GetBkColor) и установки нового (SetBkColor). Если значение текущего цвета равно CLR_NONE, т. е. цвет фона не задан, перерисовка списка просмотра осуществляется значительно медленнее, чем при задании цвета.

Цвет фона — не единственный цвет, значение которого можно получить или изменить. Функции GetTextColor и SetTextColor используются для работы с цветом букв текста, функции GetTextBkColor и SetTextBkColor — для работы с цветом фона текстовых строк, а функции GetBkImage и SetBkImage — для работы с цветом фона изображений.

Функции для работы со списком в целом

Следующие функции предназначены для управления выводом части записей списка в случае, если все они не могут быть расположены в окне просмотра списка.

BOOL CListCtrl::**GetViewRect**(LPRECT lpRect) —

возвращает размеры прямоугольника, ограничивающего все записи списка в режиме крупных или мелких значков.

BOOL CListCtrl::**GetItemRect**(int nItem, LPRECT lpRect, UINT nCode) —

возвращает параметры прямоугольника lpRect, ограничивающего весь или часть элемента. Параметры: nItem — индекс элемента, чьи параметры должны быть возвращены; nCode — часть элемента, для которого возвращаются параметры; может принимать одно из следующих значений:

LVIR_BOUNDS	Возвращаются размеры всего элемента, включая значок и текстовую метку
LVIR_ICON	Возвращаются размеры крупного или мелкого значка элемента
LVIR_LABEL	Возвращаются размеры текстовой метки элемента

BOOL CListCtrl::**GetSubItemRect**(int iItem, int iSubItem, int nArea, CRect& ref) —

параметры: iItem — индекс родительского элемента подэлемента; iItem — индекс подэлемента, параметры которого должны быть возвращены; nArea — часть элемента, для которого возвращаются параметры; допустимые значения аналогичны значениям аргумента nCode функции GetItemRect. Значения прямоугольника возвращаются в параметре ref.

int CListCtrl::**GetTopIndex**( ) —

возвращает индекс верхней видимой записи.

int CListCtrl::**GetCountPerPage**( ) —

возвращает количество записей, которые могут поместиться по вертикали в видимой части просмотра списка при просмотре в режиме списка или таблицы.

BOOL CListCtrl::**GetOrigin**(LPPOINT lpPoint) —

возвращает координаты, с которых начинается вывод элементов просмотра списка.

BOOL CListCtrl::**EnsureVisible**(int nItem, BOOL bPartialOK) —

гарантирует, что запись, заданная индексом (параметр nItem), будет видимой. Достигается это путем прокрутки. Параметр bPartialOK определяет допустимость отображения только части элемента. Если значение параметра равно TRUE, то прокрутка не производится, если запись уже частично видна.

(См. разд. "Просмотр списка с возможностью перемещения записей")

CSize CListCtrl::**SetIconSpacing**(int cx, int cy)

и

CSize CListCtrl::**SetIconSpacing**(CSize size) —

устанавливают новое расстояние между значками по осям x и y.

BOOL CListCtrl::**Scroll**(CSize size) —

осуществляет прокрутку по вертикали и горизонтали на заданное количество пикселей.

BOOL CListCtrl::**RedrawItems**(int nFirst, int nLast) —

заставляет просмотр списка перерисовать записи с индексами, лежащими в диапазоне от nFirst и до nLast включительно. Указанная группа записей *не* перерисовывается немедленно — это происходит по получении списком сообщения WM_PAINT.

Для немедленной перерисовки конкретной записи служит функция:

BOOL CListCtrl::**Update**(int nItem) —

перерисовывает заданную запись просмотра списка. При этом, возможно, производится переупорядочивание записей (в случае, если просмотр списка имеет стиль LVS_AUTOARRANGE).

BOOL CListCtrl::**Arrange**(UINT nCode) —

инициирует переупорядочивание записей. Способ упорядочивания задается параметром nCode, который может принимать одно из следующих значений:

LVA_ALIGNLEFT	Записи должны быть выровнены по левому краю окна
LVA_ALIGNTOP	Записи должны быть выровнены по верхнему краю окна
LVA_DEFAULT	Записи должны быть переупорядочены в соответствии с текущим установленным типом выравнивания
LVA_SNAPTOGRID	Значки должны быть помещены в ближайшую точку сетки для выравнивания

Указанные значения могут комбинироваться также со следующими:

LVA_SORTASCENDING	Сортировка записей в порядке возрастания
LVA_SORTDESCENDING	Сортировка записей в порядке убывания

Следующие две функции позволяют получить и установить выделенный элемент

int CListCtrl::**GetHotItem**( )

и

int CListCtrl::**SetHotItem**(int iIndex) —

параметр iIndex задает индекс текущего выделенного элемента в просмотре списка.

Рабочие области просмотра списка

В элементе управления "просмотр списка" реализована поддержка *рабочих областей* — прямоугольных виртуальных областей, используемых для упорядочивания элементов просмотра списка. В этом случае перемещение элементов возможно только внутри области. Каждая из таких областей может иметь свои горизонтальные полосы прокрутки.

Исходно все осталось как и раньше — рабочие области автоматически не создаются и взаимодействие с просмотром списка осуществляется как и прежде. Для создания (а также переопределения) такой области необходимо вызвать функцию:

void CListCtrl::**SetWorkAreas**(int nWorkAreas, LPRECT prc) —

параметр nWorkAreas определяет номер создаваемой рабочей области, prc указывает на адрес памяти, где записаны ее координаты (в единицах клиентской области).

Примечание

Создавать рабочие области можно только для режима крупных или мелких значков.

Получить координаты конкретной рабочей области можно с помощью функции:

void CListCtrl::**GetWorkAreas**(int nWorkAreas, LPRECT prc) —

имеет те же параметры, что и функция SetWorkAreas.

Общее число созданных в просмотре списка рабочих областей можно узнать воспользовавшись функцией GetNumberOfWorkAreas. Для одного списка просмотра можно создавать до LV_MAX_WORKAREAS (на настоящий момент равно 16) рабочих областей. Разные рабочие области могут пересекаться. При этом элемент будет относиться к той области, индекс которой меньше.

Списки изображений

Значки для записей просмотра списка содержатся в списках изображений, которые необходимо создать и определить, если есть необходимость сопровождать записи значками. Один список изображений содержит полнораз-

мерный значок (для режима крупных значков), второй — значки меньшего размера (для остальных режимов). Можно определить и третий список, который будет отображать определяемые приложением состояния записей.

Для работы со списком изображений предназначены две функции:

CImageList* CListCtrl::**GetImageList**(int nImageList)

и

CImageList* CListCtrl::**SetImageList**(CImageList* pImageList, int nImageList) —

первая возвращает указатель на текущий список изображений, вторая устанавливает текущий новый список изображений. Параметр nImageList в обеих функциях может принимать одно из следующих значений:

LVSIL_NORMAL	Список изображений с крупными значками
LVSIL_SMALL	Список изображений с мелкими значками
LVSIL_STATE	Список с изображениями для отображения состояний

Наличие значков только одного размера (только мелких или только крупных) позволяет при помощи масштабирования использовать имеющиеся значки для всех режимов вывода. В любом случае, для поддержки всех режимов отображения требуется создать оба списка изображений, причем важно, чтобы они имели одинаковое количество изображений, и соответствующие изображения располагались в одинаковой последовательности, поскольку одно и то же значение (номер по порядку) используется для идентификации необходимого значка в обоих списках.

Если список состояний изображений задан, то просмотр списка резервирует место слева от каждого значка в записи для изображения состояния. Приложение может использовать изображения состояний, например, выбран/не выбран, для индикации состояния записей, которые определяются приложением. Ненулевое значение в битах с 12 по 15 задает индекс изображения состояния, нуль обозначает отсутствие изображения состояния. В случаях, когда просмотр списка использует режим крупных значков, изображения состояний обычно не используются.

По умолчанию, когда уничтожается просмотр списка, уничтожаются и его списки изображений. Однако если просмотр списка имеет стиль LVS_SHAREIMAGELISTS, то приложение само отвечает за уничтожение списка изображений, когда он больше не нужен. Этот стиль, например, следует указывать в том случае, если один список изображений используется для нескольких просмотров списка.

BOOL CListCtrl::**GetCheck**(int nItem)

и

BOOL CListCtrl::**SetCheck**(int nItem, BOOL fCheck = TRUE) —

позволяют получить и, соответственно, установить состояние изображения элемента (видимое — fCheck = TRUE, или невидимое — fCheck = FALSE).

Виртуальные списки

Виртуальный список — это список с неограниченным числом записей. Такой список удобно использовать при работе с базами данных. Для создания виртуального списка служит функция CreateWindow или CreateWindowEx, которой в качестве части параметра dwStyle передается значение LVS_OWNERDATA (его можно использовать с большинством стилей, кроме LVS_SORTASCENDING и LVS_SORTDESCENDING). Кроме того, автоматически устанавливается стиль LVS_AUTOARRANGE.

В рассматриваемом классе для работы с виртуальными списками есть функция:

void CListCtrl::**SetItemCountEx**(int iCount,
DWORD dwFlags = LVSICF_NOINVALIDATEALL) —

устанавливает число элементов в просмотре списка (параметр iCount); dwFlags позволяет определить поведение виртуального списка после установки нового размера. Допустимы комбинации следующих значений:

LVSICF_NOINVALIDATEALL	Элемент управления не перерисовывается до того, как не обработается какой-либо его элемент — устанавливается по умолчанию
LVSICF_NOSCROLL	Позиция отображаемого элемента не изменяется

Основные и дополнительные поля

Каждая запись просмотра списка может состоять из изображения, показывающего текущее состояние записи, значка, основного поля (собственно записи) и значения, определяемого приложением. С каждой записью может быть связано одно или более дополнительных полей. *Дополнительное поле* — это текстовая строка, которая в режиме таблицы (см. раздел "Режимы вывода" выше) может быть выведена в столбце справа от значка и основного поля. Все записи в одном просмотре списка могут иметь одинаковое количество полей. Используя сообщения просмотра списка, можно добавлять, изменять, получать информацию и удалять записи. Можно также находить запись с заданными атрибутами.

Структура LVITEM определяет записи, включая основное поле, и дополнительные поля просмотра списка. Поле структуры iItem задает индекс записи (нумерация начинается с нуля), iSubItem задает индекс дополнительного поля (нумерация начинается с единицы). Дополнительные поля задают текст, номер значка в списке изображений, состояние и данные записи. Под данными записи в данном случае понимается определяемое приложением значение, связанное с записью просмотра списка.

Записи по запросу

Записи по запросу (иначе, записи обратного вызова, Callback Items) — это записи, для которых приложение (а не элемент управления), сохраняет

текст или значки, или то и другое. Несмотря на то что просмотр списка сохраняет их, можно создать запись по запросу, например, для исключения дублирования информации, в частности, текста. Информация для таких записей будет запрашиваться по мере необходимости. Для реализации этого механизма необходимо предусмотреть реализацию обработки уведомлений LVN_GETDISPINFO и LVN_GETDISPINFO.

Кроме того, приложение может взять на себя (временно или постоянно) обработку ряда событий. В этом случае речь идет о задании или изменении маски обратного вызова всего элемента управления. Маска определяет, какие биты состояния записей поддерживаются приложением. По умолчанию маска обратного вызова равна нулю, это означает, что элемент управления самостоятельно отслеживает все изменения состояния записей.

Для получения текущей маски обратного вызова или ее изменения используются две следующие функции:

UINT CListCtrl::**GetCallbackMask**( )

и

BOOL CListCtrl::**SetCallbackMask**(UINT nMask) —

параметр nMask, а также возвращаемое значение функции GetCallbackMask могут содержать любую комбинацию следующих значений:

LVIS_CUT	Требует обработки выбора записи списка для выполнения вырезания и вставки (cut-and-paste)
LVIS_DROPHILITED	Требует обработки назначения целевой записи списка для выполнения перетаскивания (drag-and-drop)
LVIS_FOCUSED	Требует обработки получения фокуса записи списка
LVIS_SELECTED	Требует обработки при выборе записи списка
LVIS_OVERLAYMASK	Приложение берет на себя выбор индекса значка в списке изображений для каждого элемента списка
LVIS_STATEIMAGEMASK	Приложение берет на себя выбор индекса значка в списке изображений состояний для каждого элемента списка

Изменение содержимого просмотра списка

Следующая группа функций предназначена для работы с конкретными записями, а также для их добавления, изменения и удаления.

int CListCtrl::**GetItemCount**( ) —

возвращает число записей в просмотре списка.

void CListCtrl::**SetItemCount**(int nItems) —

подготавливает просмотр списка к добавлению большого количества новых записей. При этом перед добавлением большого количества записей элемент управления один раз производит перераспределение памяти, вместо того, чтобы делать это всякий раз при добавлении новой записи.

UINT CListCtrl::**GetSelectedCount**( ) —

возвращает число выбранных записей в просмотре списка (для списков множественного выбора).

int CListCtrl::**SetSelectionMark**(int iIndex)

и

int CListCtrl::**GetSelectionMark**( ) —

устанавливает или возвращает индекс первого выделенного элемента при множественном выборе.

После вставки новой записи, ее атрибуты можно запросить или изменить. Для этого применяются функции, использующие структуру LVITEM: GetItem, SetItem и SetItem.

Следующие функции позволяют работать с частью атрибутов записи списка без использования структуры LV_ITEM и громоздких функций, описанных ранее:

DWORD CListCtrl::**GetItemData**(int nItem) —

возвращает 32-битное значение, связанное с записью.

BOOL CListCtrl::**SetItemData**(int nItem, DWORD dwData) —

связывает новое 32-битное значение с записью.

UINT CListCtrl::**GetItemState**(int nItem, UINT nMask) —

возвращает текущее состояние записи в просмотре списка.

BOOL CListCtrl::**SetItemState**(int nItem, LV_ITEM* pItem)

и

BOOL CListCtrl::**SetItemState**(int nItem, UINT nState, UINT nMask) —

изменяют текущее состояние записи в просмотре списка.

Для получения копии текстовой строки записи или, наоборот, установки ее нового значения служат функции:

int CListCtrl::**GetItemText**(int nItem, int nSubItem, LPTSTR lpszText, int nLen),

CString CListCtrl::**GetItemText**(int nItem, int nSubItem)

и

BOOL CListCtrl::**SetItemText**(int nItem, int nSubItem, LPTSTR lpszText) —

позволяют работать не только с основным полем (nSubItem равно 0), но и со всеми дополнительными полями.

Следующие функции предназначены для управления положением записи в окне просмотра списка:

BOOL CListCtrl::**GetItemPosition**(int nItem, LPPOINT lpPoint) —

возвращает расположение заданной записи в просмотре списка.

BOOL CListCtrl::**SetItemPosition**(int nItem, LPPOINT lpPoint) —

помещает запись в заданную позицию в просмотре списка.

Примечание

В разных режимах позиция определяется по-разному, например, в режиме крупных значков начало координат определяется левым верхним углом всего поля, требуемого для вывода всех записей, даже если это поле смещено вверх относительно окна просмотра списка; в режиме таблицы начало координат определяется уже самим окном просмотра списка, и позиция записи может иметь, в том числе, и отрицательные значения по каждой из осей.

Поиск и сортировка записей

Первая функция этой группы предназначена для нахождения записи с определенными свойствами:

int CListCtrl::**GetNextItem**(int nItem, int nFlags) —

в случае успешного завершения возвращает индекс найденной записи, иначе –1. Параметр nItem задает индекс записи, с которой начинается поиск, сама она из поиска исключается. Если задано значение –1, то поиск идет с начала списка. Параметр nFlags задает пространственное расположение записи, поиск которой осуществляется, и ее состояние. Расположение может определяться следующими флагами:

LVNI_ABOVE	Поиск идет вверх от указанной записи
LVNI_ALL	Поиск идет по индексам (по умолчанию)
LVNI_BELOW	Поиск идет вниз от указанной записи
LVNI_TOLEFT	Поиск идет влево от указанной записи
LVNI_TORIGHT	Поиск идет вправо от указанной записи

Состояние может не указываться или быть комбинацией следующих флагов:

LVNI_DROPHILITED	Запись имеет установленный флаг LVIS_DROPHILITED
LVNI_FOCUSED	Запись имеет установленный флаг LVIS_FOCUSED
LVNI_HIDDEN	Запись имеет установленный флаг LVIS_HIDDEN
LVNI_MARKED	Запись имеет установленный флаг LVIS_MARKED
LVNI_SELECTED	Запись имеет установленный флаг LVIS_SELECTED

Если у записи, которая проверяется на соответствие заданным условиям, установлены не все заданные в параметре nFlags флаги, то поиск продолжается со следующей записи.

POSITION CListCtrl::**GetFirstSelectedItemPosition**( )

и

int CListCtrl::**GetNextSelectedItem**(POSITION& pos) —

позволяют получить индекс, соответственно, первого и последующих элементов просмотра списка. Если функции возвращают NULL, то в элементе управления нет выбранных элементов.

int CListCtrl::**FindItem**(LV_FINDINFO* pFindInfo, int nStart = −1) —

осуществляет поиск строки, имеющей заданные характеристики, и возвращает индекс найденной строки или –1. Параметр nStart задает индекс записи, с которой должен начаться поиск, при этом сама она из поиска исключается, или –1, если нужно искать с начала списка. Параметр pFindInfo является указателем на структуру LV_FINDINFO, содержащую информацию для осуществления поиска.

Просмотр списка в ряде случаев никак не реагирует на действия мышью, если только при этом не происходит изменение состояния его записей. Однако класс CListCtrl позволяет определять запись по координатам, и, если требуется, узнать, например, в обработчике нажатия ее кнопки, на какой записи находится указатель мыши, следует воспользоваться функцией HitTest или SubItemHitTest. Но надо иметь в виду, что, поскольку "система координат" списка зависит от режима отображения, если одновременно не все элементы просмотра списка могут быть отображены на экране, для правильной работы этих функций следует осуществлять преобразование координат.

Следующая функция осуществляет сортировку записей просмотра списка, используя определяемую приложением функцию для сравнения записей:

BOOL CListCtrl::**SortItems**(PFNLVCOMPARE pfnCompare, DWORD dwData) —

индексы записей при этом изменяются, отражая новый порядок записей в списке. Параметр pfnCompare указывает на определяемую приложением функцию сравнения записей. Эта функция вызывается всякий раз в процессе сортировки, когда необходимо сравнить две записи. Второй параметр позволяет передавать в функцию сравнения некоторое значение в качестве третьего параметра.

Вот как должна быть описана функция сравнения:

int CALLBACK **CompareFunc**(LPARAM lParam1, LPARAM lParam2, LPARAM lParamSort) —

первые два параметра содержат 32-битные данные, ассоциированные со сравниваемыми записями. Последний параметр является копией значения парметра dwData функции CListCtrl::SortItems. Функция возвращает отрицательное значение, если первая запись должна предшествовать второй записи, положительное значение, если первая запись должна быть расположена после второй записи, и нуль, если записи эквивалентны.

Редактирование надписей записей

Поддержка пользовательского редактирования надписей записей осуществляется заданием стиля LVS_EDITLABELS и обработкой уведомления LVN_ENDLABELEDIT; что и отражено в примере этого раздела. Обработчик уведомления получает указатель на новую строку, которую ввел пользователь. Чтобы фактически осуществить замену предыдущего наименования,

следует самостоятельно заменить содержимое записи, а также при необходимости произвести дополнительные действия, например, при изменении копии имени файла, содержащейся в записи просмотра списка, требуется также изменить его имя и на диске или ином носителе. В случае, если значение указателя равно нулю, — пользователь отменил изменения, нажав клавишу <Esc> — ничего делать не придется.

Класс CListCtrl предоставляет дополнительные функции, расширяющие возможности поддержки процесса редактирования надписей записей:

CEdit* CListCtrl::**GetEditControl**( ) —

возвращает дескриптор редактора текста, используемого для редактирования надписи записи. В случае удачного завершения возвращается указатель на объект класса CEdit, в случае неудачи — NULL.

CEdit* CListCtrl::**EditLabel**(int nItem) —

позволяет программно инициировать редактирование надписи записи в просмотре списка, если просмотр списка имеет стиль LVS_EDITLABELS.

Обработка уведомлений

Когда пользователь щелкает на колонке заголовков, перемещает значки, редактирует надписи и т. п., просмотр списка посылает родительскому окну уведомления (их можно обрабатывать в функции родительского окна OnNotify):

LVN_BEGINDRAG	Сигнализирует о начале перетаскивания при помощи левой кнопки мыши (drag-and-drop)
LVN_BEGINLABELEDIT	Сигнализирует о начале редактирования надписи записи
LVN_BEGINRDRAG	Сигнализирует о начале перетаскивания при помощи правой кнопки мыши (drag-and-drop)
LVN_COLUMNCLICK	Сигнализирует о том, что пользователь нажал кнопку мыши, когда курсор находился на элементе заголовка в режиме таблицы
LVN_DELETEALLITEMS	Сигнализирует об удалении всех записей просмотра списка
LVN_DELETEITEM	Сигнализирует об удалении конкретной записи просмотра списка
LVN_ENDLABELEDIT	Сигнализирует о завершении редактирования надписи записи
LVN_GETDISPINFO	Запрашивает информацию о записи по запросу просмотра списка для ее отображения
LVN_INSERTITEM	Сигнализирует о добавлении новой записи к просмотру списка

LVN_ITEMCHANGED	Сигнализирует об изменении записи просмотра списка
LVN_ITEMCHANGING	Запрашивает подтверждение изменения записи просмотра списка
LVN_KEYDOWN	Сигнализирует о нажатии клавиши на клавиатуре
LVN_SETDISPINFO	Уведомляет родительское окно о необходимости изменения информации о записи по запросу, которая предоставляется приложением

Обработка уведомлений происходит как и для других элементов управления, за исключением того, что приложение получает информацию в структуре LVDISPINFO, которая, кроме обычного для уведомлений заголовка, содержит структуру LVITEM со всей необходимой для обработки информацией:

```
BOOL CMercuryWnd::OnNotify(
  WPARAM wParam, LPARAM lParam, LRESULT* pResult)
{
  ...
  LV_DISPINFO *pdi = (LV_DISPINFO *)lParam;
  LV_ITEM item = pdi->item;
  ...
}
```

Некоторые уведомляющие сообщения работают с другой структурой, а именно со структурой NM_LISTVIEW:

```
NM_LISTVIEW *plv = (NM_LISTVIEW *)lParam;
```

Просмотр списка с возможностью перемещения записей

Перемещение записей возможно реализовать только для режима крупных или мелких значков, что вполне естественно. Главное — нужно помнить, что индексы записей после изменения положения записей не будут отражать их фактический порядок на экране — порядок размещения записей в просмотре списка автоматически не изменяется.

Для поддержки перемещения записей просмотра списка требуется обрабатывать следующие уведомления: LVN_BEGINDRAG, LVN_BEGINRDRAG, а также сообщения от мыши: отпускание левой и правой кнопок, а также перемещение курсора.

Кроме того, следует создать еще один список изображений, который будет содержать элемент для визуализации перемещения записи списка при помощи функции CreateDragImage:

CImagelist* CListCtrl::**CreateDragImage**(int nItem, LPPOINT lpPoint) —

параметр nItem задает запись при помощи индекса, а lpPoint — текущие координаты левого верхнего угла записи.

Переопределяемые функции

Для реализации самоотображаемого списка следует переопределить функцию DrawItem, которая, как и для других элементов управления, получает всю необходимую информацию для осуществления рисования записей просмотра списка.

ListCtrl — пример приложения

В этом разделе приведен пример использования просмотра списка, в качестве записей которого используется информация о файлах текущего (для приложения) каталога, а именно — имя, тип (определяемый по расширению), размер в килобайтах, а также дата и время последней операции записи.

Приложение реализует практически все рассмотренные в предыдущих разделах механизмы, включая:

- Возможность отображения во всех четырех режимах просмотра
- Сортировку по всем четырем используемым атрибутам файла
- Редактирование в режиме таблицы надписи записи (файла) без сохранения изменений на диске
- Перетаскивание записей мышью
- Переупорядочивание записей, необходимое после перетаскивания

```
// Файл ListCtrlFrm.cpp
// Функция сортировки записей, тип сортировки определяется
// значением третьего параметра
int CALLBACK SortFunc(LPARAM lParam1,
  LPARAM lParam2, LPARAM dwType)
{
  WIN32_FIND_DATA *item1 = (WIN32_FIND_DATA *)lParam1;
  WIN32_FIND_DATA *item2 = (WIN32_FIND_DATA *)lParam2;
  switch (dwType)
  {
  case ID_VIEW_ARRANGE_BYNAME:
    return strcmp(item1->cFileName, item2->cFileName);
  case ID_VIEW_ARRANGE_BYTYPE:
    return strcmp(strrchr(item1->cAlternateFileName, '.'),
      strrchr(item2->cAlternateFileName, '.'));
```

```
  case ID_VIEW_ARRANGE_BYSIZE:
    return item1->nFileSizeLow — item2->nFileSizeLow;
  case ID_VIEW_ARRANGE_BYDATE:
    return CompareFileTime(&item1->ftLastWriteTime,
      &item2->ftLastWriteTime);
  default:
    ASSERT(FALSE);
  }

  return 0;
}
// CListCtrlFrame
CListCtrlFrame::CListCtrlFrame()
{
  // Инициализация переменных для реализации перетаскивания записей
  m_bDragging = FALSE;
  m_pImageListDrag = NULL;
}
// Функция изменения стиля просмотра списка
// для смены режимов отображения
void CListCtrlFrame::SetStyle(DWORD dwViewStyle)
{
  ASSERT(::IsWindow(m_List.m_hWnd));
  DWORD dwStyle = GetWindowLong(m_List.m_hWnd, GWL_STYLE);
  if ((dwStyle & LVS_TYPEMASK) != dwViewStyle)
    SetWindowLong(m_List.m_hWnd, GWL_STYLE,
    (dwStyle & ~LVS_TYPEMASK) | dwViewStyle);
}
BEGIN_MESSAGE_MAP(CListCtrlFrame, CFrameWnd)
  //{{AFX_MSG_MAP(CListCtrlFrame)
  ON_WM_CREATE()
  ON_COMMAND(ID_VIEW_LARGEICONS, OnViewLargeIcons)
  ON_COMMAND(ID_VIEW_SMALLICONS, OnViewSmallIcons)
  ON_COMMAND(ID_VIEW_LIST, OnViewList)
  ON_COMMAND(ID_VIEW_DETAILS, OnViewDetails)
  ON_COMMAND(ID_VIEW_ARRANGE_BYNAME, OnViewArrangeByName)
  ON_COMMAND(ID_VIEW_ARRANGE_BYTYPE, OnViewArrangeByType)
  ON_COMMAND(ID_VIEW_ARRANGE_BYSIZE, OnViewArrangeBySize)
  ON_COMMAND(ID_VIEW_ARRANGE_BYDATE, OnViewArrangeByDate)
  ON_COMMAND(ID_VIEW_ARRANGE_AUTOARRANGE,
```

```
    OnViewArrangeAutoArrange)
  ON_WM_MOUSEMOVE()
  ON_WM_LBUTTONUP()
  ON_WM_RBUTTONUP()
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()
int CListCtrlFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
  if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
    return -1;
  CRect rect;
  GetClientRect(rect);
  // Создание и инициализация просмотра списка
  m_List.Create(LVS_ICON | LVS_REPORT | LVS_EDITLABELS |
    WS_VISIBLE, rect, this, 1000);
  // Создание и заполнение списков изображений
  m_ImageNormal.Create(32, 32, TRUE, 1, 1);
  m_ImageNormal.Add(AfxGetApp()->LoadIcon(IDI_FILE));
  m_ImageSmall.Create(16, 16, TRUE, 1, 1);
  m_ImageSmall.Add(AfxGetApp()->LoadIcon(IDI_FILE));
  m_List.SetImageList(&m_ImageNormal, LVSIL_NORMAL);
  m_List.SetImageList(&m_ImageSmall, LVSIL_SMALL);
  // Добавление в просмотр списка четырех колонок для
  // реализации режима таблицы
  m_List.InsertColumn(0, _T("Name"),
    LVCFMT_LEFT, rect.Width() / 3);
  m_List.InsertColumn(1, _T("Type"),
    LVCFMT_LEFT, 2 * rect.Width() / 9);
  m_List.InsertColumn(2, _T("Size"),
    LVCFMT_RIGHT, rect.Width() / 9);
  m_List.InsertColumn(3, _T("Modified"),
    LVCFMT_LEFT, rect.Width() / 3);
  // Поиск файлов и создание записей с
  // информацией о найденных файлах
  WIN32_FIND_DATA FindFileData;
  HANDLE hFile = FindFirstFile("*.*", &FindFileData);
  if (hFile != INVALID_HANDLE_VALUE)
  {
    LV_ITEM item;
    item.iItem = 0;
```

```
item.iImage = 0;
CString csText;
SYSTEMTIME stTime;
do    {
  // Каталоги игнорируем
  if (FindFileData.dwFileAttributes &
    FILE_ATTRIBUTE_DIRECTORY)
    continue;
  item.mask = LVIF_TEXT | LVIF_IMAGE | LVIF_PARAM;
  // Для реализации сортировки дополнительно сохраняем
  // информацию о найденном файле
  WIN32_FIND_DATA *lpFindFileData = new WIN32_FIND_DATA;
  *lpFindFileData = FindFileData;
  item.lParam = (DWORD)lpFindFileData;
  // Имя файла
  item.iSubItem = 0;
  item.pszText = FindFileData.cFileName;
  m_List.InsertItem(&item);
  item.mask = LVIF_TEXT;
  // Тип файла
  item.iSubItem = 1;
  csText = (strrchr(FindFileData.cAlternateFileName, '.') + 1);
  csText += " File";
  item.pszText = (LPTSTR)(LPCTSTR)csText;
  m_List.SetItem(&item);
  // Размер файла
  item.iSubItem = 2;
  csText.Format("%dKB", FindFileData.nFileSizeLow / 1024);
  item.pszText = (LPTSTR)(LPCTSTR)csText;
  m_List.SetItem(&item);
  // Дата и время последней записи в файл
  item.iSubItem = 3;
  FileTimeToSystemTime(&FindFileData.ftLastWriteTime,
    &stTime);
  csText = CTime(stTime).Format(" %d/%m/%Y %H:%M:%S");
  item.pszText = (LPTSTR)(LPCTSTR)csText;
  m_List.SetItem(&item);
  item.iItem++;
}
```

```
    while (FindNextFile(hFile, &FindFileData));
  }
  return 0;
}
```

После загрузки приложения появится следующее окно (рис. 26.4).

Рис. 26.4. Начальное состояние окна приложения ListCtrl с демонстрацией реализованных возможностей

```
// Функции смены режима отображения
void CListCtrlFrame::OnViewLargeIcons()
{
  SetStyle(LVS_ICON);
}
void CListCtrlFrame::OnViewSmallIcons()
{
  SetStyle(LVS_SMALLICON);
}
void CListCtrlFrame::OnViewList()
{
  SetStyle(LVS_LIST);
}
void CListCtrlFrame::OnViewDetails()
{
  SetStyle(LVS_ICON | LVS_REPORT);
}
// Функции переупорядочивания (сортировки) по значениям
// различных атрибутов файла
void CListCtrlFrame::OnViewArrangeByName()
{
  m_List.SortItems(SortFunc, ID_VIEW_ARRANGE_BYNAME);
```

```
}
void CListCtrlFrame::OnViewArrangeByType()
{
  m_List.SortItems(SortFunc, ID_VIEW_ARRANGE_BYTYPE);
}
void CListCtrlFrame::OnViewArrangeBySize()
{
  m_List.SortItems(SortFunc, ID_VIEW_ARRANGE_BYSIZE);
}
void CListCtrlFrame::OnViewArrangeByDate()
{
  m_List.SortItems(SortFunc, ID_VIEW_ARRANGE_BYDATE);
}
// Функция переупорядочивания - привязки записей списка
// к сетке привязки
void CListCtrlFrame::OnViewArrangeAutoArrange()
{
  m_List.Arrange(LVA_SNAPTOGRID);
}
// Функция обработки извещений, для поддержки:
//   - редактирования названий записи;
//   - перетаскивания записей
BOOL CListCtrlFrame::OnNotify(
  WPARAM wParam, LPARAM lParam, LRESULT* pResult)
{
  NMHDR *nmhdr = (NMHDR *)lParam;
  if (nmhdr->hwndFrom == m_List.m_hWnd)
  {
    LV_DISPINFO *pdi = (LV_DISPINFO *)lParam;
    // Для поддержки перетаксивания записей, необходимо, как ни странно,
    // работать со структурой NM_LISTVIEW
    NM_LISTVIEW *plv = (NM_LISTVIEW *)lParam;
    CPoint ptItem, ptAction, ptImage;
    switch (nmhdr->code)
    {
    case LVN_ENDLABELEDIT:
      if (pdi->item.pszText != NULL)
        m_List.SetItemText(pdi->item.iItem, 0, pdi->item.pszText);
      break;
    case LVN_BEGINDRAG:
```

```
    case LVN_BEGINRDRAG:
      ASSERT(!m_bDragging);
      m_bDragging = TRUE;
      m_iItemDrag = plv->iItem;
      ptAction = plv->ptAction;
      m_List.GetItemPosition(m_iItemDrag, &ptItem);
      m_List.GetOrigin(&m_ptOrigin);
      m_pImageListDrag =
        m_List.CreateDragImage(m_iItemDrag, &ptImage);
      m_sizeDelta = ptAction - ptImage;
      m_ptHotSpot = ptAction - ptItem + m_ptOrigin;
      m_pImageListDrag->DragShowNolock(TRUE);
      m_pImageListDrag->SetDragCursorImage(0, m_ptHotSpot);
      m_pImageListDrag->BeginDrag(0, CPoint(0, 0));
      ptAction -= m_sizeDelta;
      m_pImageListDrag->DragEnter(&m_List, ptAction);
      m_pImageListDrag->DragMove(ptAction);
      SetCapture();
      break;
    }
  }
  return CFrameWnd::OnNotify(wParam, lParam, pResult);
}
// Функции обработки сообщений от мыши
void CListCtrlFrame::OnMouseMove(UINT nFlags, CPoint point)
{
  if (m_bDragging)
    m_pImageListDrag->DragMove(point - m_sizeDelta);
  CFrameWnd::OnMouseMove(nFlags, point);
}
void CListCtrlFrame::OnButtonUp(CPoint point)
{
  if (m_bDragging)
  {
    DWORD  dwStyle;
    dwStyle = GetWindowLong(m_List.m_hWnd, GWL_STYLE) &
      LVS_TYPEMASK;
    m_bDragging = FALSE;
    m_pImageListDrag->DragLeave(&m_List);
    m_pImageListDrag->EndDrag();
```

```
    if (dwStyle & (LVS_REPORT | LVS_LIST))
      MessageBeep((UINT)-1);
    else if (dwStyle == LVS_ICON || dwStyle == LVS_SMALLICON)
    {
      point -= m_ptHotSpot;
      point += m_ptOrigin;
      m_List.SetItemPosition(m_iItemDrag, point);
    }
    ReleaseCapture();
  }
}
void CListCtrlFrame::OnLButtonUp(UINT nFlags, CPoint point)
{
  OnButtonUp(point);
  CFrameWnd::OnLButtonUp(nFlags, point);
}
void CListCtrlFrame::OnRButtonUp(UINT nFlags, CPoint point)
{
  OnButtonUp(point);
  CFrameWnd::OnRButtonUp(nFlags, point);
}
// Поскольку память для реализации хранения информации о файлах выделена
// динамически, при уничтожении просмотра списка нужно ее освободить
BOOL CListCtrlFrame::DestroyWindow()
{
  for (int i = 0; i < m_List.GetItemCount(); i++)
    delete (WIN32_FIND_DATA *)m_List.GetItemData(i);
  return CFrameWnd::DestroyWindow();
}
```

Рис. 26.5. Окно приложения ListCtrl в режиме крупных значков после перетаскивания записи и перед переупорядочиванием

Просмотр дерева

Кроме возможности вывода записей последовательных списков при помощи таких элементов, как список или просмотр списка, Windows предоставляет возможность отображения (и управления) списков, организованных по принципу дерева. Для этого поддерживается элемент управления "просмотр дерева" (TREE VIEW). Первое, для чего этот элемент управления использует система Windows, — это организация просмотра дерева каталогов, а также различных индексов в справочных файлах и т. п.

Древовидные структуры предполагают наличие некоторой иерархии узлов и, естественно, что данный элемент управления может применяться только для информации, имеющей иерархическую структуру. Для создания этого элемента управления в библиотеке MFC предназначен класс CTreeCtrl. Каждая запись просмотра дерева состоит из текстовой строки (надписи) и заданного битового изображения, каждая запись может иметь одну или более подзаписей, связанных с ней. Щелчком на записи пользователь может развернуть или свернуть список подзаписей, связанных с записью. На экране подзаписи могут соединяться с записью линиями для того, чтобы наглядно представлялась их взаимосвязь.

Любая запись просмотра дерева может иметь связанные с ней подзаписи, которые называются *дочерними записями*, а сама запись называется *родительской записью*. При выводе на экран дочерние записи изображаются ниже родительской и со сдвигом в сторону для наглядного представления их зависимости от родительской записи. Запись, которая не имеет родительской и, следовательно, лежит на самом верхнем уровне иерархии, называется *корневой записью*.

Кроме класса CTreeCtrl, библиотека MFC содержит еще и класс CTreeView. Класс CTreeCtrl предназначен для создания просмотра дерева в блоке диалога, в особенности при наличии в блоке диалога других элементов управления. Напротив, если просмотр дерева должен быть создан в архитектуре "документ/представление" и будет занимать собой все окно, автоматически изменяя свои размеры при изменении размеров родительского окна, обрабатывать сообщения от меню и т. п., то необходимо использовать класс CTreeView.

Создание просмотра дерева

Для начала просмотр дерева необходимо создать. Если он задан в шаблоне блока диалога, то он будет автоматически создан при создании блока диалога. Если же он создается как дочернее окно в другом окне, то необходимо

воспользоваться функцией Create для создания соответствующего элемента управления Windows и связывания его с объектом класса CTreeCtrl:

BOOL CTreeCtrl::**Create**(DWORD dwStyle, const RECT& rect, CWnd* pParentWnd,
UINT nID) —

параметр dwStyle может содержать любые комбинации стилей элементов управления, а также следующие, свойственные рассматриваемому элементу управления:

TVS_HASLINES	Дочерние записи будут соединяться на экране линиями со своими родительскими записями
TVS_LINESATROOT	Записи первого уровня будут соединены с вершиной дерева, в противном случае, они всегда будут отображать свои дочерние записи

Оба эти стиля придают содержимому элемента управления вид древовидной структуры.

TVS_HASBUTTONS	Добавляет слева от каждой записи кнопку для сворачивания-разворачивания ветки дочерних записей (если для нее допустима операция разворачивания); кнопка содержит знак "+", если запись можно развернуть до следующего уровня, и знак "–", если ветвь дерева развернута, при этом дочерние записи могут находиться в свернутом состоянии
TVS_EDITLABELS	Позволяет пользователю редактировать записи непосредственно в просмотре дерева
TVS_SHOWSELALWAYS	Выделение с выбранной записи не снимается, даже если просмотр дерева теряет фокус ввода
TVS_DISABLEDRAGDROP	Просмотр дерева препятствует отправке уведомления TVN_BEGINDRAG
TVS_NOTOOLTIPS	Элемент управления не использует всплывающие подсказки
TVS_SINGLEEXPAND	Изменение выбора в просмотре списка приводит к его автоматическому раскрытию и к сворачиванию выбранной перед этим записи; одиночный щелчок на закрытом элементе раскрывает его, и наоборот, раскрытый элемент сворачивается

rect задает размер и расположение просмотра дерева; pParentWnd задает родительское окно (значение этого параметра не должно быть равно NULL); nID задает идентификатор просмотра дерева.

Списки изображений

Каждая запись просмотра дерева может иметь два ассоциированных с ней значка. Первый выводится на экран, когда запись развернута, второй — когда запись свернута (например, значки открытой и закрытой папок).

Кроме двух списков, описанных выше, приложение может задать список изображений для обозначения состояний записи. В этом случае просмотр дерева резервирует место слева от записи для размещения данного изображения.

Для работы с изображениями, связанными с просмотром дерева, предназначены функции:

CImageList* CTreeCtrl::**GetImageList**(UINT nImage) —

возвращает указатель на список изображений, связанных с просмотром дерева, или NULL в случае неудачного завершения. Параметр nImage задает тип списка и может принимать одно из следующих значений:

TVSIL_NORMAL	Предписывает возвращать список обычных изображений, который содержит изображения для выбранных и невыбранных записей
TVSIL_STATE	Предписывает возвращать список изображений, который содержит изображения для определяемых пользователем состояний

CImageList* CTreeCtrl::**SetImageList**(CImageList* pImageList, int nImageListType) —

устанавливает новый список изображений для просмотра дерева, тип списка задается параметром nImageListType.

Когда ветка дочерних записей развернута, дочерние записи изображаются с некоторым сдвигом относительно родительской записи. Для определения и изменения сдвига дочерней записи относительно родительской предназначены функции:

UINT CTreeCtrl::**GetIndent**( ) —

возвращает сдвиг (в пикселях) дочерней записи относительно родительской.

void CTreeCtrl::**SetIndent**(UINT nIndent) —

задает сдвиг (в пикселях) дочерней записи относительно родительской. Если задается значение, меньшее определяемого системой, то функция игнорируется.

Изменение содержимого дерева

Следующая группа функций предназначена для работы с конкретными записями, а также для их добавления, изменения и удаления:

UINT CTreeCtrl::**GetCount**( ) —

возвращает число записей, вставленных в просмотр дерева.

UINT CTreeCtrl::**GetVisibleCount**( ) —

возвращает текущее количество видимых записей просмотра дерева.

Для удаления всех записей из просмотра дерева служит функция DeleteAllItems.

Начальное положение записи при ее добавлении в просмотр дерева определяется функцией InsertItem. При вызове этой функции задаются дескриптор родительской записи и дескриптор записи, после которой должна быть вставлена новая запись. Последний должен быть дескриптором одной из

дочерних записей, принадлежащих родительской записи, или принимать одно из следующих значений: TVI_FIRST, TVI_LAST или TVI_SORT. Значения TVI_FIRST и TVI_LAST определяют, что запись должна быть помещена первой или последней в списке дочерних записей родительской записи. Если задано значение TVI_SORT, то после вставки записи, дочерние записи, принадлежащие данной родительской записи, будут отсортированы по алфавиту.

Вставку новых записей осуществляет функция:

TREEITEM CTreeCtrl::**InsertItem**(LPTV_INSERTSTRUCTlpInsertStruct),
HTREEITEM CTreeCtrl::**InsertItem**(UINT nMask, LPCTSTR lpszItem, int nImage, int nSelectedImage, UINT nState, UINT nStateMask, LPARAM lParam, HTREEITEM hParent, HTREEITEM hInsertAfter)
HTREEITEM CTreeCtrl::**InsertItem**(LPCTSTR lpszItem, HTREEITEM hParent = TVI_ROOT, HTREEITEM hInsertAfter = TVI_LAST)
HTREEITEM CTreeCtrl::**InsertItem**(LPCTSTR lpszItem, int nImage, int nSelectedImage, HTREEITEM hParent = TVI_ROOT, HTREEITEM hInsertAfter = TVI_LAST) —

все версии функции вставляют новую запись в просмотр дерева и возвращают дескриптор вставленной записи. Первые два варианта практически аналогичны по своим возможностям. Два оставшихся обеспечивают просмотр дерева лишь частью информации о новой записи. Однако их целесообразно использовать в относительно простых реализациях, когда от просмотра дерева не требуется поддержка всех функциональных возможностей. Существуют также функции, позволяющие впоследствии дополнить информацию о записи.

Чтобы не дублировать описание, приведем более подробно только описание структур TVINSERTSTRUCT и TVITEM, которые вместе содержат полный набор параметров, описывающих добавляемую запись, и необходимых для реализации добавления.

```
typedef struct _TVINSERTSTRUCT {
  HTREEITEM hParent;
  HTREEITEM hInsertAfter;
#if (_WIN32_IE >= 0x0400)
  union {
    TVITEMEX itemex;
    TVITEM item;
  } DUMMYUNIONNAME;
#else
  TVITEM item;
#endif
} TVINSERTSTRUCT;
```

Поле hParent задает дескриптор родительской записи. Если параметр равен TVI_ROOT или NULL, то запись вставляется как корневая. Поле hInsertAfter задает дескриптор записи, после которой запись должна быть вставлена, и может принимать также следующие значения:

TVI_FIRST	Запись вставляется в начало списка
TVI_LAST	Запись вставляется в конец списка
TVI_SORT	Запись вставляется в список в алфавитном порядке

Поле item является структурой TVITEM, а поле iItemEx — структурой TVITEMEX. Именно в этом поле содержится информация о добавляемой записи.

```
typedef struct _TVITEM {
  UINT mask;
  HTREEITEM hItem;
  UINT state;
  UINT stateMask;
  LPSTR pszText;
  int cchTextMax;
  int iImage;
  int iSelectedImage;
  int cChildren;
  LPARAM lParam;
} TVITEM;
```

Поле mask определяет, какие из оставшихся полей несут информацию, и может содержать комбинацию следующих значений:

TVIF_CHILDREN	Поле cChildren содержит данные
TVIF_HANDLE	Поле hItem содержит данные
TVIF_IMAGE	Поле iImage содержит данные
TVIF_PARAM	Поле lParam содержит данные
TVIF_SELECTEDIMAGE	Поле iSelectedImage содержит данные
TVIF_STATE	Обо поля state и stateMask содержат данные
TVIF_TEXT	Поле pszText содержит данные

hItem — содержит дескриптор записи, информация о которой содержится в структуре (разумеется, при добавлении новой записи этот дескриптор не задается, т. е. поле используется для запроса информации о существующей записи).

state и stateMask — задают текущее и возможные состояния для данной записи и могут содержать комбинации следующих флагов (за исключением двух последних):

TVIS_BOLD	Запись отображается полужирным шрифтом
TVIS_CUT	Запись выделена для вырезания или вставки (cut-and-paste)

TVIS_DISABLED	Запись недоступна и для ее перерисовки применяется стандартный стиль и цвет
TVIS_DROPHILITED	Запись выделена для перетаскивания (drag-and-drop)
TVIS_EXPANDED	Список дочерних записей должен быть развернут так, чтобы дочерние записи были видимы; данное значение применимо только для родительской записи
TVIS_EXPANDEDONCE	Список дочерних записей был развернут по крайней мере один раз; для родительской записи, имеющей данный стиль, уведомления TVN_ITEMEXPANDING и TVN_ITEMEXPANDED не посылаются; данное значение применимо только для родительской записи
TVIS_FOCUSED	Запись имеет фокус ввода; естественно, что может быть выделено несколько записей одновременно, но фокус ввода может иметь только одна
TVIS_EXPANDPARTIAL	Запись частично развернута
TVIS_SELECTED	Запись выбрана
TVIS_OVERLAYMASK	Подгружаемое изображение для записи включается, когда запись перерисовывается; индекс подгружаемого изображения должен быть задан в параметре state структуры TVITEM при использовании макроса INDEXTOOVERLAYMASK; подгружаемая запись должна быть добавлена в список изображений дерева с помощью функции CImageList::SetOverlayImage; это значение не может применяться совместно с другими значениями
TVIS_STATEIMAGEMASK	Изображение состояния записи включается, когда запись перерисовывается; индекс состояния записи должен быть задан в параметре state структуры TV_ITEM при использовании макроса INDEXTOSTATEIMAGEMASK; это значение не может применяться совместно с другими значениями

pszText — указатель на строку, содержащую надпись записи, если заданы соответствующие атрибуты в структуре. Если параметр имеет значение LPSTR_TEXTCALLBACK, то это строка обратного вызова (см. разд. "Записи по запросу" выше) и родительское окно само отвечает за хранение текстовой строки. В этом случае просмотр дерева посылает родительскому окну уведомление TVN_GETDISPINFO, когда необходимо вывести на экран, отсортировать или отредактировать надпись, и посылает уведомление TVN_SETDISPINFO, если надпись записи изменена. Если структура должна получать атрибуты записи, то это указатель на буфер, в который она получит надпись. Поле cchTextMax определяет размер буфера, на который указывает параметр pszText, если структура получает атрибуты записи. Если же она задает атрибуты записи, то данный параметр игнорируется. Поля iImage и iSelectedImage задают индексы значков для невыбранного и выбранного состояний. Если значения одного или обоих полей равны I_IMAGECALLBACK, то родительское окно отвечает за хранение изображений. Поле cChildren содержит количество дочерних записей, связанных с данной записью. Если значение данного поля равно I_CHILDRENCALLBACK, то родительское окно отвечает за перерисовку дочерних записей. Это поле заполняется функцией, возвращающей информацию о записи. Поле lParam задает некоторое 32-битное значение, ассоциируемое с записью.

Структура TVITEMEX имеет одно дополнительное поле в дополнение к полям структуры TVITEM — int iIntegral, в которой задается высота записи в приращениях стандартной высоты. Кроме того, для поля mask может быть

установлен еще один флаг — TVIF_INTEGRAL, открывающий доступ к полю iIntegral.

Для удаления ранее вставленной записи служит функция:

BOOL CTreeCtrl::**DeleteItem**(HTREEITEM hItem) —

обратите внимания, что вместо индексов записей используются их дескрипторы. Если параметр hItem равен TVI_ROOT, то будут удалены все записи.

Для изменения или получения всех или части параметров записи предназначены функции SetItem и GetItem.

Каждая запись просмотра дерева имеет текущее состояние, например, она может быть выбрана, недоступна, развернута и т. д. Во многих случаях просмотр дерева автоматически устанавливает состояния записей, отражая действия, производимые пользователем. Для получения или изменения только состояния записи служат функции GetItemState и SetItemState.

Для получения или изменения только изображений, связанных с записью, служат функции GetItemImage и SetItemImage, а для получения или изменения только надписи записи — функции GetItemText и SetItemText.

BOOL CTreeCtrl::**GetItemRect**(HTREEITEM hItem, LPRECT lpRect, BOOL bTextOnly) —

возвращает TRUE, если запись hItem видима, в этом случае размеры прямоугольника содержатся в переменной lpRect. Координаты задаются относительно левого верхнего угла окна просмотра дерева. Если значение bTextOnly равно TRUE, то параметр содержит только размеры надписи для данной записи. Иначе, он содержит размеры прямоугольника, который содержит запись целиком.

BOOL CTreeCtrl::**ItemHasChildren**(HTREEITEM hItem) —

возвращает значение TRUE, если указанная запись имеет дочерние записи.

HTREEITEM CTreeCtrl::**GetChildItem**(HTREEITEM hItem) —

возвращает дескриптор первой дочерней записи

Записи могут быть свернуты или развернуты простым щелчком мыши. Если дочерние записи некоторой записи развернуты, то они выводятся ниже ее. Если они свернуты, то они на экран не выводятся. Переключение из одного состояния в другое происходит автоматически при двойном щелчке мыши на записи или нажатии специальной кнопки, расположенной слева от родительской записи (если запись имеет стиль TVS_HASBUTTONS). Приложение также может разворачивать и сворачивать ветви дерева путем вызова функции Expand:

BOOL CTreeCtrl::**Expand**(HTREEITEM hItem, UINT nCode) —

сворачивает или разворачивает дочерние записи, той записи, что задана параметром hItem, параметр nCode определяет, что нужно сделать с веткой дочерних записей, и может содержать одно из следующих значений:

TVE_COLLAPSE	Сворачивание поддерева записей
TVE_COLLAPSERESET	Сворачивание поддерева с одновременным удалением всех дочерних записей
TVE_EXPAND	Разворачивание поддерева
TVE_TOGGLE	Разворачивание поддерева, если оно свернуто, и сворачивание, если развернуто

Следующая функция только выбирает запись:

HTREEITEM CTreeCtrl::**Select**(HTREEITEM hItem, UINT nCode) —

осуществляет выбор записи с одновременной прокруткой элемента управления так, чтобы запись стала видимой. Параметр nCode определяет действия, которые следует произвести помимо собственно выбора записи. Возможны следующие значения:

TVGN_CARET	Установить выбор заданной записи
TVGN_DROPHILITE	Перерисовать запись как запись назначения перетаскивания (drag-and-drop)
TVGN_FIRSTVISIBLE	Прокрутить записи так, чтобы заданная запись стала видимой

Эти операции также выполняет следующая пара функций:

HTREEITEM CTreeCtrl::**SelectItem**(HTREEITEM hItem) —

выделяет заданную параметром hItem запись просмотра дерева.

HTREEITEM CTreeCtrl::**SelectDropTarget**(HTREEITEM hItem) —

перерисовывает заданную запись как запись назначения операции перетаскивания.

Поиск и сортировка записей дерева

Функции этой группы предназначены для поиска записей по различным критериям, но в отличие от последовательных списков, для просмотра дерева, прежде всего, поиск осуществляется по тому, как искомая запись соотносится с другими.

HTREEITEM CTreeCtrl::**GetNextItem**(HTREEITEM hItem, UINT nCode) —

возвращает дескриптор следующей записи, подходящей по заданным параметрам, или NULL, если таковая не найдена. Параметр hItem определяет дескриптор записи, относительно которой производится поиск. Параметр nCode задает критерии поиска и может содержать один из следующих параметров:

TVGN_CARET	Вернуть текущую выбранную запись
TVGN_CHILD	Вернуть первую дочернюю запись; в этом случае параметр hItem должен быть равен NULL
TVGN_DROPHILITE	Вернуть запись, которая является конечной целью при операции перетаскивания записи просмотра дерева
TVGN_FIRSTVISIBLE	Вернуть первую видимую запись

TVGN_NEXT	Вернуть следующую запись, которая лежит на том же уровне, что и запись, относительно которой производится поиск
TVGN_NEXTVISIBLE	Вернуть видимую запись, следующую за данной
TVGN_PARENT	Вернуть родительскую запись
TVGN_PREVIOUS	Вернуть предыдущую запись, которая лежит на том же уровне, что и запись, относительно которой производится поиск
TVGN_PREVIOUSVISIBLE	Вернуть видимую запись, предшествующую данной записи
TVGN_ROOT	Вернуть первую дочернюю запись корневой записи, поддереву которой принадлежит указанная в параметре hItem запись

Если нужно получить записи, лежащие на том же уровне иерархии, что и данная запись, то можно воспользоваться следующими функциями:

HTREEITEM CTreeCtrl::**GetNextSiblingItem**(HTREEITEM hItem) —

возвращает дескриптор следующей записи.

HTREEITEM CTreeCtrl::**GetPrevSiblingItem**(HTREEITEM hItem) —

возвращает дескриптор предыдущей записи.

Следующая функция возвращает дескриптор родительской записи заданной параметром hItem записи:

HTREEITEM CTreeCtrl::**GetParentItem**(HTREEITEM hItem)

Следующие несколько функций связаны с поиском записей, видимых в окне элемента управления:

BOOL CTreeCtrl::**EnsureVisible**(HTREEITEM hItem) —

гарантирует, что указанная в hItem запись видна. Если нужно, то функция разворачивает необходимую ветку дерева и прокручивает записи так, чтобы указанная была видна.

HTREEITEM CTreeCtrl::**GetFirstVisibleItem**( ) —

возвращает дескриптор первой видимой записи.

HTREEITEM CTreeCtrl::**GetNextVisibleItem**(HTREEITEM hItem) —

возвращает дескриптор следующей и предыдущей видимой записи относительно заданной в параметре hItem.

HTREEITEM CTreeCtrl::**GetPrevVisibleItem**(HTREEITEM hItem) —

возвращает дескриптор предыдущей видимой записи относительно заданной в параметре hItem.

HTREEITEM CTreeCtrl::**GetSelectedItem**( ) —

возвращает дескриптор текущей выбранной записи дерева.

Класс CTreeCtrl позволяет определить запись по координатам, и, если требуется, узнать, на какой записи находится указатель мыши. Для этого следу-

ет воспользоваться функцией HitTest, например, в обработчике нажатия кнопки мыши.

К этой же группе мы отнесли функции, осуществляющие сортировку записей просмотра дерева. Их две. Первая предназначена для сортировки по алфавиту дочерних записей родительской записи. Она определяет уровни дочерних записей, которые должны быть отсортированы:

BOOL CTreeCtrl::**SortChildren**(HTREEITEM hItem) —

сортирует дочерние записи родительской записи, указанной в hItem. Если значение параметра hItem равно NULL, то будет отсортирован весь просмотр дерева.

Если же необходимо задать собственный критерий для сортировки записей, то позволяет это сделать функция SortChildrenCB:

BOOL CTreeCtrl::**SortChildrenCB**(LPTV_SORTCB pSort) —

сортирует дочерние записи, используя определяемую приложением функцию сортировки. Параметр pSort является указателем на структуру TV_SORTCB:

```
typedef struct _TVSORTCB {
  HTREEITEM hParent;
  PFNTVCOMPARE lpfnCompare;
  LPARAM lParam;
} TVSORTCB;
```

Поле lpfnCompare — указатель на определяемую приложением функцию сортировки записей, вызываемую всякий раз, когда необходимо сравнить две записи:

int CALLBACK **CompareFunc**(LPARAM lParam1, LPARAM lParam2, LPARAM lParamSort) —

должна возвращать отрицательное значение, если первая запись должна предшествовать второй, положительное, если первая запись должна следовать за второй, и нуль, если записи равны. Параметры функции сортировки аналогичны параметрам функции сортировки просмотра списка, которая была описана в соответствующем разделе.

Обработка уведомлений

Когда пользователь работает с просмотром дерева, посылаются различные уведомления, для обработки которых можно назначить функции-обработчики:

TVN_BEGINDRAG	Начало перетаскивания (drag-and-drop) с левой кнопкой мыши
TVN_BEGINLABELEDIT	Начало редактирования надписи записи
TVN_BEGINRDRAG	Начало перетаскивания (drag-and-drop) с правой кнопкой мыши

TVN_DELETEITEM	Удаление конкретной записи просмотра дерева
TVN_ENDLABELEDIT	Завершение редактирования надписи записи
TVN_GETDISPINFO	Запрос информации о записи по запросу просмотра дерева для ее отображения
TVN_ITEMEXPANDED	Смена состояния записи просмотра дерева (сворачивание или разворачивание)
TVN_ITEMEXPANDING	Запрос подтверждения смены состояния записи просмотра дерева (сворачивание или разворачивание)
TVN_KEYDOWN	Нажатие клавиши на клавиатуре
TVN_SELCHANGED	Смена выбора записи просмотра дерева
TVN_SELCHANGING	Запрос подтверждения смены выбора записи просмотра дерева
TVN_SETDISPINFO	Уведомляет родительское окно о необходимости изменения информации о записи по запросу, которая предоставляется приложением

Просмотр дерева посылает уведомления в форме сообщения WM_NOTIFY, например:

```
BOOL CMarsWnd::OnNotify(WPARAM wParam, LPARAM lParam, LRESULT* pResult)
{
  ...
  TVDISPINFO *ptvdi = (TV_DISPINFO *)lParam;
  TVITEM item = ptvdi->item;
  ...
}
```

Некоторые уведомления работают с другой структурой (NMTREEVIEW):

```
  NMTREEVIEW *pnmtv = (NMTREEVIEW *)lParam;
```

Механизм обработки уведомлений в принципе тот же, что и для просмотров списка, рассмотренных в предыдущем разделе.

Так же, как и класс CListCtrl для просмотров списка, класс CTreeCtrl имеет в своем составе функции поддержки редактирования надписей записи GetEditControl и EditLabel, функцию CreateDragImage, создающую временный список изображений для реализации механизма перемещения записей просмотра дерева, функции GetBkColor, SetBkColor, GetTextColor, SetTextColor, GetInsertMarkColor и SetInsertMarkColor, работы с цветами фона, текста и вставляемой записи, GetToolTips и SetToolTips для работы со всплывающими подсказками, функцию EnsureVisible, гарантирующую видимость конкретного элемента дерева и некоторые другие, в работе которых вы сможете разобраться без труда.

Класс CMonthCalCtrl

Объект класса CMonthCalCtrl инкапсулирует функциональные возможности элемента управления Windows "календарь" (month calendar). Этот элемент управления позволяет:

- Просматривать различные месяцы
- Отобразить текущий день, щелкнув текст "сегодняшнее число"
- Выбрать во всплывающем меню месяц или год

Кроме того, можно одновременно показывать несколько месяцев одновременно и выделять специальным образом определенные дни, такие как праздники.

Начнем, как обычно, с создания:

BOOL CMonthCalCtrl::Create(DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID)

и

BOOL CMonthCalCtrl::**Create**(DWORD dwStyle, const POINT& pt, CWnd* pParentWnd, UINT nID) —

создают объект Windows календарь и присоединяют его к объекту класса CMonthCalCtrl. Как обычно, параметр dwStyle представляет комбинацию стилей Windows и свойственных этому элементу управления:

MCS_DAYSTATE	Элемент управления будет посылать уведомление для определения того, какие дни отображать полужирным шрифтом
MCS_MULTISELECT	Можно выбирать одновременно несколько дней из диапазона. Величину диапазона можно настраивать (по умолчанию одна неделя)
MCS_NOTODAY	В нижней части элемента управления не будет отображаться "сегодняшнее" число
MCS_NOTODAYCIRCLE	"Сегодняшнее" число не будет обведено
MCS_WEEKNUMBERS	В элементе управления будет отображаться номер недели в году (1–52)

Размер и положение элемента управления задаются параметром rect, либо pt; pParentWnd — указатель на родительское окно (не может быть NULL); nID — идентификатор элемента управления.

После того как элемент управления создан, можно получать от него и передавать ему различные параметры.

BOOL CMonthCalCtrl::**GetMinReqRect**(RECT* pRect) const —

возвращает (через параметр pRect) минимальные размеры, необходимые для отображения одного полного месяца.

BOOL CMonthCalCtrl::**SizeMinReq**(BOOL bRepaint = TRUE) —

устанавливает минимальный размер элемента управления, необходимый для отображения одного месяца. Параметр bRepaint задает необходимость перерисовки (TRUE по умолчанию).

BOOL CMonthCalCtrl::**SetFirstDayOfWeek**(int iDay, int* lpnOld = NULL)

и

int CMonthCalCtrl::**GetFirstDayOfWeek**(BOOL* pbLocal = NULL) —

позволяют установить (iDay) или узнать день, с которого будет начинаться неделя; параметр может принимать следующие значения:

Значение	День недели
0	Понедельник (Monday)
1	Вторник (Tuesday)
2	Среда (Wednesday)
3	Четверг (Thursday)
4	Пятница (Friday)
5	Суббота (Saturday)
6	Воскресенье (Sunday)

В параметре lpnOld возвращается предыдущее значение первого дня недели. Параметр pbLocal позволяет определить, совпадают ли установки календаря с аналогичными системными (если его значение NULL, то совпадают).

Как и для всех элементов управления, для календаря также можно манипулировать цветами:

COLORREF CMonthCalCtrl::**SetColor**(int nRegion, COLORREF ref) —

устанавливает цвет различных областей элемента управления. Значение цвета задается тройкой RGB в параметре ref.

COLORREF CMonthCalCtrl::**GetColor**(int nRegion) const —

возвращает значение цвета соответствующей области элемента управления.

Область, для которой устанавливается, либо возвращается значение цвета, задается параметром nRegion:

MCSC_BACKGROUND	Цвет фона, между месяцами
MCSC_MONTHBK	Цвет фона внутри месяца
MCSC_TEXT	Цвет текста внутри месяца
MCSC_TITLEBK	Цвет фона в заголовке элемента управления
MCSC_TITLETEXT	Цвет текста в заголовке элемента управления

MCSC_TRAILINGTEXT	Цвет текста, используемый для заголовка и "особых" дней, в качестве которых выступают дни предыдущего и следующего месяцев, показываемые в текущем месяце

В классе имеется функция, которая позволяет отображать каждый день месяца индивидуально — полужирным или обычным начертанием:

BOOL CMonthCalCtrl::**SetDayState**(int nMonths, LPMONTHDAYSTATE pStates) —

параметр *nMonth* задает размер массива с информацией о каждом дне, на который указывает параметр pStates. Тип данных LPMONTHDAYSTATE:

typedef DWORD MONTHDAYSTATE, FAR * LPMONTHDAYSTATE;

представляет собой битовые поля, где каждый бит (с 1 по 31) соответствует дню месяца. Если бит установлен, то этот день отображается полужирным шрифтом.

Информацию о "сегодняшнем дне" (today) можно установить или получить с помощью функций

BOOL CMonthCalCtrl::**GetToday**(CTime& refDateTime) const,
BOOL CMonthCalCtrl::**GetToday**(LPSYSTEMTIME pDateTime) const,

и

void CMonthCalCtrl::**SetToday**(const CTime* pDateTime),
void CMonthCalCtrl::**SetToday**(const LPSYSTEMTIME pDateTime)

Назначение и принципы использования оставшихся функций класса, достаточно тривиальны, поэтому мы не будем останавливаться на них подробно, а только перечислим:

GetCurSel	Устанавливает параметры выбранной в текущий момент даты
SetCurSel	Возвращает параметры выбранной в текущий момент даты
SetMaxSelCount	Устанавливает максимальное число дней, которые могут быть выбраны в месяце
GetMaxSelCount	Возвращает максимальное число дней, которые могут быть выбраны в месяце
SetRange	Устанавливает минимальное и максимальное значения даты, доступные в календаре
GetRange	Возвращает минимальное и максимальное значения даты, доступные в календаре
SetSelRange	Для заданного диапазона устанавливает даты в состояние "выбрано"
GetSelRange	Возвращает диапазон дат, которые находятся в состоянии "выбрано"
GetMonthRange	Возвращает информацию о нижнем и верхнем значениях отображаемых месяцев

Обработка уведомлений

При работе пользователя с элементом управления "календарь" посылаются различные уведомления, для обработки которых можно создать функции-обработчики:

MCN_GETDAYSTATE	Требуется информация о днях, которые должны быть отображены полужирным шрифтом
MCN_SELCHANGE	Произошло изменение выбранного дня или диапазона
MCN_SELECT	Пользователь выбрал день

Как создавать функции-обработчики, надеемся, рассказывать заново не надо.

Осталось рассмотреть последний элемент управления, которого, кстати, не было в предыдущих версиях библиотеки MFC.

Класс CDateTimeCtrl

Объект класса CDateTimeCtrl инкапсулирует функциональные возможности сборного элемента управления времени и даты — "дата/время" и предоставляет простой интерфейс для обмена этой информацией с пользователем. Интерфейс содержит поля, каждое из которых отображает свою часть информации, хранящуюся в элементе управления. Пользователь может изменять эту информацию.

Рассмотрим кратко возможности класса.

BOOL CDateTimeCtrl::**Create**(DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID) —

создает элемент управления Windows и присоединяет его к объекту класса. Параметр dwStyle позволяет задать стиль элемента управления. Это может быть комбинация следующих флагов:

DTS_APPCANPARSE	Пользователь может перейти в режим редактирования текста в клиентской области элемента управления, нажав клавишу <F2> на клавиатуре. После окончания пользовательского ввода элемент управления посылает уведомление DTN_USERSTRING, которое владелец может обрабатывать
DTS_LONGDATEFORMAT	Дата отображается в развернутом формате, например, "Четверг, 01 Июля, 1999"
DTS_RIGHTALIGN	Раскрывающийся список элемента управления будет отображать текст справа налево, а не слева направо, как установлено по умолчанию
DTS_SHOWNONE	Если установлен этот стиль, то слева от текущей даты появляется флажок. Когда флажок снят, дата отображается серым цветом, и приложение не может получить информацию из элемента управления

DTS_SHORTDATEFORMAT	Дата отображается в коротком формате, например, "01.07.99"
DTS_TIMEFORMAT	Элемент управления отображает время, а не дату
DTS_UPDOWN	Справа от поля, где отображается дата (или время), располагается счетчик, с помощью которого можно изменять значение даты (или времени). По умолчанию этот флаг снят, и справа от поля располагается кнопка раскрывающегося списка, нажатие которой приводит к появлению элемента управления "календарь"

rect — ссылка на структуру RECT, где определяется положение и размер элемента управления; pParentWnd — указатель на родительское окно (не должен быть равен NULL); параметр nID — уникальный идентификатор элемента управления.

Из оставшихся функций особо остановимся только на одной, которая непосредственно связана именно с этим элементом управления:

BOOL CDateTimeCtrl::**SetFormat**(LPCTSTR pstrFormat) —

устанавливает новое значение формата изображения элемента управления, которое определяется строкой pstrFormat. В ней можно записывать произвольный текст и задавать специальные символы форматирования:

"d"	Для дня используется одна или две цифры
"dd"	Для дня всегда используются две цифры — для чисел от 1 до 9 слева добавляется 0
"ddd"	Отображается сокращенное обозначение дня недели
"dddd"	Отображается полное обозначение дня недели
"h"	Для часов используется одна или две цифры в 12-часовом формате
"hh"	Для часов всегда используются две цифры в 12-часовом формате
"H"	Для часов используется одна или две цифры в 24-часовом формате
"HH"	Для часов всегда используются две цифры в 24-часовом формате
"m"	Для минут используется одна или две цифры
"mm"	Для минут всегда используются две цифры
"M"	Для месяца используется одна или две цифры
"MM"	Для месяца всегда используются две цифры
"MMM"	Для месяца используется трехбуквенная аббревиатура
"MMMM"	Отображается полное название месяца
"y"	Для года используется одна цифра
"yy"	Для года используются две цифры
"yyy"	Полное обозначение года

Помимо рассмотренных символов форматирования можно использовать один или несколько символов "X", которые информируют о том, что в данном поле желательно поместить специальную информацию (непрерывная последовательность символов "X"

трактуется как одно поле). Для этого необходимо определить обработчики уведомляющих сообщений DTN_FORMAT, DTN_WMKEYDOWN и DTN_FORMATQUERY.

Примечание

Раздел справочной системы Visual C++ 6 — Date and Time Picker Controls на удивление хорошо и точно описывает все нюансы рассматриваемых вопросов форматирования. Поэтому мы не будем останавливаться на этом более подробно.

Остальные функции "работают" собственно с элементом управления "календарь", который ассоциирован с рассматриваемым элементом управления. Указатель на него можно получить с помощью функции:

CMonthCalCtrl* CDateTimeCtrl::**GetMonthCalCtrl**( ) const —

возвращает указатель на объект класса CMonthCalCtrl, связанный с элементом управления даты и времени. Если в момент вызова окно невидимо, функция возвращает NULL.

Остальные функции позволяют манипулировать непосредственно "календарем" и вам уже практически знакомы.

COLORREF CDateTimeCtrl::**GetMonthCalColor**(int iColor) const —

позволяет узнать цвет части календаря, которая задается параметром iColor.

COLORREF CDateTimeCtrl::**SetMonthCalColor**(int iColor, COLORREF ref) —

устанавливает значение цвета (параметр ref) для определенной части календаря, которая задается параметром iColor и может принимать одно из следующих значений:

MCSC_BACKGROUND	Цвет фона между месяцами
MCSC_MONTHBK	Цвет фона внутри месяца
MCSC_TEXT	Цвет текста внутри месяца
MCSC_TITLEBK	Цвет фона в заголовке календаря
MCSC_TITLETEXT	Цвет текста в заголовке календаря
MCSC_TRAILINGTEXT	Цвет, используемый для заголовка и "пустых дней" — дней между месяцами

void CDateTimeCtrl::**SetMonthCalFont**(
HFONT hFont, BOOL bRedraw = TRUE) —

устанавливает новый шрифт элемента управления, который задается дескриптором hFont. Параметр bRedraw определяет, когда требуется перерисовать элемент управления, чтобы применить новый шрифт: значение по умолчанию (TRUE) возлагает эту задачу на сам элемент управления, а FALSE говорит о необходимости немедленной перерисовки.

Установить и получить системное время позволяют функции:

BOOL CDateTimeCtrl::**SetTime**(const CTime* pTimeNew),
BOOL CDateTimeCtrl::**SetTime**(LPSYSTEMTIME pTimeNew = NULL)

DWORD CDateTimeCtrl::**GetTime**(CTime& timeDest) const,
DWORD CDateTimeCtrl::**GetTime**(LPSYSTEMTIME pTimeDest) const —

в качестве параметра может использоваться указатель как на объект класса CTime, так и на структуру SYSTIME. Возвращаемое двойное слово показывает состояние элемента управления. Если возвращается GDT_NONE, то у элемента управления устанавливается стиль DTS_SHOWNONE. При возврате же GDT_VALID системное время сохраняется в месте назначения.

Обработка уведомлений

Когда пользователь работает с элементом управления "дата/время", посылаются различные уведомления, для обработки которых можно назначить функции-обработчики:

DTN_CLOSEUP	Удален раскрывающийся элемент управления "календарь"
DTN_DATECHANGE	Произошли изменения в элементе управления
DTN_DROPDOWN	Отображается раскрывающийся элемент управления "календарь"
DTN_FORMAT	Часть текста форматирования представлена полями обратного вызова
DTN_FORMATQUERY	Информация о максимальном размере текста, отображаемого в полях обратного вызова
DTN_USERSTRING	Пользователь закончил работу с элементом управления (посылается только в случае установленного стиля DTS_APPCANPARSE)
DTN_WMKEYDOWN	Пользователь нажал клавишу клавиатуры, находясь в поле обратного вызова элемента управления

Механизм обработки уведомлений в принципе не отличается от рассмотренных ранее.

Глава 27

Расширенный текстовый редактор

Расширенный текстовый редактор представляет собой окно, в котором пользователь может вводить и редактировать текст. В отличие от текстового поля, расширенный редактор поддерживает как изменение атрибутов отдельных символов и слов, так и форматирование абзацев. Кроме того, он позволяет работать с объектами OLE.

Редактор предоставляет программный интерфейс для форматирования текста. Поэтому само приложение должно включать необходимые компоненты интерфейса с пользователем, чтобы ему были доступны эти возможности. Редактор поддерживает изменение атрибутов символов, абзацев и просто выделенного текста. Примерами различных атрибутов текста являются полужирное и курсивное начертание, подчеркнутый текст и т. п. Примером атрибутов абзаца являются выравнивание (по левому краю, по центру и т. п.), отступы, размещение различных позиций табуляции. Однако для доступа к этим возможностям приложение должно иметь соответствующие панель инструментов, пункты меню, диалоги форматирования. Также необходимы функции для запроса у редактора текущих атрибутов выделенного текста и вывода их в диалоговом окне для пользователя.

Расширенный редактор поддерживает все операции и уведомляющие сообщения, используемые обычным многострочным текстовым полем. Поэтому приложение, уже использующее текстовое поле, может быть легко модифицировано для использования расширенного редактора. Дополнительные и уведомляющие сообщения дают приложению возможность использовать новые возможности, предоставляемые расширенным редактором.

Класс CRichEditCtrl

Расширенный редактор, как элемент управления Windows, представлен в библиотеке MFC классом CRichEditCtrl.

Создать редактор можно двумя способами. Первый заключается в создании окна соответствующего оконного класса Windows или с использованием

класса CRichEditCtrl библиотеки MFC. Второй способ заключается в использовании архитектуры "документ/представление" для создания расширенного редактора. В этом случае будут использованы следующие классы: CRichEditDoc (документ), CRichEditView (представление), а также классы CRichEditCtrl и CRichEditCntrItem.

Кроме рассмотренных способов, начиная с пятой версии Visual C++, появилась возможность добавлять расширенный редактор в блок диалога в качестве элемента.

Независимо от того, создается объект класса CRichEditCtrl разработчиком или в рамках некоторой функции библиотеки MFC, основная работа с содержимым редактора осуществляется при помощи функций-членов именно этого класса. Чтобы лучше разобраться с тем, как же все-таки создается объект данного класса, рассмотрим соответствующие функции.

Создание объекта класса CRichEditCtrl

Конструктор без параметров:

CRichEditCtrl::**CRichEditCtrl**( ) —

создает объект класса CRichEditCtrl.

Для создания собственно расширенного редактора Windows и обеспечения связи его с объектом класса необходимо вызвать функцию Create:

BOOL CRichEditCtrl::**Create**(DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID) —

параметр dwStyle задает стили редактора. Допустимыми являются любые стили, применимые для текстового поля. Кроме того, можно использовать новые стили:

ES_SAVESEL	Позволяет сохранять отображение текущей выделенной части содержимого редактора даже в том случае, если окно редактора теряет фокус ввода
ES_SUNKEN	Окно редактора имеет "утопленный" вид. Это достигается путем создания соответствующей рамки вокруг окна, когда левая и верхняя грань отрисовываются более темным цветом, чем правая и нижняя
ES_DISABLENOSCROLL	Позволяет отображать полосы прокрутки даже в том случае, если в них нет необходимости
ES_SELECTIONBAR	Аналогичен стилю WS_MAXIMIZE

Параметр rect задает размер и расположение окна редактора. Может быть объектом класса CRect или структурой RECT. Параметр pParentWnd задает родительское окно. Значение этого параметра не должно быть равно NULL. Параметр nID задает идентификатор редактора.

Во время выполнения функции Create Windows посылает редактору сообщения WM_NCCREATE, WM_NCCALCSIZE, WM_CREATE и WM_GETMINMAXINFO. По умолчанию они обрабатываются функциями

OnNcCreate, OnNcCalcSize, OnCreate и OnGetMinMaxInfo базового класса CWnd. Для введения дополнительной обработки этих сообщений следует создать свой, производный от CRichEditCtrl, класс и переопределить функции-обработчики перечисленных сообщений.

Изменение параметров редактора

Кроме возможности задать стили редактора при его создании, при помощи функций SetOptions и SetReadOnly можно впоследствии, т. е. после создания, изменить некоторые из заданных стилей или добавить новые.

void CRichEditCtrl::**SetOptions**(WORD wOp, DWORD dwFlags) —

позволяет установить или изменить ряд стилей объекта класса CRichEditCtrl. Параметр wOp определяет тип выполняемой операции и может принимать одно из следующих значений:

ECOOP_SET	Установить стили, заданные в dwFlags, с отменой текущих
ECOOP_OR	К текущим стилям должны быть добавлены стили, заданные в dwFlags
ECOOP_AND	Должны быть сохранены только те из текущих стилей, которые совпадают со стилями, установленными в dwFlags
ECOOP_XOR	Должны быть сохранены только те из текущих стилей, которые не встречаются среди стилей, заданных в dwFlags

Значения стилей, комбинация которых может содержаться в параметре dwFlags:

ECO_AUTOWORDSELECTION	Предписывает автоматически выделять слово в редакторе, если на нем сделан двойной щелчок мышью
ECO_AUTOHSCROLL	Редактор автоматически прокручивает текст вправо на 10 символов, если пользователь вводит символ в конце строки. Если пользователь нажимает клавишу <Enter>, то происходит переход к началу следующей строки
ECO_AUTOVSCROLL	Автоматически прокручивает текст на одну строку вверх, если пользователь нажал клавишу <Enter> в последней строке
ECO_NOHIDESEL	Отменяет снятие выделения с текста (по умолчанию). При потере редактором фокуса ввода выделение должно быть снято с текста, выделенного в данный момент в редакторе
ECO_READONLY	Устанавливает стиль "только для чтения" для содержимого редактора, исключая возможность пользовательского ввода и редактирования содержимого редактора
ECO_WANTRETURN	Приводит к тому, что в многострочный редактор при нажатии пользователем клавиши <Enter> вставляется символ перевода строки. Если данный флаг не установлен, то при нажатии клавиши <Enter> родительскому окну редактора посылается сообщение, имитирующее нажатие кнопки, заданной по умолчанию (например, кнопки **OK** в блоке диалога). Установка данного флага для однострочного редактора ни на что не влияет

ECO_SAVESEL	Приводит к сохранению выделенного участка содержимого редактора, когда тот теряет фокус. По умолчанию все содержимое редактора выделяется, когда он вновь получает фокус ввода
ECO_VERTICAL	Выводит текст и объекты в вертикальном направлении. Эта возможность доступна только для дальневосточной (Far east) версии Windows

Если нужно установить или снять флаг ECO_READONLY — ("только для чтения"), то можно воспользоваться как предыдущей функцией, позволяющей изменять большинство флагов редактора, так и следующей специализированной функцией:

BOOL CRichEditCtrl::**SetReadOnly**(BOOL bReadOnly = TRUE) —

позволяет установить или снять указанный флаг в зависимости от того, с каким значением параметра bReadOnly она вызвана — значение TRUE устанавливает режим "только для чтения", а значение FALSE — снимает. Возвращает TRUE в случае успешного завершения и FALSE — в противном случае.

Редактор предоставляет возможность изменения цвета фона окна редактора, позволяя задать цвет, отличный от заданного системного цвета. Для изменения цвета символов текста, естественно, никакой аналогичной функции не предусмотрено, т. к. цвет может задаваться вообще для каждого символа в отдельности: цвет символов определяется параметрами шрифта, при помощи которого и осуществляется весь символьный вывод.

COLORREF CRichEditCtrl::**SetBackgroundColor**(BOOL bSysColor, COLORREF cr) —

устанавливает новое значение цвета фона и возвращает ранее установленное. Параметр bSysColor указывает, что для установки цвета фона должно быть взято значение, установленное в системе. Если данный параметр имеет значение TRUE, то следующий параметр, в котором задается новое значение цвета, игнорируется.

Следующие две функции работают не столько с параметрами редактора, сколько с состоянием его содержимого, позволяя определить или, наоборот, установить изменение содержимого редактора.

BOOL CRichEditCtrl::**GetModify**( ) —

возвращает значение TRUE, если действительно были произведены изменения, и FALSE, если содержимое редактора не менялось. При своей работе использует флаг модификации, который поддерживает внутри себя редактор и который существует для определения того, сохранялось ли измененное содержимое редактора после предыдущего сохранения или нет. Несмотря на то что он в зависимости от ситуации автоматически устанавливается и снимается редактором, его можно устанавливать или снимать вызовом следующей функции:

void CRichEditCtrl::**SetModify**(BOOL bModified = TRUE) —

параметр bModified позволяет задать, был ли текст изменен (TRUE) или нет (FALSE).

Пример следующей функции показывает использование функций создания и настройки объекта класса CRichEditCtrl:

```
int CRichEditFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
  if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
    return -1;
  CRect rect;
  GetClientRect(&rect);
  // Создаем объект "редактор", в дальнейшем размеры
  // его будут меняться в зависимости от требований
  // уведомления EN_REQUESTRESIZE
  m_REditCtrl.Create(WS_CHILD | WS_VISIBLE |
    WS_VSCROLL | ES_MULTILINE | ES_NOHIDESEL |
    ES_SAVESEL | ES_SUNKEN, rect, this, 1000);
  // После создания объекта "редактор" добавляем еще два параметра
  m_REditCtrl.SetOptions(ECOOP_OR,
    ECO_AUTOVSCROLL | ECO_AUTOHSCROLL);
  // Меняем цвет редактора и задаем маску уведомлений
  m_REditCtrl.SetBackgroundColor(FALSE, RGB(224, 224, 224));
  m_REditCtrl.SetEventMask(ENM_REQUESTRESIZE);
  m_REditCtrl.SetFocus();
  return 0;
}
```

Настройка окна редактора

Вполне может оказаться необходимым изменить область форматного вывода так, чтобы она отличалась от клиентской области окна, что позволит, например, добавить поля к тексту, если такая область будет меньше клиентской. Для получения и установки прямоугольника форматирования редактора предназначены следующие функции:

void CRichEditCtrl::**GetRect**(LPRECT lpRect)

и

void CRichEditCtrl::**SetRect**(LPCRECT lpRect) —

первая возвращает текущий установленный прямоугольник форматирования, а вторая — устанавливает его. Параметр lpRect является объектом типа CRect или указателем на структуру типа RECT, куда первой функцией помещается информация о текущем прямоугольнике форматирования, или где должен быть первоначально задан новый прямоугольник форматирования для второй функции. *Прямоугольник форматирования* редактора — это прямоугольник, в пределах которого размещается и форматируется текст, содержащийся в редакторе. При создании объекта CRichEditCtrl прямоугольник форматирования устанавливается совпадающим с клиентской областью, в дальнейшем при необходимости с помощью данных функций его можно изменить.

Приложение может изменять размеры редактора так, что они всегда будут соответствовать размеру (по высоте) содержащегося в нем текста. Редактор поддерживает такую возможность путем посылки родительскому окну уведомляющего сообщения EN_REQUESTRESIZE тогда, когда меняется размер его содержимого. При получении данного сообщения приложение должно изменить размеры редактора в соответствии с размерами, указанными в структуре REQRESIZE, для чего можно воспользоваться функцией CWnd::SetWindowPos или CWnd::MoveWindow (что гораздо проще).

Кроме автоматической генерации сообщения о необходимости изменить размер окна, существует функция, позволяющая принудительно послать это сообщение, предварительно вычислив нужный размер:

void CRichEditCtrl::**RequestResize**( ) —

> принуждает объект "редактор" послать своему родительскому окну уведомляющее сообщение о необходимости изменить размер. Для того чтобы получать такие сообщения, необходимо сделать доступной данную возможность, вызвав функцию CRichEditCtrl::SetEventMask со значением параметра равным ENM_REQUESTRESIZE. Польза данной возможности вызывает сомнение, особенно если учесть, что размер окна редактора может благополучно превысить размер родительского окна, так что часть содержимого редактора может стать совершенно недоступной, т. к. в этом случае даже полоса прокрутки самого редактора здесь не поможет.

Далее приведем пример функции обработки уведомляющего сообщения, позволяющей фактически изменять окно редактора.

```
BOOL CRichEditFrame::OnNotify(
  WPARAM wParam, LPARAM lParam, LRESULT* pResult)
{
  NMHDR *nmhdr = (NMHDR *)lParam;
  // Проверим только сообщения от интересующего
  // окна — окна редактора
  if (nmhdr->hwndFrom == m_REditCtrl.m_hWnd)  {
    // Обрабатываем сообщение EN_REQUESTRESIZE
    if (nmhdr->code == EN_REQUESTRESIZE)    {
      REQRESIZE *lpResize = (REQRESIZE *)lParam;
      m_REditCtrl.MoveWindow(&lpResize->rc);
    }
  }
  return CFrameWnd::OnNotify(wParam, lParam, pResult);
}
```

Размер содержимого редактора

Данная группа функций предназначена для настройки максимального размера буфера редактора. Пользуясь этими функциями следует помнить, что внедренные объекты OLE, не являющиеся, естественно, текстом, считаются

равными одному символу каждый. Символ "возврат каретки/перевод строки" равен, как всегда, двум символам.

long CRichEditCtrl::**GetTextLength**( ) —

позволяет получить длину всего текста, содержащегося в редакторе.

Для получения длины буфера служит следующая функция:

long CRichEditCtrl::**GetLimitText**( ) —

позволяет определить, какого размера текст пользователь может ввести в редактор, — она возвращает количество символов (байтов), которое в состоянии принять редактор.

Функция, позволяющая изменить это ограничение:

void CRichEditCtrl::**LimitText**(long nChars = 0) —

ограничивает длину текста, который пользователь может ввести в окне редактора. Параметр nChars — задает в байтах длину текста, который может ввести пользователь. Если значение параметра нуль, то длина текста устанавливается по умолчанию в UINT_MAX.

Функция LimitText задает длину текста, который пользователь может ввести, но она не влияет на длину теста, уже находящегося в редакторе. Также она не влияет на текст, помещаемый в редактор с помощью функции CWnd::SetWindowText. Если приложение использует эту функцию, чтобы поместить в редактор большее количество текста, чем задано функцией CRichEditCtrl::LimitText, пользователь может удалить какой-либо текст, уже находящийся в редакторе. Однако ограничение длины текста предотвратит замену существующего текста новым.

Работа со строками

int CRichEditCtrl::**GetLineCount**( ) —

возвращает число строк в данном объекте CRichEditCtrl.

int CRichEditCtrl::**GetLine**(int nIndex, LPTSTR lpszBuffer)

и

int CRichEditCtrl::**GetLine**(int nIndex, LPTSTR lpszBuffer, int nMaxLength) —

копируют в буфер строку текста из объекта CRichEditCtrl. Скопированная строка не содержит символ конца строки. Функция возвращает число символов, скопированное в буфер. Параметр nIndex указывает индекс строки, которая должна быть скопирована. Параметр lpszBuffer указывает на буфер, в который будет скопирована строка. Если используется первый вариант функции, то в первом слове буфера должно быть занесено значение, равное числу символов, которое буфер в состоянии принять. Параметр nMaxLength задает максимальное количество символов, которые могут быть скопированы в буфер. Первый вариант функции CRichEditCtrl::GetLine использует для получения этого значения первое слово буфера.

int CRichEditCtrl::**GetFirstVisibleLine**( ) —

возвращает индекс первой видимой строки в расширенном редакторе.

int CRichEditCtrl::**LineIndex**(int nLine = −1) —

возвращает символьный индекс для заданной строки. *Символьный индекс* — это число символов от начала буфера редактора до заданной строки. Параметр nLine задает индекс (номер) требуемой строки. Если его значение –1, то возвращается символьный индекс для строки, содержащей каретку, т. е. строки текущей. Если заданный номер строки больше числа строк, содержащихся в редакторе, то возвращается значение –1.

long CRichEditCtrl::**LineFromChar**(long nIndex) —

возвращает индекс строки в редакторе по заданному в nIndex символьному индексу. Если в nIndex задано значение –1, то возвращается индекс строки, содержащей первый символ выделенного текста. Если нет выделенного текста, то возвращается индекс строки, содержащей каретку. Объект OLE считается при этом за один символ.

int CRichEditCtrl::**LineLength**(int nLine = −1)

Функция позволяет получить длину заданной строки в редакторе. Если функция вызывается для многострочного редактора, то возвращается длина заданной строки. Если же она вызывается для однострочного редактора, то возвращается длина всего текста, содержащегося в редакторе. Параметр nLine задает символьный индекс для символа, содержащегося в строке, длину которой необходимо определить. Если его значение –1, то возвращается длина текущей строки, не включая выделенный текст в пределах строки. Если функция вызывается для однострочного редактора, то данный параметр игнорируется. Символьный индекс можно получить по индексу строки с помощью функции LineIndex.

void CRichEditCtrl::**LineScroll**(int nLines, int nChars = 0) —

прокручивает текст в окне многострочного редактора. Параметр nLines задает число строк, на которое необходимо прокрутить текст по вертикали. Параметр nChars задает число позиций символов для прокрутки по горизонтали. Этот параметр игнорируется, если редактор имеет стиль ES_RIGHT или ES_CENTER.

Если размер текущей строки плюс значение, заданное в nLines, превышает общее число строк в редакторе, то последняя строка прокручивается в верхнюю часть окна редактора. Функцию можно использовать для горизонтальной прокрутки за последний символ любой строки.

Примечание

Иногда под *строкой* понимают последовательность символов, оканчивающихся символом конца строки. В данном случае в нашем распоряжении нет такого символа. Мы имеем возможность отличать только абзацы. При этом в данном случае под строкой понимается последовательность символов, расположенных на одной линии. Поэтому может оказаться, что последовательность символов разбивается на несколько строк в результате того, что все символы не поместились на одной, несмотря на отсутствие символов конца абзаца.

Естественно, в этом случае функция GetLineCount вернет значение больше единицы. Тонкость здесь заключается в том, что в результате увеличения ширины окна редактора вся эта последовательность может оказаться на одной строке. Из этого следует, что основной "координатой" символа является его символьный индекс, т. е. номер от начала всего текста, а не номер строки, где он теперь находится, и его смещение в ней.

Что касается выравнивания абзацев, то оно влияет на все строки, находящиеся в нем.

Последняя функция данной группы относится не к строкам, а к отдельным символам или объектам. Но отводить для нее отдельную группу вряд ли имеет смысл.

CPoint CRichEditCtrl::**GetCharPos**(long lChar) —

позволяет определить координаты левого верхнего угла символа или объекта, символьный индекс которого задан параметром lChar. Если значение lChar превышает индекс последнего символа, содержащегося в редакторе, то возвращаются координаты того места, куда следующий символ, с соответствующими установками шрифтов и выравнивания абзацев, может быть вставлен.

Выделение содержимого редактора

Редактор позволяет выбирать символы и OLE-объекты, используя клавиатуру или мышь. Приложение может получить информацию о текущем выделении в редакторе, узнать, что оно (выделение) произошло, а также сделать его видимым или нет.

Для того чтобы узнать текущее выделение текста в расширенном редакторе, можно воспользоваться функцией GetSel:

void CRichEditCtrl::**GetSel**(CHARRANGE & cr)

или

void CRichEditCtrl::**GetSel**(long& nStartChar, long& nEndChar) —

возвращает начальную и конечную позиции выделения; cr — ссылка на структуру CHARRANGE, получающую границы текущего выделения.

Кроме того, что редактор позволяет узнать, какой участок выделен, он также позволяет и установить такое выделение программно:

void CRichEditCtrl::**SetSel**(long nStartChar, long nEndChar)

или

void CRichEditCtrl::**SetSel**(CHARRANGE & cr) —

устанавливает выделение участка текста в редакторе. Параметры ее вызова аналогичны параметрам функции CRichEditCtrl::GetSel. Отметим, что редактор при вызове данной функции не прокручивает текст в окне с тем, чтобы выделенный участок текста был виден.

Для выделения всего текста в редакторе следует вызвать функцию с начальным индексом, равным 0 и конечным индексом, равным –1.

long CRichEditCtrl::**GetSelText**(LPTSTR lpBuf)

и

CString CRichEditCtrl::**GetSelText**( ) —

позволяет получить текст, выделенный в редакторе. Первый вариант функции копирует выделенный текст в буфер, на который указывает параметр lpBuf, и возвращает число скопированных символов. Скопированная в буфер строка не содержит символ конца абзаца. При использовании данной функции нужно быть уверенным, что буфер имеет дос-

таточный размер для получения строки (для определения количества символов в выделенном тексте можно воспользоваться функцией GetSel).

Вторая версия функции возвращает объект класса CString, содержащий выделенный текст, и соответственно не требует выяснения его длины.

Редактор, кроме того что позволяет получить текущее выделение, также позволяет определить, что содержится в нем. Для этого предназначена функция:

WORD CRichEditCtrl::**GetSelectionType**( ) —

возвращаемое значение может содержать комбинацию следующих флагов:

SEL_EMPTY	В редакторе ничего не выделено
SEL_TEXT	В выделении содержится только текст
SEL_OBJECT	В выделенном участке содержится по меньшей мере один объект OLE
SEL_MULTICHAR	В выделенном участке текста содержится более одного символа
SEL_MULTIOBJECT	В выделенном участке содержится более одного элемента OLE

Когда необходимо выполнить поиск фрагмента текста в редакторе, можно воспользоваться функцией:

long CRichEditCtrl::**FindText**(DWORD dwFlags, FINDTEXTEX* pFindText) —

критерии поиска задаются параметром dwFlags, который может быть нулем или комбинацией следующих значений:

FT_MATCHCASE	Осуществлять поиск с учетом регистра
FT_WHOLEWORD	Осуществлять поиск слова целиком

Параметр pFindText — указатель на структуру типа FINDTEXTEX, которая передает функции параметры для поиска и получает результаты. Сама функция возвращает символьную позицию текущего совпадения или значение –1, если больше нет совпадений.

Для удаления выделенного текста служит функция:

void CRichEditCtrl::**Clear**( ) —

действие этой функции, а также большинство других, можно отменить, вызвав функцию:

BOOL CRichEditCtrl::**Undo**( ) —

позволяет отменить последнюю выполненную операцию редактирования. Действие данной функции можно также отменить повторным вызовом функции Undo. Так, например, если был (ошибочно) удален текст в редакторе, то, вызвав функцию Undo, удаление текста можно отменить — текст будет восстановлен, если же вы опять решили его удалить, то достаточно еще раз вызвать Undo и текст вновь будет удален. Таким образом, редактор поддерживает только однократную отмену произведенных действий.

Следующая функция сбрасывает флаг отмены последней операции:

void CRichEditCtrl::**EmptyUndoBuffer**( ) —

после ее выполнения уже нельзя отменить последнюю выполненную операцию.

void CRichEditCtrl::**ReplaceSel**(LPCTSTR lpszNewText) —

заменяет выделенный в редакторе текст заданной строкой текста. Параметр lpszNewText — указатель на текстовую строку, содержащую нуль-символ. Для замены всего текста в редакторе следует воспользоваться функцией CWnd::SetWindowText. Если в редакторе нет выделенного текста, то строка вставляется с позиции, в которой находится каретка.

Это важный факт, потому что программная вставка в произвольное место может быть произведена именно с помощью данной функции. Для перемещения каретки служит функция SetSel, которой в качестве начала и конца области выделения задается одно и то же число — символьный индекс места будущей вставки.

Результат вставки не может быть отменен.

void CRichEditCtrl::**HideSelection**(BOOL bHide, BOOL bPerm) —

показывает или скрывает текущее выделение в редакторе. Параметры: bHide задает, должно ли выделение быть скрыто (значение TRUE) или показано (значение FALSE); bPerm задает, должно ли выделение быть постоянным (обычно при потере редактором фокуса ввода выделение в нем делается невидимым, а при получении — вновь видимым), при вызове функции с данным параметром, равным TRUE, для редактора устанавливается стиль ECO_NOHIDESEL (см. описание функции SetOptions в разделе "Изменение параметров редактора"), приводящий к тому, что при потере редактором фокуса ввода выделение не делается невидимым.

Работа с буфером обмена

Расширенный редактор имеет функции, предназначенные для работы с буфером обмена. К сожалению, для вставки объектов OLE требуется провести большую дополнительную работу, текст же вставляется элементарно, причем с сохранением всех заданных параметров, таких как шрифт, цвет и т. п.

void CRichEditCtrl::**Copy**( ) —

копирует выделенный текст (если он конечно есть) в буфер обмена.

void CRichEditCtrl::**Cut**( ) —

удаляет (вырезает) выделенный текст (если он есть) в редакторе и копирует удаленный текст в буфер обмена. Для восстановления текста, удаленного функцией Cut, можно вызвать функцию Undo.

Для удаления выделенного текста без помещения его в буфер обмена служит функция Clear.

void CRichEditCtrl::**Paste**( ) —

вставляет данные из буфера обмена в редактор с текущей позиции курсора. Данные вставляются лишь в том случае, если буфер обмена содержит данные в формате, который может быть распознан редактором.

void CRichEditCtrl::**PasteSpecial**(UINT nClipFormat, DWORD dvAspect = 0, HMETAFILE hMF = 0) —

вставляет данные из буфера обмена в редактор в заданном формате. Параметр nClipFormat задает формат буфера обмена для вставки в редактор, dvAspect задает тип устройства для данных, которые будут вставлены в редактор, hMF — дескриптор метафайла, содержащего представление значка вставляемого объекта.

BOOL CRichEditCtrl::**CanPaste**(UINT nFormat = 0) —

позволяет узнать, может ли быть содержимое буфера обмена помещено в заданном формате. Параметр nFormat позволяет указать требуемый формат. Если его значение равно 0, то осуществляется проверка для любого формата.

Здесь следует пояснить о форматах буфера обмена. Функция Windows API ::RegisterClipboardFormat позволяет зарегистрировать необходимый формат для буфера обмена, передав при вызове данной функции в параметре lpszFormat текстовую строку, задающую желаемое имя для нового формата:

UINT::**RegisterClipboardFormat**(LPCTSTR lpszFormat) —

при успешном завершении возвращает идентификатор нового зарегистрированного формата — число в диапазоне от 0×C000 до 0×FFFF. Данное значение для требуемого формата и должно использоваться в качестве одного из параметров при вызовах вышеописанных функций в случаях, когда нужно задать необходимый формат буфера обмена.

Расширенный редактор использует два формата: так называемый формат расширенного текста CF_RTF и формат текста и объектов расширенного редактора CF_RETEXTOBJ.

Печать содержимого редактора

Имеется возможность переопределить вывод редактора на заданное устройство, например, на принтер. Для того чтобы задать пределы форматирования текста для устройства вывода, предназначена функция FormatRange:

long CRichEditCtrl::**FormatRange**(FORMATRANGE* pfr,
BOOL bDisplay = TRUE) —

возвращает увеличенный на единицу индекс последнего символа, попавшего в заданную параметром pfr область. Параметр pfr является указателем на структуру типа FORMATRANGE, которая содержит информацию об устройстве вывода. Значение NULL означает, что кэшируемая информация внутри редактора может быть освобождена. Значение TRUE параметра bDisplay задает, что текст должен быть выведен, FALSE — что текст должен быть только измерен.

Вызов функции FormatRange как правило сопровождается вызовом функции DisplayBand:

BOOL CRichEditCtrl::**DisplayBand**(LPRECT pDisplayRect) —

выводит часть содержимого редактора (текст и объекты OLE), которые ранее были отформатированы функцией CRichEditCtrl::FormatRange. Данные текст и объекты OLE обрезаются в соответствии с размерами области, заданной параметром pDisplayRect, который является указателем на объект типа RECT или CRect.

Следующая функция предназначена для задания устройства, для которого редактор осуществляет форматирование текста, и имеет два варианта вызова:

BOOL CRichEditCtrl::**SetTargetDevice**(HDC hDC, long lLineWidth)

и

BOOL CRichEditCtrl::**SetTargetDevice**(CDC& dc, long lLineWidth) —

задает устройство вывода для редактора и толщину линии (параметр lLineWidth) для режима WYSIWYG. В случае успешного завершения функции редактор получает в свое распоряжение контекст устройства, заданный параметрами hDC или dc.

Уведомляющие сообщения

Расширенный редактор поддерживает все уведомляющие сообщения, используемые простым редактором текста, и кроме того, имеет ряд дополнительных. Для того чтобы определить, какие уведомляющие сообщения редактор может посылать родительскому окну, предназначена функция:

long CRichEditCtrl::**GetEventMask**( ) —

позволяет определить маску события (event mask), которая должна быть установлена для редактора. *Маска события* определяет, какие уведомляющие сообщения могут приниматься родительским окном от объекта-редактора.

Соответствующая ей функция SetEventMask позволяет задать эту маску:

DWORD CRichEditCtrl::**SetEventMask**(DWORD dwEventMask) —

маска указывается в параметре dwEventMask.

Приведем уведомляющие сообщения, характерные как для обычного, так и для расширенного редактора:

EN_CHANGE	Пользователь произвел действие, которое, возможно, изменило текст в редакторе. В отличие от сообщения EN_UPDATE, это сообщение посылается после того, как система Windows уже обновила окно редактора
EN_ERRSPACE	Элемент управления не может получить достаточно памяти для размещения текста
EN_HSCROLL	Пользователь сделал щелчок на горизонтальной полосе прокрутки редактора. Родительское окно уведомляется до того, как будет произведено обновление экрана
EN_KILLFOCUS	Редактор теряет фокус ввода
EN_MAXTEXT	Текущая вставка превышает заданное число символов для редактора и поэтому она усечена. Сообщение посылается, если редактор не имеет стиля ES_AUTOHSCROLL, и число символов для вставки превышает ширину редактора. Также посылается, если редактор не имеет стиля ES_AUTOVSCROLL, и общее число строк вместе с текстом для вставки превышает высоту окна редактора
EN_SETFOCUS	Посылается, когда редактор получает фокус ввода

EN_UPDATE	Посылается редактором текста, когда он собирается отобразить измененный текст на экране, после того как элемент управления отформатировал текст, но еще не вывел изменения на экране (с тем, чтобы можно было, в случае необходимости, изменить размеры окна)
EN_VSCROLL	Пользователь щелкнул на вертикальной полосе прокрутки. Родительское окно уведомляется до того, как экран будет перерисован

Следующие уведомляющие сообщения характерны только для расширенного редактора:

EN_MSGFILTER	Обработка данного уведомляющего сообщения позволяет редактору или его родительскому окну отфильтровывать весь ввод, поступающий от клавиатуры и мыши. Обработчик данного сообщения может изменять сообщения, поступающие от клавиатуры или мыши путем изменения содержимого структуры MSGFILTER
EN_PROTECTED	Обработка данного сообщения позволяет определить момент, когда пользователь пытается редактировать текст, предназначенный только для чтения
EN_DROPFILES	Обработкой данного сообщения пользователю можно обеспечить возможность перетаскивания файлов в область расширенного редактора для их редактирования. Задаваемая структура ENDROPFILES содержит информацию о перетаскиваемых файлах
EN_SELCHANGE	Обрабатывая данное сообщение, приложение может обеспечить себе определение момента, когда произведено изменение текущего выделения в редакторе. Данное уведомляющее сообщение указывает на структуру типа SELCHANGE, которая содержит информацию о новом выделении
EN_CORRECTTEXT	Уведомление только для версий Windows, поддерживающих ввод информации с помощью светового пера
EN_IMECHANGE	Уведомление только для азиатских версий Windows
EN_OLEOPFAILED	Уведомляет о возникновении ошибки при работе с объектом OLE
EN_REQUESTRESIZE	Уведомление о том, что размер содержимого редактора стал больше или меньше, чем размер текущего окна редактора
EN_SAVECLIPBOARD	Уведомляет о том, что при уничтожении редактора в буфере обмена осталась информация, которую туда поместил редактор
EN_STOPNOUNDO	Уведомляет о невозможности сохранить информацию для реализации в дальнейшем операции отмены последнего произведенного действия

А теперь приведем значения масок, с помощью которых можно уменьшить поток уведомляющих сообщений от редактора к родительскому окну:

ENM_NONE	Запрещает посылку всех уведомляющих сообщений (значение 0)
ENM_CHANGE	Разрешает посылку сообщения EN_CHANGE
ENM_UPDATE	Разрешает посылку сообщения EN_UPDATE
ENM_SCROLL	Разрешает посылку сообщений, связанных с прокруткой содержимого редактора
ENM_KEYEVENTS	Разрешает посылку сообщений, связанных с обработкой нажатия клавиши на клавиатуре
ENM_MOUSEEVENTS	Разрешает посылку сообщений, связанных с действиями мышью
ENM_REQUESTRESIZE	Разрешает посылку сообщения EN_REQUESTRESIZE
ENM_SELCHANGE	Разрешает посылку сообщения EN_SELCHANGE
ENM_DROPFILES	Разрешает посылку сообщения EN_DROPFILES
ENM_PROTECTED	Разрешает посылку сообщения EN_PROTECTED
ENM_CORRECTTEXT	Разрешает посылку сообщения EN_CORRECTTEXT
ENM_IMECHANGE	Разрешает посылку сообщения EN_IMECHANGE

Расширенные возможности

Большая часть того, что мы уже рассмотрели, не свидетельствует о большом отличии расширенного редактора от обычного текстового поля. Действительно, без возможности устанавливать отличные от принятых по умолчанию параметры форматирования достоинства расширенного редактора несколько сомнительны.

Форматирование символов

Как уже говорилось, одним из достоинств расширенного редактора является возможность форматирования символов. Имеются функции как для собственно форматирования, так и для получения информации о форматировании. У символов можно менять шрифт, размер, цвет, полужирное и наклонное начертания, устанавливать или снимать атрибут "только для чтения".

Для форматирования символов предназначена следующая пара функций:

DWORD CRichEditCtrl::**GetDefaultCharFormat**(CHARFORMAT & cf) —

позволяет получить текущее, установленное по умолчанию, форматирование символов и возвращает параметр dwMask структуры CHARFORMAT. Параметр cf — указатель на структуру типа CHARFORMAT, получающую информацию об атрибутах форматирования и определенную следующим образом:

```
typedef struct _charformat {
  UINT cbSize;
  _WPAD _wPad1;
  DWORD dwMask;
  DWORD dwEffects;
  LONG yHeight;
  LONG yOffset;
  COLORREF crTextColor;
  BYTE bCharSet;
  BYTE bPitchAndFamily;
  TCHAR szFaceName[LF_FACESIZE];
  _WPAD _wPad2;
 } CHARFORMAT;
```

Поле cbSize содержит размер структуры в байтах. Поле dwMask содержит информацию о том, какие из полей структуры содержат действительную информацию, и может быть комбинацией следующих значений:

CFM_BOLD	Значение CFE_BOLD поля dwEffects является действительным
CFM_COLOR	Поле crTextColor и значение CFE_AUTOCOLOR поля dwEffects являются действительными
CFM_FACE	Поле szFaceName содержит значение
CFM_ITALIC	Значение CFE_ITALIC поля dwEffects является действительным
CFM_OFFSET	Поле yOffset содержит значение
CFM_PROTECTED	Значение CFE_PROTECTED поля dwEffects является действительным
CFM_SIZE	Поле yHeight содержит значение
CFM_STRIKEOUT	Значение CFE_STRIKEOUT поля dwEffects является действительным
CFM_UNDERLINE	Значение CFE_UNDERLINE поля dwEffects является действительным

Поле dwEffects содержит эффекты форматирования символов и может быть комбинацией следующих стилей:

CFE_AUTOCOLOR	Цвет текста совпадает со значением, полученным функцией ::GetSysColor со значением единственного параметра равным COLOR_WINDOWTEXT
CFE_BOLD	Символы имеют полужирное начертание
CFE_ITALIC	Символы выводятся курсивом
CFE_STRIKEOUT	Символы зачеркнуты
CFE_UNDERLINE	Символы подчеркнуты
CFE_PROTECTED	Символы защищены от изменения ("только для чтения")

Поле yHeight содержит высоту символов. Поле yOffset — смещение символов от базовой линии: если значение параметра положительно, то символы находятся над базовой линией, если отрицательно, то под ней. Поле crTextColor задает цвет символа. Если задано значение CFE_AUTOCOLOR, то данный параметр игнорируется. Поле bCharSet задает набор символов. Может содержать одно из значений, задаваемых для поля lfCharSet структуры LOGFONT. Поле bPitchAndFamily содержит флаги стиля шрифта, аналогично полю lfPitchAndFamily структуры LOGFONT. Поле szFaceName содержит строку, являющуюся названием шрифта.

Соответствующая предыдущей, функция SetDefaultCharFormat позволяет изменить атрибуты форматирования символа, заданные по умолчанию:

BOOL CRichEditCtrl::**SetDefaultCharFormat**(CHARFORMAT & cf) —

задает структуре типа CHARFORMAT, на которую должен указывать параметр cf, необходимые новые атрибуты форматирования, при этом изменяются только те атрибуты, которые указаны в поле dwMask структуры.

Для работы с форматированием не отдельных символов, а группы выделенных символов, предназначена следующая пара функций:

DWORD CRichEditCtrl::**GetSelectionCharFormat**(CHARFORMAT & cf) —

позволяет получить атрибуты форматирования текущего выделенного в редакторе текста. Параметры функции и тип возвращаемого значения целиком совпадают с описанными параметрами предыдущей функции с той лишь разницей, что структура, на которую указывает параметр cf, получает атрибуты форматирования первого символа в выделенном в редакторе тексте, а возвращаемое функцией значение содержит маску атрибутов, которые установлены для всего выделенного текста.

Соответствующая ей функция SetSelectionCharFormat позволяет установить атрибуты для выделенного в редакторе текста:

BOOL CRichEditCtrl::**SetSelectionCharFormat**(CHARFORMAT& cf) —

для назначения параметров необходимо в структуре CHARFORMAT, на которую указывает cf, задать те атрибуты, которые необходимо изменить (при этом изменяются только те атрибуты, которые заданы в поле dwMask).

Для форматирования текущего выделенного слова в редакторе предназначена следующая функция:

BOOL CRichEditCtrl::**SetWordCharFormat**(CHARFORMAT & cf) —

устанавливает атрибуты форматирования для текущего выделенного слова, и ее параметры целиком идентичны одноименным параметрам описанных выше функций.

Следует отметить, что у символов, для которых установлен атрибут "только для чтения", редактор не позволяет изменять атрибуты, и в случае попытки пользователя сделать это, редактор посылает уведомляющее сообщение EN_PROTECTED родительскому окну, которое должно разрешить или запретить данное изменение. Для того чтобы родительское окно получало такое сообщение, следует включить данную возможность при помощи вызова функции SetEventMask.

В большинстве случаев для управления форматом символов достаточно использовать только функции GetSelectionCharFormat и SetSelectionCharFormat, что мы и делаем в функции CRichEditFrame::OnFormatFont в примере данной части книги:

```
void CRichEditFrame::OnFormatFont()
{
  CHARFORMAT cf;
  m_REditCtrl.GetSelectionCharFormat(cf);
  CFontDialog dlg(cf);
  if (dlg.DoModal() == IDOK)
  {
    dlg.GetCharFormat(cf);
    m_REditCtrl.SetSelectionCharFormat(cf);
  }
  m_REditCtrl.SetFocus();
}
```

Форматирование абзацев

Расширенный редактор предоставляет возможность форматирования не только символов, но и абзацев. Атрибуты абзаца включают выравнивание, позиции табуляции, отступы и маркировку. Для форматирования абзацев предназначена функция :

DWORD CRichEditCtrl::**GetParaFormat**(PARAFORMAT & pf) —

позволяет получить атрибуты форматирования текущего абзаца и возвращает значение поля dwMask структуры PARAFORMAT, на которую указывает параметр pf данной функции.

Структура PARAFORMAT определена следующим образом:

```
typedef struct _paraformat {
  UINT cbSize;
  _WPAD _wPad1;
  DWORD dwMask;
  WORD wNumbering;
  WORD wReserved;
  LONG dxStartIndent;
  LONG dxRightIndent;
  LONG dxOffset;
  WORD wAlignment;
  SHORT cTabCount;
  LONG rgxTabs[MAX_TAB_STOPS];
} PARAFORMAT;
```

Поле cbSize содержит размер структуры в байтах. Поле dwMask содержит информацию о том, какие из полей структуры содержат действительную информацию, и может быть комбинацией следующих значений:

PFM_ALIGNMENT	Поле wAlignment содержит значение
PFM_NUMBERING	Поле wNumbering содержит значение
PFM_OFFSET	Поле dxOffset содержит значение
PFM_OFFSETINDENT	Поле dxStartIndent содержит относительное значение
PFM_RIGHTINDENT	Поле dxRightIndent содержит значение
PFM_STARTINDENT	Поле dxStartIndent содержит значение
PFM_TABSTOPS	Поля cTabCount и rgxTabs содержат значения

Если заданы одновременно PFM_STARTINDENT и PFM_OFFSETINDENT, то PFM_STARTINDENT имеет преимущество.

Поле wNumbering задает маркировку, может быть равно нулю или PFN_BULLET. Поле dxStartIndent задает отступ первой строки абзаца. Если установлено форматирование абзаца и задано значение PFM_OFFSETINDENT, то этот параметр трактуется как относительное значение, которое добавляется к начальному отступу каждого редактируемого абзаца. Поле dxRightIndent задает размер отступа справа от правого поля. Поле dxOffset — отступ второй и последующих строк абзаца относительно начального выравнивания. Первая строка имеет отступ, если данный параметр отрицателен, и имеет отступ в обратную сторону, если он положителен. Поле wAlignment задает выравнивание абзаца. Может принимать одно из следующих значений:

PFA_LEFT	Абзац выравнивается по левому краю
PFA_RIGHT	Абзац выравнивается по правому краю
PFA_CENTER	Абзац выравнивается по центру

Поле cTabCount задает количество позиций табуляции, определенных в поле rgxTabs, являющихся массивом абсолютных позиций табуляции.

Если функция CRichEditCtrl::GetParaFormat вызвана для нескольких абзацев одновременно, то поле pf получает атрибуты первого из них. При этом возвращаемое функцией значение указывает, какие из атрибутов установлены для всех выделенных абзацев.

Для установки параметров абзаца служит следующая функция:

BOOL CRichEditCtrl::**SetParaFormat**(PARAFORMAT & pf) —

назначение параметров аналогично функции GetParaFormat.

В качестве примера использования возможностей по форматированию абзацев была использована только возможность выравнивания:

```
void CRichEditFrame::OnParagraphLeft()
{
  PARAFORMAT pf;
  m_REditCtrl.GetParaFormat(pf);
```

```
  pf.wAlignment = PFA_LEFT;
  m_REditCtrl.SetParaFormat(pf);
}
void CRichEditFrame::OnParagraphCenter()
{
  PARAFORMAT pf;
  m_REditCtrl.GetParaFormat(pf);
  pf.wAlignment = PFA_CENTER;
  m_REditCtrl.SetParaFormat(pf);
}
void CRichEditFrame::OnParagraphRight()
{
  PARAFORMAT pf;
  m_REditCtrl.GetParaFormat(pf);
  pf.wAlignment = PFA_RIGHT;
  m_REditCtrl.SetParaFormat(pf);
}
```

На рис. 27.1 представлен результат работы редактора, созданного на базе класса CRichEditCtrl.

Примечание

Маркировка в верхней части редактора получена в результате копирования части текста из редактора Word, т. к. в самом приложении реализовано только выравнивание для абзацев.

Расширенный редактор и объекты OLE

Как уже упоминалось, для того чтобы реализовать вставку объектов OLE в редактор и позволить осуществлять с этим объектом дополнительные действия (изменение размеров и редактирование), в чистом виде класс CRichEditCtrl не подходит. Следующая пара функций, предназначенных для работы с объектами OLE, обеспечивает связь редактора с необходимыми в работе OLE-интерфейсами.

IRichEditOle* CRichEditCtrl::**GetIRichEditOle**( ) —

возвращает указатель на интерфейс IRichEditOle для данного редактора или NULL, если интерфейс недоступен.

BOOL CRichEditCtrl::**SetOLECallback**(IRichEditOleCallback* pCallback) —

напротив, устанавливает для расширенного редактора интерфейс OLE, который и позволит обрабатывать объекты OLE. Параметр pCallback является указателем на объект IRichEditOleCallback, который будет использоваться редактором для получения ресурсов OLE и требуемой информации. Объект CRichEditCtrl должен вызвать функцию

IUnknown::AddRef для увеличения счетчика ссылок на объект, заданный параметром pCallback.

Поскольку в данную книгу не вошли вопросы работы с объектами OLE, то для получения более подробной информации по этому вопросу рекомендуется обратиться к другой литературе на эту тему.

Рис. 27.1. Пример форматирования абзаца

Заканчивая обзор собственно расширенного редактора, который "в чистом виде", как вы увидели, поддерживает только возможности форматирования символов текста и абзацев, следует сказать, что для реализации редактора, позволяющего включать в текст самые разнообразные объекты, такие как рисунки, видео- и аудиоклипы и т. п. все уже сделано! (В рамках библиотеки классов MFC, конечно.)

Однако при помощи расширенного редактора или на его основе реализовать что-то подобное MS Word попросту невозможно. Дело в том, что сам по себе расширенный редактор не поддерживает ни нумерацию абзацев, ни их выравнивание по ширине, ни создание таблиц или рисунков. Чтобы это сделать, потребуется написать тысячи строк кода, и интерфейс OLE здесь не поможет.

К счастью, необходимость в создании мощного редактора у вас вряд ли возникнет, как и у кого-нибудь другого, — есть много редакторов как общего пользования или универсальных (MS Word, Word Perfect), так и редакторов для конкретного применения (PageMaker, CorelDRAW!).

Применение расширенного редактора может оказаться полезным для реализации функций быстрого просмотра, генерации простых отчетов с элементами форматирования и т. п. Это средство легко вставляется в приложения, а механизм OLE обеспечивает доступ к бесконечному множеству форматов и приложений.

Расширенный редактор в рамках архитектуры "документ/представление"

В предыдущем разделе был рассмотрен один из самых общих элементов управления — расширенный текстовый редактор. Был представлен класс CRichEditCtrl с его функциями, позволяющими создавать и работать с данным элементом управления. Но стоит рассмотреть его и с другой стороны.

Вы уже познакомились с архитектурой "документ/представление" (document/view) — мощным средством, заложенным в библиотеку MFC. Эта архитектура, применительно к расширенному редактору, охватывает следующие классы:

- CRichEditView — обеспечивает функционирование расширенного редактора в рамках архитектуры "документ/представление", отвечая за отображение данных документа и за взаимодействие с пользователем
- CRichEditDoc — отвечает за создание собственно документа и за работу с объектами OLE
- CRichEditCntrItem — обеспечивает хранение объектов OLE

Примером реализации расширенного редактора на базе класса CRichEditView и классов CRichEditDoc и CRichEditCntrItem является такое приложение, как WordPad. WordPad входит в комплект поставки, начиная с Windows 95, и может быть установлен на компьютере при инсталляции операционной системы как одно из приложений. Этот редактор известен многим пользователям Windows, и хотя он не является таким мощным средством редактирования текстов, как MS Word, однако пригоден для создания различных документов, содержащих элементы форматирования символов и абзацев, позволяет делать вставки в документ элементов, создаваемых другими приложениями, например, включать цветные картинки из графического редактора или звуковые комментарии.

На рис. 27.2 и 27.3 показан пример документа, в который вставлена таблица, созданная в MS Word 97, причем на рис. 27.3 показан пример редактирования этой таблицы "по месту" (средствами MS Word 97, без выхода из редактора).

Само приложение-редактор создано автоматически при помощи средства AppWizard без применения ручного программирования вообще. Тем не менее, даже такое приложение является хорошей базой для развития в более серьезный редактор.

Рис. 27.2. Документ, содержащий таблицу

Рис. 27.3. Редактирование "по месту"

Развитием может быть как раз нечто подобное приложению WordPad, исходные тексты которого поставляются вместе с остальными примерами, входящими в состав пакета Visual C++, и могут быть хорошей иллюстрацией применения указанных классов для реализации приложения, использующего все возможности расширенного редактора.

Перейдем к более близкому знакомству с классами CRichEditView, CRichEditDoc и CRichEditCntrItem.

Класс CRichEditView

Класс CRichEditView — это класс, производный от класса CCtrlView, по своему назначению аналогичный классу CEditView. Он является представлением, содержащим элемент управления RICH EDIT, и может быть использован для реализации на его основе расширенного редактора. Так же, как и объекты класса CEditView, объекты данного класса могут использоваться с документами и шаблонами документов.

Данный класс реализует в полной мере все возможности, представляемые расширенным редактором, что означает кроме ввода и исправления текста ряд следующих функций:

- Форматирование символов — позволяет менять шрифт, его размер, цвет, полужирное и наклонное начертание, использовать атрибут "только для чтения"
- Форматирование абзацев — позволяет изменять выравнивание текста, задавать отступы, устанавливать позиции табуляции и маркировки
- Вставка объектов OLE — позволяет включать в документ цветные иллюстрации, звуковые комментарии, музыкальные или видеофрагменты

Конструктор класса не имеет аргументов:

CRichEditView::**CRichEditView**( )

Как уже говорилось, данный класс реализует такой элемент управления, как расширенный редактор в рамках архитектуры "документ-представление", соответственно класс имеет функцию, возвращающую ссылку на объект класса CRichEditCtrl, который собственно и реализует сам элемент управления — расширенный редактор:

CRichEditCtrl& CRichEditView::**GetRichEditCtrl**( )

После получения ссылки на класс CRichEditCtrl появляется возможность воспользоваться его функциями для обеспечения требуемой работы расширенного редактора.

Также имеется функция, возвращающая ссылку на связанный с данным классом класс CRichEditDoc, обеспечивающий создание и работу с собственно документом:

CRichEditDoc* CRichEditView::**GetDocument**( )

Атрибуты форматирования

Основное преимущество расширенного редактора перед простым редактором — это возможность форматирования текста. Данный класс предоставляет для этого целый ряд функций:

CHARFORMAT& CRichEditView::**GetCharFormatSelection**( ) —

возвращает структуру типа CHARFORMAT, которая содержит атрибуты форматирования текущего выделения в редакторе.

void CRichEditView::**SetCharFormat**(CHARFORMAT cf) —

позволяет установить необходимые атрибуты

Для получения атрибутов форматирования абзацев имеется функция:

PARAFORMAT& CRichEditView::**GetParaFormatSelection**( ) —

возвращает ссылку на структуру типа PARAFORMAT, содержащую информацию об атрибутах форматирования абзацев.

void CRichEditView::**SetParaFormat**(PARAFORMAT& pf) —

позволяет установить новые атрибуты для выделенного абзаца:

Эти четыре функции дублируют функции класса CRichEditCtrl и могут быть использованы вместо них. Следующая функция задает форматирование символов только для выбранного текста:

void CRichEditView::**OnCharEffect**(DWORD dwMask, DWORD dwEffect) —

параметры dwMask и dwEffect задают соответственно, какие параметры форматирования символов для текущего выделенного текста должны быть изменены и список новых значений для этих параметров. Данные поля целиком аналогичны одноименным полям структуры CHARFORMAT.

Выравнивание выбранного абзаца позволяет изменить следующая функция:

void CRichEditView::**OnParaAlign**(WORD wAlign) —

параметр wAlign задает требуемое выравнивание и может принимать одно из следующих значений:

PFA_LEFT	Выравнивать абзац по левому краю
PFA_RIGHT	Выравнивать абзац по правому краю
PFA_CENTER	Выравнивать абзац по центру

Для получения длины текста, содержащегося в редакторе, служит функция, также имеющая аналог в классе CRichEditCtrl:

long CRichEditView::**GetTextLength**( )

Следующая группа функций служит для изменения параметров печати текста:

CSize CRichEditView::**GetPaperSize**( ) —

позволяет получить размеры листа для печати

void CRichEditView::**SetPaperSize**(CSize sizePaper) —

позволяет изменить их нужным образом.

Для получения и изменения текущих установленных полей страницы предназначены следующие две функции:

CRect CRichEditView::**GetMargins**( )

и

void CRichEditView::**SetMargins**(const CRect & rectMargin) —

функцию SetMargins нужно вызвать для установки полей печати для представления расширенного редактора. Если член данных класса m_nWordWrap установлен в значение WrapToTargetDevice, то после использования данной функции нужно вызвать функцию WrapChanged для корректировки характеристик печати.

Получить ширину области печати позволит функция:

int CRichEditView::**GetPrintWidth**( )

А получить полные размеры области печати позволяет функция:

CRect CRichEditView::**GetPrintRect**( )

Размеры страницы, используемые для печати, возвращает функция:

CRect CRichEditView::**GetPageRect**( )

Это была группа функций, предназначенных для изменения и получения различных параметров печати текста.

Следующие две функции предназначены для получения указателей на объекты OLE:

CRichEditCntrItem* CRichEditView::**GetSelectedItem**( ) —

возвращает текущий выбранный объект OLE, содержащийся в редакторе (объект CRichEditCntrItem).

CRichEditCntrItem* CRichEditView::**GetInPlaceActiveItem**( ) —

возвращает указатель на объект OLE, который содержится в расширенном редакторе и находится в данный момент в активном состоянии.

Изменение параметров отображения

Для задания размера отступа для маркированного списка используется компонент данных m_nBulletIndent:

int CRichEditView::**m_nBulletIndent**

По умолчанию этот размер устанавливается равным 720 единицам, что составляет полдюйма (чуть больше сантиметра).

Член данных m_nWordWrap указывает тип переноса слов в данном редакторе:

int CRichEditView::**m_nWordWrap** —

может иметь одно из следующих значений:

WrapNone	Слова не переносятся автоматически
WrapToWindow	Перенос слов базируется на ширине окна
WrapToTargetDevice	Перенос слов базируется на характеристиках установленного устройства вывода

Поиск текста

Класс CRichEditView имеет в своем составе две функции для поиска текста в расширенном редакторе:

BOOL CRichEditView::**FindText**(LPCTSTR lpszFind,
BOOL bCase = TRUE, BOOL bWord = TRUE) —

осуществляет поиск текстовой строки, заданной в lpszFind, параметр bCase, если он установлен, задает, что поиск должен идти с учетом регистра, а bWord — при значении TRUE указывает, что должен вестись поиск целого слова. Данная функция, в отличие от следующей, выводит на экран курсор ожидания, пока идет операция поиска.

BOOL CRichEditView::**FindTextSimple**(LPCTSTR lpszFind,
BOOL bCase = TRUE, BOOL bWord = TRUE) —

также осуществляет поиск заданного текста и имеет те же параметры, но не выводит курсор ожидания.

Для реализации средств поиска, замены и т. п. класс предоставляет вспомогательную функцию:

void CRichEditView::**AdjustDialogPosition**(CDialog* pDlg) —

перемещает на экране заданный параметром pDlg блок диалога так, чтобы он не закрывал собой текущее выделение в редакторе. Она может быть полезна, в частности, для диалогов поиска и замены фрагментов текста в редакторе.

Вставка объектов

Следующая функция позволяет определить, в каком формате содержит данные буфер обмена: в текстовом формате, формате текста расширенного редактора или в формате текста расширенного редактора с объектами OLE:

BOOL CRichEditView::**IsRichEditFormat**(CLIPFORMAT cf)

Проверка возможности вставки содержимого буфера осуществляется следующей функцией:

BOOL CRichEditView::**CanPaste**( ) —

позволяет определить, имеют ли данные в буфере соответствующий формат или нет.

Функция DoPaste позволяет вставить в расширенный редактор объект OLE:

void CRichEditView::**DoPaste**(
COleDataObject& dataobj, CLIPFORMAT cf,
HMETAFILEPICT hMetAPIct) —

объект данных определяется параметром dataobj, параметр hMetAPIct задает метафайл, представляющий вставляемый объект, а cf задает требуемый формат для буфера обмена. Если значение cf равно нулю, то используется представление объекта в виде значка, если же cf не нуль и hMetAPIct не равен NULL, то используется представление объекта из метафайла, заданного в hMetAPIct.

Следующая функция вставляет в расширенный редактор запись как объект OLE:

HRESULT CRichEditView::**InsertItem**(CRichEditCntrItem* pItem) —

применяется для вставки объекта класса CRichEditCntrItem. Параметр pItem должен быть указателем на объект этого класса.

Для вставки в расширенный редактор файла как объекта OLE служит следующая функция:

void CRichEditView::**InsertFileAsObject**(LPCTSTR lpszFileName) —

параметр lpszFileName задает имя вставляемого файла. Данная функция также осуществляет вставку объекта в виде объекта класса CRichEditCntrItem.

Подготовка текста и его печать

Как всегда перед осуществлением печати всего документа или его части, его следует подготовить. Для этого служит следующая функция:

long CRichEditView::**PrintInsideRect**(CDC* pDC,
RECT& rectLayout, long nIndexStart,
long nIndexStop, BOOL bOutput) —

форматирует текст в пределах прямоугольника, заданного параметром rectLayout для устройства, задаваемого в параметре pDC. Обычно после вызова данной функции вызывается функция CRichEditCtrl::DisplayBand, которая и осуществляет вывод отформатированного текста. Параметры nIndexStart и nIndexStop задают индексы первого и последнего символов для определения текста, который должен быть отформатирован. Значение TRUE параметра bOutput задает, что текст должен быть выведен, FALSE — что текст должен быть только измерен.

Следующая функция также осуществляет форматирование текста для заданного устройства:

long CRichEditView::**PrintPage**(CDC* pDC,
long nIndexStart, long nIndexStop) —

параметр pDC определяет устройство вывода, параметры nIndexStart и nIndexStop задают индексы символов, являющихся границами форматируемого текста. Следует иметь в виду, что используются не логические, а физические размеры листа, и вследствие того, что большинство принтеров имеют поля, на которых они физически не могут печатать, то форматирование текста без учета этих технологических полей приведет к тому, что текст по краям листа будет обрезан. Поэтому перед печатью следует установить поля вызовом функции SetMargins. Получить значения полей можно при помощи функций GetPageRect и GetPrintRect. Для вывода текста на печать следует воспользоваться, так же, как и при использовании предыдущей функции, функцией CRichEditCtrl::DisplayBand.

Переопределяемые функции

В классе есть ряд виртуальных функций:

virtual void CRichEditView::**OnInitialUpdate**( ) —

обновляет представление, когда оно впервые присоединяется к документу, но до того, как оно выведено на экран. Реализация по умолчанию вызывает функцию CView::OnUpdate. Данную функцию следует переопределить для выполнения инициализации. Так, если приложение работает с документом фиксированной длины, то в данной

функции можно установить границы для прокрутки в окне в соответствии с размером документа. Если же документ имеет переменную длину, то нужно воспользоваться функцией OnUpdate для обновления границ прокрутки всякий раз, когда изменяются размеры документа.

virtual BOOL CRichEditView::**IsSelected**(const CObject* pDocItem) —

показывает, выбран указанный объект OLE или нет. Параметр pDocItem является указателем на объект, находящийся в представлении. Данную функцию следует переопределить, если ваш собственный класс представления имеет отличающийся метод для обработки выделения объектов OLE.

Поиск и замена

Группа переопределяемых функций для поиска и замены текста:

virtual void CRichEditView::**OnFindNext**(LPCTSTR lpszFind, BOOL bNext, BOOL bCase, BOOL bWord) —

осуществляет поиск строки в тексте. Параметр lpszFind задает строку для поиска; параметр bNext — направление (TRUE — вперед, а FALSE — назад); Параметр bCase — задает учет регистра при выполнении поиска; параметр bWord — определяет, будет ли производиться сравнение при поиске целыми словами или нет. Данную функцию следует переопределить, если имеется желание задать свои критерии для поиска строки.

Если же заданный текст не был найден, то библиотека вызывает для обработки следующую функцию:

virtual void CRichEditView::**OnTextNotFound**(LPCTSTR lpszFind) —

в параметре lpszFind содержится текст поиска. Реализация по умолчанию данной функции вызывает системную функцию MessageBeep; если нужно вывести сообщение о неудачном поиске, то можно переопределить данную функцию.

Следующая функция осуществляет не только поиск строки, но и замену ее на новую:

virtual void CRichEditView::**OnReplaceAll**(LPCTSTR lpszFind, LPCTSTR lpszReplace, BOOL bCase, BOOL bWord) —

параметр lpszReplace задает новую строку, которой должна заменяться найденная строка, а остальные параметры данной функции аналогичны одноименным параметрам функции OnFindNext. Функция вызывается библиотекой, когда в блоке диалога поиска и замены текста выбрана опция замены всех вхождений искомой строки в тексте на новую. Данную функцию следует переопределить, если требуется внести свои критерии для осуществления поиска в представлении документа.

Для замены одного выбранного фрагмента текста новым текстом библиотека вызывает следующую функцию:

virtual void CRichEditView::**OnReplaceSel**(LPCTSTR lpszFind, BOOL bNext, BOOL bCase, BOOL bWord, LPCTSTR lpszReplace) —

параметры целиком аналогичны одноименным параметрам предыдущих функций. Данную функцию также следует переопределить, если требуется внести свои критерии для осуществления поиска текста.

Класс CRichEditDoc

Данный класс является производным от классов, обеспечивающих функциональные возможности для работы с документами в архитектуре "документ/представление", а также от классов, обеспечивающих работу документа как сервера объектов OLE, и наследует все возможности вышележащих классов по работе с документами и объектами OLE. Кроме того, класс имеет некоторые дополнительные свойства.

Для того чтобы получить указатель на объект типа CRichEditView, с которым связан данный объект класса CRichEditDoc, предназначена функция GetView:

CRichEditView* CRichEditDoc::**GetView**( )

Класс содержит переопределяемую функцию для создания объекта класса CRichEditCntrItem:

virtual CRichEditCntrItem* CRichEditDoc::**CreateClientItem**(
REOBJECT* preo = NULL) —

создает объект класса CRichEditCntrItem, добавляет его в данный документ и возвращает указатель на него. Данная функция не выполняет какой-либо инициализации OLE. Параметр preo является указателем на структуру типа REOBJECT, описывающую создаваемую запись OLE, на основе которой и создается новый объект класса CRichEditCntrItem.

Для определения того, содержит ли документ информацию о форматировании, служит следующая функция:

int CRichEditDoc::**GetStreamFormat**( ) —

указывает, должны ли входной и выходной потоки включать информацию о форматировании, и возвращает один из двух флагов:

SF_TEXT	Показывает, что редактор не содержит информации о форматировании
SF_RTF	Показывает, что редактор содержит информацию о форматировании

Возвращаемое функцией значение основывается на состоянии члена данных класса m_bRTF: если оно равно TRUE, то возвращается SF_RTF, иначе — SF_TEXT.

Глава 28

Наборы свойств

Набор свойств представляет собой особый вид диалогового окна, применяемый для модификации свойств как самого приложения, так и каких-то внешних объектов. Набор свойств содержит три основные части:

- *Блок диалога*, внутри которого размещаются все остальные элементы набора свойств
- *Одна или несколько страниц свойств*. Одновременно на экране может быть видна только одна из них, находящаяся как бы поверх других
- *Вкладки*, расположенные вверху каждой страницы. Пользователь, выбирая одну из вкладок, выбирает соответствующую ей страницу свойств

В качестве примера набора свойств можно привести блок диалога просмотра и изменения параметров Visual C++ (рис. 28.1):

Рис. 28.1. Окно **Options** оболочки Visual C++ 6

Использовать набор свойств весьма удобно в тех случаях, когда имеются группы некоторых параметров, которые пользователь может изменять. Он позволяет организовать большое количество информации в небольшие группы, выводимые на экран поочередно, что значительно облегчает и упрощает работу.

Наборы свойств могут применяться не только для отображения свойств объектов, но и для группирования сходных по назначению или по используемым параметрам действий. Например, приложение MS Word 97 таким образом группирует действия для поиска, замены и перехода (рис. 28.2):

Рис. 28.2. Окно **Найти и заменить** приложения MS Word 97

Создание набора свойств

Библиотека MFC предоставляет специальные классы для создания набора свойств и обеспечения работы с ним: CPropertySheet и CPropertySheetEx позволяют создать набор свойств, CPropertyPage и CPropertyPageEx — классы, предназначенные для создания страниц свойств, которые включаются в набор свойств. Несколько особняком стоит класс CTabCtrl, позволяющий создавать вкладки и работать с ними. Дело в том, что в общем случае нет необходимости напрямую работать с вкладками в наборе свойств, достаточно лишь задать текст, который нужно поместить на них, а все остальное сделает библиотека, но все же есть возможность получения доступа к объектам данного класса и использования его функций для создания вкладок, отличающихся от создаваемых по умолчанию.

Проще всего создать набор свойств, воспользовавшись средством добавления в проект различных программных компонентов, включающих и набор свойств. Для этого следует выбрать команду **Insert\Component**, а затем вставить в проект набор свойств с требуемыми характеристиками (рис.28.3).

При этом определяется следующее:

- Будет ли добавляемый компонент набором свойств или мастером

- Поддерживается ли возможность предварительного просмотра произведенных изменений (по сути это добавление рисованного объекта и ничего больше)
- Будет ли набор свойств модальным или нет
- Задается класс из набора существующих в проекте классов, который будет иметь функцию создания набора свойств и возможность доступа к его элементам
- Количество страниц
- Последнее, что можно сделать, — заменить имена создаваемых классов

Если впоследствии понадобится добавить новые страницы свойств, для этого следует воспользоваться механизмом, изложенным ниже.

Рис. 28.3. Окно выбора добавляемых в проект программных компонентов

Создание страницы свойств

Для того чтобы включить набор свойств в приложение, необходимо в первую очередь создать шаблоны ресурсов диалога для каждой из страниц свойств, которые затем войдут в набор свойств. При этом необязательно придерживаться правила, что все страницы должны иметь один размер — их размеры могут отличаться, в этом случае библиотека будет ориентироваться на размер самой большой страницы для выделения места в наборе свойств, и размеры остальных страниц будут подогнаны под этот размер. Данная операция ничем не отличается от создания шаблона обычного блока диалога. Для этого, как и раньше, можно (и нужно) пользоваться редактором ресурсов. Для использования созданной страницы в наборе свойств следует

соблюсти некоторые условия — необходимо не забыть задать в разделе **Dialog Properties** при создании шаблона страницы следующие стили:

- В поле **Caption** группы **General** — текст, который должен появиться на вкладке, принадлежащей данной странице свойств
- Стиль **Child** в группе **Styles** — все страницы свойств являются дочерними окнами
- В группе **Styles** окна **Border** должен быть выбран тип рамки **Thin**
- Стиль **Titlebar** в группе **Styles** — иначе вкладка, связанная со страницей, будет безымянной
- Стиль **Disabled** в группе **More Styles**

Примечание

Как вы понимаете, создавать набор свойств можно и не используя редактор ресурсов — для этого в классах имеются специальные функции. Но зачем усложнять себе жизнь, выполняя работу, которую удобнее поручить другому, а самому заняться более интересными вопросами!

После того как создан шаблон блока диалога и в нем размещены необходимые элементы управления, можно перейти к следующей операции — используя средство ClassWizard, создать объект, производный от класса CPropertyPage и связанный с созданным шаблоном страницы набора свойств. Для этого необходимо, не закрывая только что созданный шаблон страницы свойств в редакторе ресурсов, выбрать в главном меню пункт **View\ClassWizard**. При этом фокус ввода должен принадлежать созданному шаблону страницы свойств. В этом случае автоматически запустится ClassWizard и появится диалоговое окно с предложением добавить новый класс в приложение. Нужно выбрать пункт **Add New Class** и в следующем появившемся окне **Create New Class** выбрать в списке класс **CPropertyPage** в качестве базового (опция **Base class** группы **Class information**), следует также обязательно ввести имя нового класса.

Затем нужно с помощью ClassWizard создать переменные для хранения значений свойств, задаваемых в данной странице свойств. Процесс создания переменных в ClassWizard (кнопка **Add Variable**) полностью аналогичен процессу создания переменных для обычного блока диалога, ведь страница свойств также является хоть и специализированным, но все же блоком диалога.

Создание объекта "набор свойств"

Следующим шагом будет создание объекта класса CPropertySheet в тесте программы. Обычно объект класса CPropertySheet создается в функции-обработчике команды, осуществляющей вывод на экран набора свойств.

```
CPropertySheet propsheet;
```

Конструктор класса имеет несколько реализаций:

CPropertySheet::**CPropertySheet**( ),
CPropertySheet::**CPropertySheet**(UINT nIDCaption,
CWnd* pParentWnd = NULL, UINT iSelectPage = 0),
CPropertySheet::**CPropertySheet**(LPCTSTR pszCaption,
CWnd* pParentWnd = NULL, UINT iSelectPage = 0) —

nIDCaption — идентификатор заголовка для набора свойств; pParentWnd — указатель на родительское окно. Если его значение равно NULL, то родительским окном является главное окно приложения; iSelectPage — индекс страницы, выводимой первой при отображении набора свойств на экран. По умолчанию выводится первая страница набора свойств; pszCaption — указатель на строку, содержащую заголовок для набора свойств (не может быть равным NULL).

```
CPropertySheet propsheet("Параметры", pParentWnd);
```

или

```
CPropertySheet propsheet("Параметры ", GetParent());
```

Если не нужно добавлять в набор свойств дополнительные элементы управления (например, окно предварительного просмотра) или выводить на экран немодальный набор свойств, вообще не требуется создавать класс, производный от класса CPropertySheet.

Примечание

Немодальный набор свойств по умолчанию не содержит кнопок, и скорее всего в него потребуется добавить элементы управления для того, например, чтобы его можно было в дальнейшем, по крайней мере, закрыть.

Настройка окна набора свойств

По умолчанию набор свойств выделяет окно для размещения страниц свойств с вкладками, для модальных наборов свойств в этом окне также размещаются кнопки **OK**, **Cancel** (отмена) и **Apply** (применить). Можно добавлять другие необходимые для работы элементы управления. Например, можно добавить в набор свойств дополнительное окно просмотра, в котором будут выводиться изображения, находящиеся в выбираемых пользователем файлах. Дополнительные элементы управления можно добавлять в обработчике OnCreate. Однако лучше это делать в функции OnInitDialog, т. к. на момент вызова этой функции уже созданы окна страниц свойств, доступ к которым также может понадобиться:

```
BOOL CPropSheet::OnInitDialog()
{
  BOOL bResult = CPropertySheet::OnInitDialog();
  // Добавляем окно просмотра в набор свойств
```

```
  CRect rectWnd;
  GetWindowRect(rectWnd);
  SetWindowPos(NULL, 0, 0,
    rectWnd.Width() + 100, rectWnd.Height(),
    SWP_NOMOVE | SWP_NOZORDER | SWP_NOACTIVATE);
  CRect rectPreview(rectWnd.Width() + 25, 25,
  rectWnd.Width()+75, 75);
  m_wndPreview.Create(NULL, NULL, WS_CHILD | WS_VISIBLE,
    rectPreview, this, 0x1000);
  CenterWindow();
  return bResult;
}
```

Вставка дополнительных элементов управления в набор свойств обычно требует увеличения размеров его диалогового окна. После вызова функции базового класса OnCreate или OnInitDialog соответственно необходимо вызвать функцию CWnd::GetWindowRect для получения значений высоты и ширины текущего окна набора свойств, задать требуемые размеры и вызвать функцию CWnd::MoveWindow или CWnd::SetWindowPos для изменения размеров окна, в котором выводится набор свойств.

Добавление страниц

Наиболее удобный способ обеспечения доступа к страницам свойств внутри набора свойств является встраивание объектов CPropertyPage в класс, производный от CPropertySheet. Такой подход отличается от разработки модальных наборов свойств, когда объект-владелец набора свойств создает объекты CPropertyPage, а затем передает их набору свойств, вызывая функцию CPropertySheet::AddPage.

На этом этапе следует сделать еще одно важное замечание: если нужно создать несколько объектов класса CPropertySheet, например массив наборов свойств, то следует воспользоваться не конструктором CPropertySheet, а функцией Construct:

void CPropertySheet::**Construct**(UINT nIDCaption,
    CWnd* pParentWnd = NULL, UINT iSelectPage = 0)

и

void CPropertySheet::**Construct**(LPCTSTR pszCaption,
    CWnd* pParentWnd = NULL, UINT iSelectPage = 0) —

параметры те же, что и у конструктора класса CPropertySheet. Ее можно вызывать, если ни один из конструкторов с параметрами данного класса еще не вызывался. В случае создания массива объектов CPropertySheet следует обязательно вызвать данную функцию для каждого элемента массива.

Ниже приведен пример, иллюстрирующий применение этих функций:

```
int i;
CPropertySheet propsheetarray[3];
// Для одиночного набора свойств не требуется
// вызывать функцию Construct
CPropertySheet oneSheet;
UINT sheetsID[3] = {IDS_SHEET_1, IDS_SHEET_2, IDS_SHEET_3};
for (i = 0; i < 3; i++)
{
  // функция Construct вызывается для каждого
  // создаваемого набора свойств
  propsheetarray[i].Construct(sheetsID[i]);
}
```

Следующим важным шагом на пути создания набора свойств будет выполнение следующих операций для каждой добавляемой в набор свойств страницы:

- Необходимо создать по одному объекту каждого класса, производного от класса CPropertyPage, который ранее был создан с помощью ClassWizard
- Вызвать функцию CPropertySheet::AddPage для каждой страницы, которая должна быть добавлена в набор свойств

Обычно функция, в которой создается объект CPropertySheet, на этом же шаге также создает и объекты класса CPropertyPage. Однако если создается производный от CPropertySheet класс, то можно встроить объекты класса CPropertyPage в объект класса CPropertySheet и вызвать из конструктора производного от CPropertySheet класса функцию CPropertySheet::AddPage для каждой страницы свойств. Функция AddPage добавляет страницу свойств в набор свойств, однако не создает окно для этой страницы. Следовательно, необязательно ждать создания окна набора свойств и можно вызвать функцию AddPage непосредственно из конструктора, для того чтобы добавить страницу в набор свойств.

```
// Первый способ
CPropertySheet propsheet;
CFirstPage page1;  // Класс, производный от CPropertyPage
CSecondPage page2; // Класс, производный от CPropertyPage
propsheet.AddPage(&page1);
propsheet.AddPage(&page2);
// Второй способ
class CPropSheet : public CPropertySheet
{
```

```
public:
  CPropSheet(CWnd* pParentWnd = NULL);
public:
  CFirstPage m_Page1;  // Класс, производный от CPropertyPage
  CSecondPage m_Page2; // Класс, производный от CPropertyPage
...
};
CPropSheet::CPropSheet(CWnd* pWndParent)
  : CPropertySheet(IDS_PROPSHT_CAPTION, pWndParent)
{
  AddPage(&m_Page1);
  AddPage(&m_Page2);
}
```

Для вывода на экран набора свойств необходимо вызвать одну из функций: CPropertySheet::DoModal или CPropertySheet::Create.

Создание модального окна свойств

Для того чтобы создать набор свойств как модальный блок диалога, необходимо вызвать функцию DoModal:

virtual int CPropertySheet::**DoModal**( ) —

> выводит на экран модальный набор свойств. Функция возвращает значения IDOK или IDCANCEL, а в случае неуспешного завершения возвращается 0. Следует иметь в виду, что если набор свойств создан как мастер, то возвращаются другие значения — ID_WIZFINISH или IDCANCEL (см. описание функции SetWizardMode).

После завершения работы функции окна и все страницы, связанные с набором свойств, удаляются. Сами объекты остаются существовать. Как правило, значения данных от объектов CPropertyPage считываются после того, как функция DoModal вернула значение IDOK.

Создание немодального окна набора свойств

Обычно создаваемая страница свойств является модальной. Так же как и модальный диалог, пользователь должен закрыть окно модального набора свойств прежде, чем он сможет работать с другими объектами данного приложения (окнами, меню и т. п.). Однако существуют случаи, когда необходимо создать немодальный набор свойств, который позволяет, не закрыв его, перейти к работе с другими объектами приложения. Для того чтобы создать немодальный набор свойств, следует вызвать функцию CPropertySheet::Create, вместо CPropertySheet::DoModal.

BOOL CPropertySheet::**Create**(CWnd* pParentWnd = NULL,
DWORD dwStyle = (DWORD) −1, DWORD dwExStyle = 0) —

pParentWnd — указатель на родительское окно (если равен NULL, то родительским окном является главное окно приложения); dwStyle — задает стили окна (по умолчанию задается комбинация стилей WS_SYSMENU, WS_POPUP, WS_CAPTION, DS_MODALFRAME, DS_CONTEXT_HELP и WS_VISIBLE); dwExStyle — задает расширенные стили окна (по умолчанию задается стиль WS_EX_DLGMODALFRAME).

Функцию можно вызывать как непосредственно из конструктора, так и после того, как объект набора свойств был уже создан. Функция возвращает управление непосредственно после создания набора свойств. Для уничтожения набора свойств следует вызвать функцию CWnd::DestroyWindow.

При создании немодального набора свойств придется выполнить несколько дополнительных операций. Одной из них является обеспечение обмена данными между набором свойств и внешними объектами в то время, когда окно набора свойств открыто. Эта задача целиком совпадает с аналогичной задачей для обычных немодальных блоков диалога. В данном случае такой обмен проще реализовать, создав производный от CPropertySheet класс. Одним из методов обеспечения обмена является определение указателя из набора свойств на внешний объект. Далее следует определить функцию в классе, производном от CPropertySheet, которая будет отслеживать изменения данных набора свойств всякий раз, когда фокус ввода передается от одного внешнего объекта другому. Она вызывается, естественно, не объектом набора свойств, а самим приложением. Такая функция осуществляет сброс текущих данных каждой страницы для отображения свойств текущего внешнего объекта. Для этого она должна иметь доступ к объектам CPropertyPage, определенным в классе CPropertySheet.

Обмен данными

Кроме вывода на экран набора свойств, следует обеспечить обмен данными между страницами свойств и объектом-владельцем набора свойств. Как и при работе со всеми блоками диалога, процесс обмена между набором свойств и приложением играет очень важную роль (что впрочем неудивительно, ведь набор свойств и служит для того, чтобы задавать требуемые значения или параметры, необходимые для работы приложения).

Обмен данными с набором свойств реально является обменом данными с его отдельными страницами. Процедура обмена при этом ничем не отличается от обмена в обычном блоке диалога в силу того, что объект класса CPropertyPage является просто специализированным объектом класса CDialog. При этом используются средства обмена данными диалога (dialog data exchange, DDX), которые позволяют осуществлять обмен между элементами управления блока диалога и переменными объекта-владельца блока диалога.

Важное различие в обмене данными между обычным блоком диалога и набором свойств заключается в том, что набор свойств имеет в своем составе несколько страниц, для каждой из которых необходимо обеспечить обмен данными.

Операция Apply

Следует остановиться еще на одной важной детали — обработке нажатия кнопки **Apply** (применить). Набор свойств имеет одну важную способность, которой не имеют обычные блоки диалога, а именно — позволяет пользователям изменять параметры, задаваемые на странице свойств, во внешнем объекте до того, как окно с набором свойств будет закрыто. Это осуществляется с помощью кнопки **Apply**, нажатие которой приводит к изменению у объекта *всех* параметров *всех* страниц в наборе свойств, а не только для текущей активной страницы свойств. По умолчанию эта кнопка недоступна. Необходимо самостоятельно написать код программы, делающий ее доступной для использования в требуемый момент работы с набором свойств. Необходимо также написать код, реализующий выполнение необходимых действий при нажатии этой кнопки.

Если не требуется предоставлять пользователю описанную возможность, то в общем нет необходимости удалять саму кнопку. Ее можно оставить недоступной для того, чтобы сохранить общность с другими приложениями Windows. Для того чтобы объявить, что параметры в странице свойств изменились и сделать кнопку **Apply** доступной, следует вызвать функцию CPropertyPage::SetModified со значением ее единственного параметра равным TRUE. Если какая-либо из страниц свойств содержит изменения, которые еще не применены к объекту, то даже если текущая страница таких свойств не содержит, кнопка **Apply** все равно будет доступна. Это значит, что достаточно вызвать функцию SetModified хотя бы для одной из страниц, чтобы сделать кнопку **Apply** набора свойств доступной. Функцию SetModified следует вызывать всякий раз, когда пользователь внес какие-либо изменения параметров. Одним из путей определения этого является обработка уведомляющих сообщений от элементов управления страницы свойств, таких, например, как EN_CHANGE или BN_CLICKED.

При нажатии кнопки **Apply** набор свойств должен сообщить своему владельцу или какому-то другому внешнему объекту приложения о том, что необходимо модифицировать текущие параметры в наборе свойств. Одновременно кнопка должна стать недоступной, для чего нужно вызвать функцию SetModified со значением параметра равным FALSE. Однако чтобы сделать кнопку недоступной, необходимо вызвать эту функцию для каждой из страниц свойств, для которых был произведен вызов функции SetModified со значением параметра равным TRUE, в противном случае будет считаться, что операция по обновлению не была совершена. Обратите внимание, что только Windows хранит информацию о состоянии страницы. Так что при

небольшом количестве страниц свойств можно смело вызвать функцию SetModified со значением параметра равным FALSE для всех страниц набора свойств, чтобы гарантировать результат.

Для создания обработчика команды **Apply** следует использовать следующий код:

```
BEGIN_MESSAGE_MAP(CPropSheet, CPropertySheet)
  //{{AFX_MSG_MAP(CPropPSheet)
  ON_COMMAND(ID_APPLY_NOW, OnApplyNow)
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()
void CPropSheet::OnApplyNow()
{
  Default();
  ...
  m_Page1.SetModified(FALSE);
  m_Page2.SetModified(FALSE);
}
```

Далее, как обычно, приведем обзор классов CPropertySheet и CPropertyPage, используемых для создания набора свойств, а также новых — CPropertySheetEx и CPropertyPageEx, реализующих некоторые дополнительные возможности.

Класс CPropertySheet

Как уже говорилось, данный класс обеспечивает возможность создания набора свойств.

Для того чтобы познакомиться с различными параметрами набора свойств, рассмотрим в первую очередь член данных этого класса m_psh:

PROPSHEETHEADER CPropertySheet::**m_psh**

В структуре PROPSHEETHEADER хранятся основные параметры набора свойств. Ее можно использовать для изменения этих параметров уже после его создания, но до того, как он будет выведен на экран вызовом функции DoModal.

```
typedef struct _PROPSHEETHEADER {
  DWORD dwSize;
  DWORD dwFlags;
  HWND hwndParent;
  HINSTANCE hInstance;
  union {
```

```
    HICON hIcon;
    LPCTSTR pszIcon;
  };
  LPCTSTR pszCaption;
  UINT nPages
  union {
    UINT nStartPage
    LPCTSTR pStartPage;
  };
  union {
    LPCPROPSHEETPAGE ppsp;
    HPROPSHEETPAGE FAR *phpage;
  };
  PFNPROPSHEETCALLBACK pfnCallback;
#if (_WIN32_IE >= 0x0400)
  union {
    HBITMAP hbmWatermark;
    LPCTSTR pszbmWatermark;
  };
  HPALETTE hplWatermark;
  union {
    HBITMAP hbmHeader;
    LPCSTR pszbmHeader;
  };
#endif
} PROPSHEETHEADER
```

dwSize — размер данной структуры в байтах; dwFlags — массив флагов, указывающих, какие переменные в данной структуре должны быть использованы, а какие проигнорированы. Может быть комбинацией следующих значений:

PSH_DEFAULT	Использование значений по умолчанию для всех переменных структуры
PSH_HASHELP	Выводить кнопку **Help** (справка). Эта кнопка будет доступна только в том случае, если флаг PSP_HASHELP также установлен в структуре PROPSHEETPAGE для текущей страницы свойств. Если флаг PSP_HASHELP установлен для какой-либо из страниц в наборе свойств, то кнопка **Help** появится автоматически, независимо от того, установлен или нет флаг PSH_HASHELP. Однако флаг PSH_HASHELP полезен тогда, когда ни одна из страниц не имеет установленного флага PSP_HASHELP, но потом могут быть еще добавлены страницы свойств, содержащие кнопку **Help**

PSH_HEADER	Сообщает, что для заголовка используется специальный битовый массив, который задается в поле pszbmHeader, если установлен флаг PSH_USEHBMHEADER, или в поле hbmHeader, если этот флаг не установлен. Этот флаг игнорируется, если одновременно не установлен флаг PSH_WIZARD97
PSH_MODELESS	Сообщает функции ::PropertySheet, что набор свойств необходимо создать как немодальный блок диалога (по умолчанию создается модальный). Если установлен этот флаг, то функция ::PropertySheet немедленно после создания диалога возвращает дескриптор набора свойств. (При использовании класса CPropertySheet манипуляции флагом осуществляется функциями класса.)
PSH_NOAPPLYNOW	Удаляет кнопку **Apply**
PSH_PROPSHEETPAGE	Сообщает о необходимости использовать поле ppsp структуры и игнорировать phpage
PSH_PROPTITLE	Добавляет слева строку "Properties for" (в русской версии Windows — "Свойства:") к строке, заданной в поле pszCaption
PSH_STRETCHWATERMARK	Растягивает битовый массив, чтобы заполнить фон набора свойств вместо перекрытия "водяных знаков" и битового массива заголовка. Этот флаг игнорируется, если не установлены флаги PSH_WIZARD97 и PSH_WATERMARK, или флаг PSH_HEADER
PSH_USECALLBACK	Сообщает о необходимости вызывать функцию, заданную в поле pfnCallback при инициализации набора свойств, определяемого данной структурой
PSH_USEHBMHEADER	Сообщает, что для задания битового массива заголовка используется поле hbmHeader. Если флаг не установлен, то информация находится в поле pszbmHeader. Этот флаг игнорируется, если не установлены флаги PSH_WIZARD97 и PSH_HEADER
PSH_USEHBMWATERMARK	Сообщает, что для задания битового массива фона используется поле hbmWatermark. Если флаг не установлен, то информация находится в поле pszbmWatermark. Этот флаг игнорируется, если не установлены флаги PSH_WIZARD97 и PSH_WATERMARK
PSH_USEHICON	Сообщает о необходимости использовать пиктограмму, заданную дескриптором в поле hIcon, в заголовке набора свойств
PSH_USEHPLWATERMARK	Для рисования битовых массивов фона и/или заголовка используется палитра, заданная в поле hplWatermark. Этот флаг игнорируется, если не установлены флаги PSH_WIZARD97 и PSH_WATERMARK, или флаг PSH_HEADER

PSH_USEICONID	Сообщает о необходимости использовать значок, заданный именем в поле pszIcon, в заголовке набора свойств
PSH_USEPAGELANG	Язык для набора свойств берется из ресурсов первой страницы
PSH_USEPSTARTPAGE	Сообщает о необходимости использования поля pStartPage и игнорирования поля nStartPage для определения начальной страницы набора свойств
PSH_WATERMARK	Сообщает, что для фона используется битовый массив, который в зависимости от значения флага PSH_USEHBMWATERMARK берется либо из поля pszbmWatermark, либо из поля hbmWatermark. Этот флаг игнорируется, если не установлен флаг PSH_WIZARD97
PSH_WIZARD	Создается мастер
PSH_WIZARD97	Создается набор свойств мастера с возможностью вывода в фоне и/или в заголовке битовых массивов
PSH_WIZARDCONTEXTHELP	В набор свойств мастера добавлено системное меню. Этот флаг игнорируется, если не установлен флаг PSH_WIZARD или PSH_WIZARD97
PSH_WIZARDHASFINISH	В окне мастера отображается кнопка **Finish** (готово). Этот флаг игнорируется, если не установлен флаг PSH_WIZARD или PSH_WIZARD97
PSH_RTLREADING	Задает вывод заголовка набора свойств для чтения справа налево (для арабского языка или иврита)

Примечание

Все флаги, которые требуют наличия установленного флага PSH_WIZARD97, не поддерживаются в классе CPropertySheet. Для работы с этим стилем в библиотеке реализован новый класс — CPropertySheetEx.

hwndParent — дескриптор родительского окна; hInstance — дескриптор объекта, чья пиктограмма или заголовок должны быть загружены (если заданы pszIcon или pszCaption, то данный параметр должен быть заполнен); hIcon — дескриптор значка, помещаемого в заголовке набора свойств (если dwFlags не содержит значения PSH_USEHICON, то данный параметр игнорируется); pszIcon — предназначен для задания значка, который должен быть загружен и помещен в заголовке набора свойств, может содержать идентификатор значка или указатель на строку, задающую ее имя (если в dwFlags не задано значение PSH_USEHICONID, то данный параметр будет проигнорирован); pszCaption — задает заголовок набора свойств. Параметр может содержать идентификатор строкового ресурса или указатель на строку, содержащую заголовок. Если в dwFlags задано значение

PSH_PROPTITLE, то к заголовку будет добавлена слева строка "Properties for" (в русской версии Windows — "Свойства:"); nPages — число элементов в массиве phpage; nStartPage — задает индекс страницы, которая появится на экране первой при создании набора свойств; pStartPage — задает имя начальной страницы, появляющейся на экране первой при создании набора свойств (параметр должен содержать идентификатор строкового ресурса или указатель на строку, содержащую имя); ppsp — указатель на массив структур PROPSHEETPAGE, определяющих страницы набора свойств. Если в dwFlags не задано значение PSH_PROPSHEETPAGE, то данный параметр будет проигнорирован; phpage — указатель на массив, содержащий дескрипторы страниц набора свойств. Каждый дескриптор должен быть создан предшествующим вызовом функции ::CreatePropertySheetPage (если в dwFlags не задано значение PSH_PROPSHEETPAGE, то данный параметр будет проигнорирован); pfnCallback — указатель на функцию приложения, которая вызывается при инициализации набора свойств. Более подробно это рассмотрено в описании функции ::PropSheetProc (если в dwFlags не задано значение PSH_USECALLBACK, то данный параметр будет проигнорирован); hbmWatermark и pszbmWatermark — дескриптор или указатель на текстовую строку ресурса битового массива, используемого для фона набора свойств (что именно задается флагом PSH_USEHBMWATERMARK); hplWatermark — дескриптор палитры, используемой для вывода битовых массивов (если в dwFlags не задано значение PSH_USEHBMHEADER, то данный параметр будет проигнорирован); hbmHeader и pszbmHeader — дескриптор или указатель на текстовую строку ресурса битового массива, используемого для заголовка набора свойств (что именно, задается флагом PSH_USEHBMHEADER).

Как видите, многие параметры требуется задавать только в случае создания набора свойств средствами Win32 API, что достаточно сложно. Гораздо проще воспользоваться тем, что предлагает библиотека классов MFC.

Операции над набором свойств

Несомненный интерес представляют функции, которые предназначены для выполнения операций над набором свойств.

Для добавления страниц в набор свойств служит функция, о которой уже не раз упоминалось выше:

void CPropertySheet::**AddPage**(CPropertyPage* pPage) —

> pPage — указатель на добавляемую страницу (не может быть равным NULL). Эта функция добавляет страницу в набор свойств, располагая ее вкладку справа от последней из уже имеющихся. Страницы в наборе свойств будут располагаться в том же порядке, в каком они добавлялись в него. Функция добавляет объект класса CPropertyPage в список страниц свойств класса CPropertySheet, но не создает окно для его размещения. Библиотека откладывает создание окна для страницы до того момента, пока пользователь не выберет его.

Если функция будет вызвана после того, как на экран был выведен набор свойств, то в строку вкладок будет добавлена вкладка новой страницы.

Чтобы удалить страницу из набора свойств, следует воспользоваться одним из вариантов следующей функции:

void CPropertySheet::**RemovePage**(CPropertyPage* pPage)

или

void CPropertySheet::**RemovePage**(int nPage) —

pPage — указатель на добавляемую страницу (не может быть равен NULL). nPage — индекс страницы, которая должна быть удалена (должен лежать в пределах от 0 до значения, на единицу меньшего числа страниц в наборе свойств).

Функция удаляет страницу из набора свойств и уничтожает ассоциированное с ней окно. Сам объект CPropertyPage не удаляется до того момента, пока владелец окна CPropertySheet не закроет его.

В ходе работы с набором свойств может появиться необходимость определить какие-либо из его атрибутов или изменить их. Для этого имеются следующие функции:

int CPropertySheet::**GetActiveIndex**( ) —

возвращает индекс текущей страницы в наборе свойств.

Полученное значение можно использовать при вызове функции GetPage для получения указателя на эту страницу:

CPropertyPage* CPropertySheet::**GetPage**(int nPage) —

возвращает указатель на заданную страницу. nPage — индекс страницы, указатель на которую необходимо получить (должен лежать в пределах от 0 до значения, на единицу меньшего числа страниц в наборе свойств).

int CPropertySheet::**GetPageIndex**(CPropertyPage* pPage) —

возвращает индекс заданной страницы в наборе свойств. pPage — указатель на страницу, индекс которой должен быть найден. Не должен быть равен NULL.

int CPropertySheet::**GetPageCount**( ) —

возвращает текущее число страниц в наборе свойств.

Следующая пара функций позволяет получить и изменить текущую активную страницу набора свойств:

CPropertyPage* CPropertySheet::**GetActivePage**( ) —

возвращает указатель на текущую активную страницу набора свойств.

Зачастую эта функция используется для определения текущей активной страницы для отображения внесенных изменений:

```
GetActivePage()->UpdateData(FALSE);
```

BOOL CPropertySheet::**SetActivePage**(int nPage)

и

BOOL CPropertySheet::**SetActivePage**(CPropertyPage* pPage) —

позволяет приложению сменить текущую активную страницу в наборе свойств. nPage — индекс страницы, которая должна быть установлена как текущая (должен лежать в пределах от 0 до значения, на единицу меньшего числа страниц в наборе свойств); pPage — указатель на устанавливаемую текущую страницу, не может быть равен NULL.

void CPropertySheet::**SetTitle**(LPCTSTR lpszText, UINT nStyle = 0) —

устанавливает заголовок для набора свойств. lpszText — указатель на текстовую строку, которая должна быть выведена в заголовке набора свойств; nStyle — задает стиль заголовка набора свойств (может быть равен 0 или PSH_PROPTITLE; значение PSH_PROPTITLE включает строку "Properties for" (в русской версии Windows — "Свойства:") в заголовок слева от строки, заданной в lpszText).

Вспомогательные функции

Следует отдельно отметить функцию, позволяющую получить доступ к вкладкам набора свойств:

CTabCtrl* CPropertySheet::**GetTabControl**( ) —

возвращает указатель на объект класса CTabCtrl.

Вкладки неявно входят в набор свойств и в общем случае нет необходимости работать с ними отдельно, достаточно лишь при создании страницы свойств задать текст, который должен будет появиться на вкладке страницы, все остальное за вас сделает библиотека. Но может возникнуть ситуация, когда понадобится произвести некоторые дополнительные действия с вкладками набора свойств, например, поместить на вкладки битовые изображения. Тогда можно воспользоваться данной функцией, чтобы затем с помощью функций класса CTabCtrl выполнить необходимые операции. Описание класса CTabCtrl приведено ниже.

Кроме того, класс CPropertySheet содержит функцию:

BOOL CPropertySheet::**PressButton**(int nButton) —

моделирует нажатие желаемой кнопки в наборе свойств. nButton — задает, нажатие какой кнопки нужно смоделировать. Может принимать одно из следующих значений:

PSBTN_BACK	Выбрана кнопка **Back** (назад)
PSBTN_NEXT	Выбрана кнопка **Next** (далее)
PSBTN_FINISH	Выбрана кнопка **Finish** (готово)
PSBTN_OK	Выбрана кнопка **OK**
PSBTN_APPLYNOW	Выбрана кнопка **Apply** (применить)
PSBTN_CANCEL	Выбрана кнопка **Cancel** (отмена)
PSBTN_HELP	Выбрана кнопка **Help** (справка)

Первые три кнопки применяются в режиме мастера.

Для завершения работы набора свойств служит функция:

void CPropertySheet::**EndDialog**(int nEndID) —

параметр nEndID задает значение, которое вернет функция DoModal как результат работы набора свойств.

Эта функция вызывается библиотекой, когда нажаты кнопки **OK**, **Cancel** или окно набора свойств закрывается иным способом — нажатием клавиши <Esc> или нажатием кнопки закрытия окна в правой части заголовка. А также в случае, если будет нажата кнопка **Close** (закрыть). Самим эту функцию можно вызывать в том случае, когда происходит событие, требующее закрытия набора свойств.

Примечание

Кнопка **Close** может появиться вместо кнопки **OK**. Более подробную информацию найдете в описании функции CPropertyPage::CancelToClose ниже.

Изменение параметров отображения

В заключение описания класса CPropertySheet приведем пример изменения стилей и параметров отображения набора свойств. На рис. 28.4 представлен набор свойств с установленными параметрами вывода по умолчанию.

Рис. 28.4. Набор свойств со значениями параметров вывода, заданными по умолчанию

Для того чтобы изменить некоторые флаги (параметры вывода), следует изменить конструктор класса, базирующегося на классе CPropertySheet. Если набор свойств был добавлен в качестве программного компонента, то такой класс будет создан в обязательном порядке. В противном случае аналогичные действия следует провести сразу же после создания объекта в набор свойств, перед вызовом функции CPropertySheet::DoModal или CPropertySheet::Create.

```
CPropSheet::CPropSheet(CWnd* pWndParent)
  : CPropertySheet(IDS_PROPSHT_CAPTION, pWndParent){
```

```
  m_psh.dwFlags |= PSH_PROPTITLE;
  ...
}
```

Для изменения способа отображения вкладок следует перегрузить функцию OnInitDialog следующим образом:

```
BOOL CPropSheet::OnInitDialog()
{
  BOOL bResult = CPropertySheet::OnInitDialog ();
  CWnd::ModifyStyle(GetTabControl()->m_hWnd,
    TCS_MULTILINE, TCS_SINGLELINE, 0);
  return bResult;
}
```

Как видите, в данном случае не обойтись без создания своего собственного класса. Однако есть способ более правильный, ввденный в последней версии, т. к. изменение стиля вывода вкладок осуществляется кодом библиотеки классов MFC.

void CPropertySheet::**EnableStackedTabs**(BOOL bStacked) —

позволяет изменить стиль вывода вкладок набора свойств. Если параметр bStacked равен FALSE, то вкладки располагаются в несколько рядов, а если — TRUE, то в одну строку с возможностью их прокрутки.

Эту функцию следует вызывать при создании набора свойств, но до вывода его на экран.

Рис. 28.5. Набор свойств после изменения некоторых значений параметров вывода

```
BOOL CPropSheet::OnInitDialog()
{
  // Устанавливаем стиль вывода вкладок в одну линию
  EnableStackedTabs(TRUE);
```

```
  BOOL bResult = CPropertySheet::OnInitDialog ();
  return bResult;
}
```

Результат использования этой функции приведен на рис. 28.5.

В последней версии библиотеки MFC добавлен новый класс набора свойств — CPropertySheetEx, к рассмотрению которого мы и переходим.

Класс CPropertySheetEx

Класс CPropertySheetEx предоставляет поддержку для расширенной структуры PROPSHEETHEADER, введенной в Windows 98 и Windows NT. Эта структура была описана раньше, и вы, конечно же, обратили внимание на дополнительные поля и соответствующие флаги для поддержки битового массива для отображения фона.

В классе всего три функции — конструктор, Construct и AddPage. По своей структуре функция AddPage ничем не отличается от аналогичной базового класса, а вот функции создания объекта мы рассмотрим подробно.

CPropertySheetEx::**CPropertySheetEx**( ),

CPropertySheetEx::**CPropertySheetEx**(UINT nIDCaption,
CWnd* pParentWnd = NULL,
UINT iSelectPage = 0, HBITMAP hbmWatermark = NULL,
HPALETTE hpalWatermark = NULL, HBITMAP hbmHeader = NULL)

и

CPropertySheetEx::**CPropertySheetEx**(LPCTSTR pszCaption,
CWnd* pParentWnd = NULL, UINT iSelectPage = 0,
HBITMAP hbmWatermark = NULL, HPALETTE hpalWatermark = NULL,
HBITMAP hbmHeader = NULL) —

nIDCaption — идентификатор заголовка набора свойств; pszCaption — указатель на строку, содержащую текст заголовка — эти параметры не могут быть NULL; pParentWnd — указатель на родительское окно (если значение этого параметра NULL, то родительским считается главное окно приложения); iSelectPage — индекс страницы, которая будет выбрана при первом выводе набора свойств на экран (по умолчанию это первая страница, добавленная в набор свойств); hbmWatermark — дескриптор битового массива для фона, а hbmHeader — дескриптор битового массива для области заголовка; hpalWatermark — дескриптор палитры набора свойств.

Аналогичные изменения коснулись и функции Construst:

void CPropertySheetEx::**Construct**(UINT nIDCaption,
CWnd* pParentWnd = NULL,
UINT iSelectPage = 0, HBITMAP hbmWatermark = NULL,
HPALETTE hpalWatermark = NULL, HBITMAP hbmHeader = NULL)

и

void CPropertySheetEx::**Construct**(LPCTSTR pszCaption,
CWnd* pParentWnd = NULL,
UINT iSelectPage = 0, HBITMAP hbmWatermark = NULL,
HPALETTE hpalWatermark = NULL, HBITMAP hbmHeader = NULL) —

имеют те же параметры, что и конструктор.

Во всем остальном этот класс ведет себя так же, как и базовый.

Вторым классом, который используется при создании набора свойств для создания включаемых в него страниц свойств, является класс CPropertyPage.

Класс CPropertyPage

Выше мы говорили, что объекты класса CPropertyPage представляют собой отдельные страницы, входящие в набор свойств. Прежде всего должен быть создан сам объект CPropertySheet, затем необходимо создать объекты класса CPropertyPage для каждой вставляемой в набор свойств страницы, а затем добавить созданные страницы в набор свойств функцией CPropertySheet::AddPage. Теперь можно вывести набор свойств на экран с помощью функций CPropertySheet::DoModal — для модального или CPropertySheet::Create — для немодального набора свойств.

Как и в случае класса CPropertySheet, для знакомства с возможными параметрами страницы свойств сначала рассмотрим переменную, в которой хранятся основные параметры страницы свойств:

```
PROPSHEETPAGE CPropertyPage::m_psp
```

Эту структуру можно использовать для изменения параметров страницы уже после ее создания, но до того, как она выведена на экран. Рассмотрим поля этой структуры:

```
typedef struct _PROPSHEETPAGE { // psp
    DWORD dwSize;
    DWORD dwFlags;
    HINSTANCE hInstance;
    union {
      LPCTSTR pszTemplate;
      LPCDLGTEMPLATE pResource;
    };
    union {
      HICON hIcon;
      LPCTSTR pszIcon;
```

```
    };
    LPCTSTR pszTitle;
    DLGPROC pfnDlgProc;
    LPARAM lParam;
    LPFNPSPCALLBACK pfnCallback;
    UINT FAR * pcRefParent;
#if (_WIN32_IE >= 0x0400)
    LPCTSTR pszHeaderTitle;
    LPCTSTR pszHeaderSubTitle;
#endif
} PROPSHEETPAGE;
```

dwSize — задает размер структуры в байтах (сюда входит также размер дополнительных определяемых приложением данных в конце структуры); dwFlags — набор флагов, определяющих, какие параметры структуры должны быть использованы, а какие — проигнорированы. Значение этого поля может быть комбинацией следующих значений:

PSP_DEFAULT	Использование значения по умолчанию для всех членов структуры
PSP_DLGINDIRECT	Использование значения поля pResource вместо значения поля pszTemplate. При этом шаблон страницы свойств не загружается как диалоговый ресурс, а создается в памяти
PSP_HASHELP	Кнопка **Help** (справка) должна быть доступна, если данная страница является активной
PSP_HIDEHEADER	Для набора свойств мастера не отображается область заголовка выбранной страницы
PSP_PREMATURE	Страница будет создаваться одновременно с набором свойств. Если этот флаг не установлен, то страница не создается, пока она не будет выбрана первый раз
PSP_USECALLBACK	Сообщает о необходимости вызова функции, заданной в поле pfnCallback, при создании и удалении страницы свойств, определяемой данной структурой
PSP_USEHEADERSUBTITLE	В области подзаголовка отображается текст, хранящийся в поле pszHeaderSubTitle. Этот флаг игнорируется, если установлен флаг PSP_HIDEHEADER, или не установлен флаг PSH_WIZARD97, в поле dwFlags структуры PROPSHEETHEADER
PSP_USEHEADERTITLE	В области подзаголовка отображается текст, хранящийся в поле pszHeaderTitle. Этот флаг игнорируется, если установлен флаг PSP_HIDEHEADER, или не установлен флаг PSH_WIZARD97, в поле dwFlags структуры PROPSHEETHEADER

PSP_USEHICON	Сообщает о необходимости использования пиктограммы, заданной дескриптором в поле hIcon, в заголовке страницы свойств
PSP_USEICONID	Сообщает о необходимости использования значка, заданного именем в поле pszIcon, в заголовке набора свойств
PSP_USEREFPARENT	Поддерживает счетчик ссылок, задаваемый полем pcRefParent
PSP_USETITLE	Сообщает о необходимости использования поля pszTitle в качестве указателя на строку заголовка страницы свойств, хранящуюся в шаблоне диалога
PSP_RTLREADING	Предписывает выводить заголовок страницы свойств для чтения справа налево (для арабского языка или иврита)

pszTemplate — шаблон блока диалога, используемого для создания страницы свойств, параметр может содержать идентификатор ресурса или строку, содержащую имя ресурса (если параметр dwFlags содержит флаг PSP_DLGINDIRECT, то содержимое данного параметра игнорируется); pResource — указатель на шаблон блока диалога в памяти (если параметр dwFlags не содержит флаг PSP_DLGINDIRECT, то содержимое данного параметра игнорируется); hIcon — дескриптор значка, предназначенного для размещения на вкладке страницы свойств (если параметр dwFlags не содержит флаг PSP_USEHICON, то содержимое данного параметра игнорируется); pszIcon — значок, предназначенный для размещения на вкладке страницы свойств, параметр может содержать либо идентификатор значка, либо указатель на строку с именем ресурса, содержащего значок (если параметр dwFlags не содержит флаг PSP_USEHICOND, то содержимое данного параметра игнорируется); pszTitle — заголовок страницы свойств, переопределяющий заголовок, заданный в шаблоне диалога, параметр может содержать либо идентификатор текстовой строки, либо указатель на строку с именем ресурса, содержащего заголовок (если параметр dwFlags не содержит флаг PSP_USETITLE, то содержимое данного параметра игнорируется); pfnDlgProc — указатель на процедуру блока диалога для данной страницы (она не должна вызывать функцию EndDialog); lParam — определяемые приложением данные; pfnCallback — указатель на определяемую приложением функцию обратного вызова, вызываемую при создании и уничтожении страницы свойств (если параметр dwFlags не содержит флаг PSP_USECALLBACK, то содержимое данного параметра игнорируется); pcRefParent — указатель на счетчик ссылок (если параметр dwFlags не содержит флаг PSP_USERREFPARENT, то содержимое данного параметра игнорируется); pszHeaderTitle — указатель на строку текста, помещаемого в область заголовка, а pszHeaderSubTitle — в область подзаголовка.

Создание страницы свойств

Класс имеет три версии конструктора:

CPropertyPage::**CPropertyPage**( ),
CPropertyPage::**CPropertyPage**(UINT nIDTemplate, UINT nIDCaption = 0)

и

CPropertyPage::**CPropertyPage**(LPCTSTR lpszTemplateName,
UINT nIDCaption = 0) —

nIDTemplate — идентификатор шаблона создаваемой страницы; nIDCaption — идентификатор строки, которая будет помещена на вкладку данной страницы (если параметр задан равным 0, то имя будет взято из шаблона диалога для данной страницы); lpszTemplateName — указатель на строку, содержащую имя шаблона диалога для данной страницы (не может быть равным NULL).

В случае, если создается массив страниц свойств, вместо конструктора используется функция Construct. В этом случае ее необходимо вызвать для каждой создаваемой страницы свойств.

void CPropertyPage::**Construct**(UINT nIDTemplate, UINT nIDCaption = 0)

и

void CPropertyPage::**Construct**(LPCTSTR lpszTemplateName,
UINT nIDCaption = 0) —

параметры вызова аналогичны параметрам конструктора CPropertyPage.

Изменение состояния

Выше уже описывалась обработка нажатия кнопки **Apply** (применить) в наборе свойств. Для этого предназначена следующая пара функций:

void CPropertyPage::**SetModified**(BOOL bChanged = TRUE) —

делает доступной или недоступной кнопку **Apply** набора свойств в зависимости от того, должны ли быть применимы параметры, задаваемые в странице свойств, к внешнему объекту. bChanged — может принимать значение TRUE, если с момента последнего применения измененных параметров страницы свойств пользователь внес новые изменения, и FALSE, если значения, заданные в странице свойств, уже применены к внешнему объекту или они должны быть проигнорированы.

Кнопка **Apply** станет доступной, если вызвана функция SetModified со значением параметра, равным TRUE; при значении FALSE кнопка станет недоступна только в том случае, если все другие страницы набора свойств не содержат измененных и не модифицированных у объекта параметров.

Следующая функция визуализирует невозможность отмены внесенных изменений:

void CPropertyPage::**CancelToClose**( ) —

заменяет надпись на кнопке **OK** на надпись **Close** (закрыть) и делает недоступной для использования кнопку **Cancel** (отмена) для предупреждения пользователя о том, что произве-

денные изменения уже не могут быть отменены. К исходному состоянию наименование кнопок возвращается после выполнения команды **Apply**. Однако если функцию CancelToClose выполнить в обработчике команды ID_APPLY_NOW, возвращения названия уже не произойдет. Последнее обстоятельство требует от разработчика хорошенько продумать место использования описываемой возможности, чтобы не запутать пользователей.

В странице свойств имеется возможность с помощью следующей функции послать сообщение страницам набора свойств и получить ответ от них:

LRESULT CPropertyPage::**QuerySiblings**(WPARAM wParam, LPARAM lParam) —

параметры wParam и lParam содержат дополнительные параметры посылаемого сообщения. Функция возвращает ненулевое значение или же 0 — если все страницы вернули 0. Если страница вернула ненулевое значение, то следующим страницам сообщение не посылается.

Переопределяемые функции

Класс CPropertyPage содержит целый ряд переопределяемых функций. Рассмотрим их по порядку.

Если страница выбрана пользователем и становится активной (получает фокус ввода), то библиотека вызывает функцию OnSetActive:

virtual BOOL CPropertyPage::**OnSetActive**( ) —

реализация по умолчанию создает окно для страницы, если оно еще не было создано и делает его активным.

Данную функцию нужно переопределить, если требуется выполнить какие-либо операции во время активизации страницы. Если данная функция переопределяется, то сначала нужно вызвать из нее реализацию функции по умолчанию, а затем выполнять требуемые действия.

Если текущая страница набора свойств становится неактивной, библиотека вызывает функцию:

virtual BOOL CPropertyPage::**OnKillActive**( ) —

реализация по умолчанию копирует данные из элементов управления страницы в соответствующие переменные набора свойств, осуществляя их проверку. Если при этом происходит ошибка, то страница не теряет фокус ввода. Функцию нужно переопределить, если необходимо осуществлять дополнительную проверку данных.

После успешного завершения функции OnKillActive библиотека вызывает функцию:

virtual void CPropertyPage::**OnOK**( ) —

вызывается при нажатии кнопок **OK**, **Apply** и **Close** сразу после вызова функции OnKillActive. Данную функцию нужно переопределить, если необходимо выполнить какие-либо действия над текущей страницей перед тем, как будет удален весь набор свойств. Реализация по умолчанию помечает данную страницу как не имеющую изменений, это означает, что данные были успешно обновлены функцией OnKillActive.

При нажатии кнопок **OK** или **Apply** вызывается функция :

virtual BOOL CPropertyPage::**OnApply**(1) —

принимаются *все* изменения для *всех* страниц набора свойств, набор свойств теряет фокус ввода, и данная функция возвращает значение TRUE. Чтобы данная функция могла быть вызвана, прежде необходимо вызвать функцию SetModified со значением параметра равным TRUE. Она делает доступной кнопку **Apply**, если пользователь произвел изменения параметров в странице свойств. Реализация данной функции по умолчанию вызывает функцию OnOK.

Переопределив данную функцию, можно дать возможность приложению выполнить требуемые действия при нажатии пользователем кнопки **Apply**. Переопределенная функция должна возвращать значение TRUE (если изменения принимаются) или FALSE (если они не принимаются).

Перегрузка данной функции для каждой страницы свойств, наверное, вряд ли имеет смысл, гораздо проще обработать команду в функции класса, производного от CPropertySheet, где можно обработать изменения сразу всех страниц. Тем более, что функция OnApply вызывается зачастую даже в том случае, если собственно на странице никаких изменений произведено не было. Во всяком случае вызов этой команды при активной второй странице приводит к вызову функций OnApply и для первой страницы, и для второй.

При нажатии кнопки **Cancel** библиотека вызывает следующую группу функций:

virtual void CPropertyPage::**OnCancel**( ) —

реализация по умолчанию отменяет все произведенные изменения. Функцию можно переопределить, если необходимо дополнительно при этом совершать какие-либо действия.

virtual void CPropertyPage::**OnReset**( ) —

изменения во всех страницах набора свойств после предыдущего нажатия пользователем кнопки **Apply** отменяются, набор свойств не теряет фокус ввода. Реализация по умолчанию не совершает никаких действий.

virtual BOOL CPropertyPage::**OnQueryCancel**( ) —

вызывается после того, как была нажата кнопка **Cancel**, но до совершения самой операции отмены. Функция должна возвращать FALSE, если нужно запретить операцию отмены, и TRUE — если нужно ее разрешить. Реализация по умолчанию возвращает TRUE.

Как мы уже неоднократно упоминали, в библиотеку классов MFC добавлены несколько новых классов, в число которых входит и CPropertyPageEx, расширяющий возможности работы со страницами свойств. Рассмотрим его.

Класс CPropertyPageEx

Новый класс библиотеки MFC, который предоставляет поддержку для расширенной структуры PROPSHEETPAGE, введенной в Windows 98 и Windows NT. Эта структура содержит дополнительные флаги и компоненты для поддержки широкой области заголовка, приспособленной для размещения как

заголовка, так и подзаголовка. Саму структуру мы рассмотрели раньше, поэтому здесь остановимся только на том новом, что появилось в этом классе.

Новых функций всего две — конструктор и функция Construct, но они существенно облегчают работу с новыми элементами страницы свойств.

CPropertyPageEx::**CPropertyPageEx**( ),
CPropertyPageEx::**CPropertyPageEx**(UINT nIDTemplate, UINT nIDCaption = 0, UINT nIDHeaderTitle = 0, UINT nIDHeaderSubTitle = 0),

и

CPropertyPageEx::**CPropertyPageEx**(LPCTSTR lpszTemplateName, UINT nIDCaption = 0, UINT nIDHeaderTitle = 0, UINT nIDHeaderSubTitle = 0) —

nIDTemplate — идентификатор или lpszTemplateName указатель на строку, содержащую имя шаблона страницы; nIDCaption — идентификатор имени страницы, располагаемого на соответствующей вкладке (если используется значение по умолчанию — 0, то имя берется из шаблона этой страницы); nIDHeaderTitle — идентификатор названия, располагаемого в заголовке страницы свойств; nIDHeaderSubTitle — идентификатор названия, располагаемого в области подзаголовка страницы свойств.

Вызов этой функции создает объект класса CPropertyPageEx. Однако отображаться на экране он будет только после выполнения следующих действий:

- Страница добавлена в набор свойств с помощью функции CPropertyPageEx::AddPage
- Вызвана одна из функций DoModal или Create набора свойств
- Пользователь выбрал эту страницу

Как и раньше в случае нескольких параметров, например, при определении необходимых параметров непосредственно во время выполнения, вместо конструктора следует использовать функцию Construct:

void CPropertyPageEx::**Construct**(UINT nIDTemplate, UINT nIDCaption = 0, UINT nIDHeaderTitle = 0, UINT nIDHeaderSubTitle = 0)

и

void CPropertyPageEx::**Construct**(LPCTSTR lpszTemplateName, UINT nIDCaption = 0, UINT nIDHeaderTitle = 0, UINT nIDHeaderSubTitle = 0) —

имеет те же параметры, что и конструктор.

Чтобы закончить с наборами свойств, рассмотрим один класс, который инкапсулирует возможности элемента управления Windows, — "вкладка".

Вкладки

Элемент управления "вкладка" (TAB) имитирует обычную вкладку в записной книжке. Когда пользователь выбирает одну из вкладок, на экран выводится соответствующая страница, которая обычно представляет собой отдельное окно или блок диалога. Можно создать вкладку, имеющую вид кнопки. В этом случае при выборе вкладки вместо появления новой страницы будет выполняться какая-либо операция. Класс CTabCtrl, входящий в библиотеку MFC, как раз и позволяет создавать вкладки, существенно упрощая работу с ними.

Чаще всего вкладки используются совместно с наборами свойств (PROPERTY SHEET). Хотя они и создаются автоматически при создании набора свойств, необходимо лишь задать текст, который должен на них появиться, вместе с тем имеется возможность получить доступ к элементу управления "вкладка" и затем воспользоваться функциями класса CTabCtrl.

Каждая вкладка может иметь надпись, состоящую из строки текста, значок и определяемое приложением 32-битовое значение. Имеется возможность создавать вкладки, внешний вид которых будет определяться пользователем.

Создание вкладок

Итак, первое, что нужно сделать, это создать вкладку. Вкладку можно создать в блоке диалога или просто в окне, не являющемся блоком диалога.

Для того чтобы использовать вкладку в блоке диалога, необходимо выполнить следующие действия:

- В редакторе ресурсов блока диалога добавить вкладку в создаваемый шаблон блока диалога, задать ее идентификатор. Кроме того, можно задать требуемые стили вкладки.
- Используя средство ClassWizard, добавить переменную типа CTabCtrl в класс блока диалога. Для этого в тот момент, когда создаваемая вкладка в редакторе ресурсов блока диалога имеет фокус ввода, нужно выбрать в главном меню пункт **View\ClassWizard**. При этом будет запущен ClassWizard, который немедленно предложит вам добавить новый класс в приложение: нужно задать имя нового класса и выбрать из списка базовых классов имя класса CTabCtrl.
- Используя средство ClassWizard, добавить функции-обработчики необходимых уведомляющих сообщений.

В функции OnInitDialog класса блока диалога при необходимости можно изменить стили вкладки.

Для создания вкладки в недиалоговом окне необходимо выполнить следующее:

- Определить вкладку в классе представления или в классе окна.
- Вызвать функцию Create для создания вкладки и задания необходимых стилей.

Поскольку вкладка изначально создается пустой, после ее создания нужно добавить в нее необходимые записи. Для добавления записи в этот элемент управления необходимо подготовить структуру типа TCITEM и вызвать функцию CTabCtrl::InsertItem, передав ей заполненную структуру. Указанные операции необходимо повторить для каждой добавляемой записи.

Когда пользователь щелкает вкладку или нажимает кнопку, элемент управления "вкладка" посылает уведомляющее сообщение родительскому окну. Можно обрабатывать эти сообщения, например, если пользователь щелкнул вкладку, то можно до вывода на экран соответствующей страницы задать значения данных, выводимых на ней.

Вкладка посылает следующие уведомления (в форме WM_NOTIFY):

NM_CLICK	Сообщает родительскому окну, что пользователь щелкнул левой кнопкой мыши внутри вкладки
NM_RCLICK	Сообщает родительскому окну, что пользователь щелкнул правой кнопкой мыши внутри вкладки
NM_REALEASEDCAPTURE	Сообщает родительскому окну, что вкладка "захватила" фокус мыши
TCN_KEYDOWN	Сообщает родительскому окну, что нажата клавиша. Параметр lParam при этом указывает на структуру NMTCKEYDOWN, поле wVKey которой содержит соответствующий виртуальный код нажатой клавиши, а flags определяет ее модификатор (Shift, Alt и т. п.)
TCN_SELCHANGE	Сообщает родительскому окну, что произошла смена текущей вкладки
TCN_SELCHANGING	Сообщает родительскому окну, что происходит смена текущей вкладки

Конструктор класса:

CTabCtrl::**CTabCtrl**( )

Для создания собственно элемента управления используется функция Create:

BOOL CTabCtrl::**Create**(DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID) —

создает элемент управления Windows "вкладка" и присоединяет ее к объекту класса CTabCtrl. Параметр rect — задает расположение и размеры вкладки, pParentWnd — задает родительское окно, параметр nID — идентификатор вкладки, а dwStyle — стиль вкладки, и может принимать следующие значения:

TCS_BUTTONS	Элементы вкладки выглядят как кнопки
TCS_FIXEDWIDTH	Все элементы вкладки при создании имеют одинаковую ширину. По умолчанию их ширина автоматически изменяется в соответствии с размерами надписей и значков. Этот стиль нельзя использовать совместно со стилем TCS_RIGHTJUSTIFY
TCS_FOCUSNEVER	Элемент вкладки никогда не получит фокус ввода
TCS_FOCUSONBUTTONDOWN	Задает, что элемент вкладки получит фокус ввода только при щелчке мышью над ним. Этот стиль обычно применяется, только если указан стиль TCS_BUTTONS
TCS_FORCEICONLEFT	Расположит значок слева, а надпись оставит по центру элемента вкладки. По умолчанию значок и надпись располагаются по центру элемента вкладки, причем значок располагается слева от надписи
TCS_FORCELABELLEFT	Значок и надпись выравниваются по левому краю
TCS_MULTILINE	Элементы вкладки выводятся в несколько строк. По умолчанию они выводятся в одну строку и могут перелистываться с помощью клавиш со стрелками
TCS_OWNERDRAWFIXED	За отрисовку элемента управления отвечает разработчик
TCS_RIGHTJUSTIFY	Выравнивает элементы вкладки по правому краю. По умолчанию они выравниваются по левому краю в каждой строке
TCS_SHAREIMAGELISTS	Указывает, что список изображений вкладки не должен уничтожаться при ее уничтожении
TCS_TOOLTIPS	Вкладки имеют связанные с ними всплывающие подсказки
TCS_TABS	Элементы вкладки изображаются на экране в виде вкладок, обрамленных рамкой. Этот стиль задан по умолчанию
TCS_SINGLELINE	Элементы вкладки выводятся только в одну строку. Если необходимо, пользователь может их пролистывать. Этот стиль задан по умолчанию
TCS_RAGGEDRIGHT	Строка элементов вкладки не растягивается для заполнения всей ширины вкладки. Стиль задан по умолчанию

Как уже говорилось, изначально вкладка создается пустой. Для вставки в нее нового элемента предназначена функция:

BOOL CTabCtrl::**InsertItem**(int nItem, TC_ITEM* pTabCtrlItem),
BOOL CTabCtrl::**InsertItem**(int nItem, LPCTSTR lpszItem),
BOOL CTabCtrl::**InsertItem**(int nItem, LPCTSTR lpszItem, int nImage)

и

BOOL CTabCtrl::**InsertItem**(UINT nMask, int nItem,
LPCTSTR lpszItem, int nImage, LPARAM lParam) —

nItem — индекс нового элемента вкладки, pTabCtrlItem — указатель на структуру TCITEM, содержащую атрибуты нового элемента. В случае успешного завершения функция воз-

вращает индекс вставленного элемента, в случае неудачи — значение –1. Три последние версии функции позволяют создать вкладку только с некоторыми задаваемыми значениями — остальные устанавливаются по умолчанию.

Примечание

Остается только догадываться, почему возвращаемое значение этой функции имеет тип BOOL. Как ни странно, но в Windows тип BOOL определен как int — typedef int BOOL, так что функция действительно может вернуть и индекс, и –1.

BOOL CTabCtrl::**DeleteItem**(int nItem) —

удаляет элемент с индексом nItem из вкладки.

BOOL CTabCtrl::**DeleteAllItems**( ) —

удаляет сразу все элементы вкладки.

В случае успешного удаления обе эти функции возвращают значение TRUE.

Вспомогательные функции

Следующая функция преобразует экранные координаты в координаты окна и обратно:

void CTabCtrl::**AdjustRect**(BOOL bLarger, LPRECT lpRect) —

bLarger — указывает, какая операция производится: если параметр равен TRUE, то параметр lpRect содержит экранные координаты и получает соответствующие значения в координатах окна, а если FALSE, то наоборот.

Для удаления заданного изображения из списка изображений вкладки служит функция:

void CTabCtrl::**RemoveImage**(int nImage) —

параметр nImage задает индекс изображения, которое должно быть удалено. Вкладка обновляет индексы изображений каждого элемента вкладки так, что каждый элемент остается связанным с тем же самым изображением, что и раньше, до удаления изображения из списка.

Кроме описанных выше функций, класс CTabCtrl имеет переопределяемую функцию, которая вызывается для отрисовки самоотображающихся элементов:

void CTabCtrl::**DrawItem**(LPDRAWITEMSTRUCT lpDrawItemStruct)

Пример использования структуры DRAWITEMSTRUCT приведен в разделе, посвященном классу CListBox.

Далее приведем обзор функций класса, предназначенных для получения и изменения атрибутов вкладки. Для получения и изменения списка изображений, ассоциированных с вкладкой, служит следующая пара функций:

HIMAGELIST CTabCtrl::**GetImageList**( ) —

возвращает дескриптор списка изображений или NULL в случае его отсутствия.

CImageList* CTabCtrl::**SetImageList**(CImageList* pImageList) —

устанавливает новый дескриптор списка изображений, задаваемый параметром pImageList и возвращает дескриптор ранее установленного списка.

Изменение вкладок

Если необходимо получить число элементов вкладки, то в этом поможет функция:

int CTabCtrl::**GetItemCount**( )

Для получения информации об элементе вкладки предназначена функция:

BOOL CTabCtrl::**GetItem**(int nItem, TCITEM* pTabCtrlItem)

А для ее изменения — функция:

BOOL CTabCtrl::**SetItem**(int nItem, TCITEM* pTabCtrlItem) —

nItem — индекс элемента в вкладке; pTabCtrlItem — указатель на структуру типа TCITEM, в которую записывается информация об элементе (функция GetItem) или в которой содержится информация для изменения (функция SetItem).

Структура TCITEM имеет следующий вид:

```
typedef struct _TCITEM {
  UINT mask;
#if (_WIN32_IE >= 0x0300)
  DWORD dwState;
  DWORD dwStateMask;
#else
  UINT lpReserved1;
  UINT lpReserved2;
#endif
  LPSTR pszText;
  int cchTextMax;
  int iImage;
  LPARAM lParam;
} TCITEM;
```

mask — значение, определяющее, какие переменные структуры должны быть возвращены или установлены. Параметр может быть равен TCIF_ALL (это означает, что все поля структуры содержат действительные значения) или нулю, или быть комбинацией следующих значений:

TCIF_TEXT	Переменная pszText содержит данные
TCIF_IMAGE	Переменная iImage содержит данные

TCIF_PARAM	Переменная lParam содержит данные
TCIF_RTLREADING	Текст, на который указывает pszText, изображается справа налево
TCIF_STATE	Переменная dwState содержит данные

dwState — определяет текущее состояние элемента и может принимать следующие значения:

TCIS_BUTTONPRESSED	Элемент выбран
TCIS_HIGHLIGHTED	Элемент выделен и отображается с использованием текущего цвета выделения

dwStateMask — определяет задействованные биты параметра dwState; pszText — указатель на оканчивающуюся нулем строку, содержащую текст для нанесения на элемент вкладки. Если же структура получает информацию об элементе вкладки, то это указатель на буфер, в который будет помещена тестовая строка элемента вкладки; cchTextMax — если структура получает информацию, то в этой переменной содержится размер буфера, на который указывает pszText; iImage — индекс в списке изображений вкладки или –1, если изображение для элемента вкладки не задано; lParam — определяемое приложением значение, ассоциированное с элементом вкладки. Если оно имеет размер больше, чем четыре байта, то необходимо определить свою структуру вместо TCITEM. В таком случае первый член новой структуры должен быть структурой типа TCITEMHEADER, которая аналогична структуре TC_ITEM, но не содержит переменной lParam. Эта структура должна иметь длину, большую длины структуры TCITEMHEADER на число байтов, равное числу дополнительных байтов на каждый элемент вкладки.

Манипулировать текущим состоянием элемента можно с помощью новых функций класса GetItemState и SetItemState.

Помимо обычных стилей вкладка теперь поддерживает и дополнительные, манипулировать которыми можно с помощью функций:

DWORD CTabCtrl::**GetExtendedStyle**( )

и

DWORD CTabCtrl::**SetExtendedStyle**(DWORD dwNewStyle, DWORD dwExMask = 0) —

dwNewStyle — дополнительные стили элемента управления:

TCS_EX_FLATSEPARATORS	Между элементами располагаются разделители. Этот дополнительный стиль имеет силу только тогда, когда для элементов установлены стили TCS_BUTTON или TCS_FLATBUTTON (используется по умолчанию)
TCS_EX_REGISTERDROP	При переносе объекта через вкладку элемент управления генерирует уведомление TCN_GETOBJECT

Функция SetExtendedStyle возвращает предыдущее значение дополнительных стилей.

BOOL CTabCtrl::**GetItemRect**(int nItem, LPRECT lpRect) —

возвращает размеры прямоугольника, ограничивающего заданную запись. Параметр nItem содержит индекс записи, а lpRect — указатель на структуру RECT, получающую информацию об ограничивающем прямоугольнике.

Для того чтобы определить текущий выбранный элемент вкладки или установить выбранным элемент вкладки, предназначена следующая пара функций:

int CTabCtrl::**GetCurSel**( ) —

возвращает индекс текущего выбранного элемента вкладки или –1, если не выбран ни один элемент.

int CTabCtrl::**SetCurSel**(int nItem) —

устанавливает в качестве выбранного (текущего) элемента вкладки элемент с индексом, заданным в nItem, и возвращает индекс ранее выбранного элемента вкладки. При вызове этой функции не посылаются уведомляющие сообщения TCN_SELCHANGING или TCN_SELCHANGE, хотя и происходит изменение выбора элемента вкладки (при выборе элемента вкладки пользователем с помощью клавиатуры или мыши эти уведомляющие сообщения посылаются с помощью сообщения WM_NOTIFY).

Для изменения высоты и ширины элемента вкладки предназначена функция:

CSize CTabCtrl::**SetItemSize**(CSize size) —

size — параметр задает новую высоту и ширину элемента вкладки в пикселях. Функция возвращает установленные ранее значения высоты и ширины вкладки.

Можно установить предельное значение ширины вкладок

int CTabCtrl::**SetMinTabWidth**(int cx) —

cx — минимальное значение ширины вкладки; если этот параметр равен –1, то используется значение по умолчанию. Функция возвращает предыдущее значение минимальной ширины элемента владки.

Для изменения размеров полей вокруг каждого значка и надписи элемента вкладки предназначена функция:

void CTabCtrl::**SetPadding**(CSize size) —

size — задает размеры полей.

Следующая функция возвращает количество строк в вкладке:

int CTabCtrl::**GetRowCount**( )

Примечание

Напомним, что элементы вкладки могут располагаться не только в одну, но и в несколько строк, если вкладка имеет стиль TCS_MULTILINE.

Если вкладка создана со стилем TCS_TOOLTIPS, то она может иметь всплывающую подсказку (TOOL TIP), и функция GetTooltips возвращает дескриптор элемента управления "всплывающая подсказка":

CWnd* CTabCtrl::**GetTooltips**( )

Если же вкладка была создана без этого стиля, то, используя функцию SetTooltips, можно установить всплывающие подсказки для данной вкладки:

void CTabCtrl::**SetTooltips**(CTooltipCtrl* pWndTip) —

параметр pWndTip задает элемент всплывающей подсказки.

Когда необходимо получить индекс элемента вкладки, имеющего фокус ввода, можно воспользоваться функцией:

int CTabCtrl::**GetCurFocus**( ) —

возвращает индекс элемента вкладки, имеющего фокус ввода.

Можно также программно выбрать вкладку, установив на нее фокус

void CTabCtrl::**SetCurFocus**(int nItem)

или подсветить ее:

BOOL CTabCtrl::**HighlightItem**(int idItem, BOOL fHighlight = TRUE) —

nItem — индекс выбираемого или подсвечиваемого элемента; fHighlight — задает состояние элемента: подсвеченное (TRUE) или не подсвеченное (FALSE).

Мастера

Мастер (Wizard) представляет собой набор свойств с последовательным набором страниц, которые пользователь может переключать (перемещение при этом идет на одну страницу вперед или назад). Мастера широко применяются в программах, например, для установки нового оборудования или создания факсимильного сообщения. Набор свойств в режиме мастера не имеет вкладок в верхней части страниц, и на экран в текущий момент времени выводится только одна страница свойств.

Рис. 28.6. Типичное окно мастера

В отличие от обычного набора свойств, мастер не имеет кнопок **OK** и **Apply** (применить), а содержит кнопки **Back** (назад), **Next** (далее) (рис. 28.6) или **Finish** (готово), **Cancel** (отмена) и **Help** (справка):

Создание мастеров

Для создания мастера нужно следовать шагам создания обычного набора свойств, но прежде чем вызывать функцию CPropertySheet::DoModal, следует вызвать функцию WizardMode:

void CPropertySheet::**WizardMode**( ) —

переводит создаваемый набор свойств в режим мастера. В этом случае при завершении работы пользователя с мастером функция CPropertySheet::DoModal вернет значение ID_WIZFINISH — если пользователь закрыл мастер, нажав кнопку **Finish**, или IDCANCEL — если была нажата кнопка **Cancel**.

Функция SetWizardMode устанавливает флаг PSF_WIZARD.

Для того чтобы были доступны требуемые кнопки мастера, нужно вызвать функцию:

void CPropertySheet::**SetWizardButtons**(DWORD dwFlags) —

параметр dwFlags может содержать комбинацию следующих значений, задающих кнопки, которые должны быть доступны:

PSWIZB_BACK	Кнопка **Back**
PSWIZB_NEXT	Кнопка **Next**
PSWIZB_FINISH	Кнопка **Finish**
PSWIZB_DISABLEDFINISH	Кнопка **Finish** недоступна

Данную функцию можно вызвать только после того, как было открыто диалоговое окно мастера. Ее невозможно вызвать раньше функции CPropertySheet::DoModal. Обычно ее вызывают из функции CPropertyPage::OnSetActive.

Для того чтобы сделать доступной кнопку **Finish** или скрыть кнопки **Back** и **Next** в тот момент, когда пользователь, работая с мастером, достигнет последней страницы, нужно вызвать функцию SetFinishText:

void CPropertySheet::**SetFinishText**(LPCTSTR lpszText) —

помещает на кнопке **Finish** текст, заданный параметром lpszText, и скрывает кнопки **Back** и **Next**, когда пользователь завершает действия на последней странице мастера.

Переопределяемые функции

Класс CPropertyPage содержит следующие переопределяемые функции, используемые при создании мастера:

virtual LRESULT CPropertyPage::**OnWizardBack**( ) —

вызывается, если нажата кнопка **Back**.

virtual LRESULT CPropertyPage::**OnWizardNext**( ) —

вызывается, если нажата клавиша **Next**.

Эти функции возвращают 0, если должен быть осуществлен переход соответственно на одну страницу назад или вперед, или −1, что означает невозможность перехода к другой странице.

Если необходим переход более чем на одну страницу, то возвращается соответствующий идентификатор диалогового окна. Другим способом перехода к нужной странице является возможность задания в качестве текущей той, которая находится перед или после нужной, при помощи функции CPropertySheet::SetActivePage.

virtual BOOL CPropertyPage::**OnWizardFinish**( ) —

вызывается библиотекой, когда пользователь нажал кнопку **Finish**. Когда библиотека вызывает данную функцию и она возвращает значение TRUE, то набор свойств не уничтожается. Данную функцию можно переопределить, если необходимо выполнить действия при нажатии пользователем кнопки **Finish**. При этом функция должна вернуть значение TRUE для того, чтобы набор свойств не был уничтожен.

Функции можно переопределить, если необходимо выполнить какие-либо дополнительные операции.

Вот собственно и все, что мы хотели рассказать о наборах свойств и мастерах.

ПОДДЕРЖКА МНОГОЗАДАЧНОСТИ

Глава 29

Вытесняющая многозадачность

Что бы там ни говорили, но фундаментальное отличие 32-разрядной платформы Windows — это вытесняющая многозадачность. Надеемся, с нами согласятся те, кто работал не только на персональных компьютерах, или хотя бы в общих чертах знакомы с теорией операционных систем и историей их развития. Начнем с некоторых общих положений, чтобы непосвященным было понятнее, о чем идет речь.

Общие положения

Поддержка центральным процессором многозадачности заключается в наличии средств сохранения и загрузки контекста процесса. Отсюда следует, что процессор подразумевает наличие практически несвязанных участков кода для каждого процесса, в том числе и для ядра операционной системы. В Win32 появились две новые концепции, определяемые терминами процесс и поток:

- *Процесс* (process) — это экземпляр программы, загруженной на выполнение. С программой же связан термин *модуль*, который представлен файлом, записанным на носитель и содержащим все, что необходимо для выполнения, — код, данные, ресурсы и т. д. Каждый процесс имеет хотя бы один поток.
- *Поток* — это "путь" (thread — "нить" в дословном переводе), по которому проходит система, выполняя код процесса. При запуске приложения система создает и процесс и поток, т. к. каждый процесс имеет хотя бы один поток. Распределение процессорного времени происходит как раз между потоками, но не между процессами.

Основной поток каждого процесса создается в момент запуска приложения. Этот поток (для разработчика) начинается в момент вызова функции main или WinMain. Однако процесс может иметь несколько потоков одновременно, причем их количество может изменяться — одни потоки создаются, дру-

гие завершаются. Завершение основного потока приводит к завершению процесса независимо от того, существуют ли другие потоки. Дополнительные потоки необходимы для выполнения вспомогательных действий или действий, на которые не должны оказывать влияние пользователи.

Распределение процессорного времени (вот где многозадачность!) происходит как раз между потоками, но не между процессами. Давайте посмотрим, в каких состояниях может находиться поток (рис. 29.1). Во-первых, он может выполняться, во-вторых, он может находиться в состоянии ожидания того момента, когда ему будет опять выделен процессор. Такое состояние называется состоянием готовности к выполнению.

Рис. 29.1. Диаграмма состояний потока

Вы спросите, зачем про это даже упоминать — и так все ясно. Если бы все было так просто, серьезные программисты опять ждали бы с нетерпением новой версии Windows или стали работать в ОС UNIX. Каждый поток может быть еще и заблокированным. Наличие этого состояния существенно расширяет возможности Windows и круг задач, решение которых теперь доступно в рамках этой операционной системы. В состоянии блокировки поток находится тогда, когда он ждет возникновения какого-либо события, будь то завершение записи в файл или синхронизирующее событие от другого потока, но об этом немного позже. Для реализации такой модели многозадачности требуются очередь готовых к выполнению потоков, упорядоченная некоторым образом, и список заблокированных потоков.

Итак, на три состояния у нас имеется четыре перехода (рис. 29.1). Изначально поток попадает в очередь готовых к выполнению потоков. Через какое-то время операционная система выделяет в его распоряжение центральный процессор, где он может выполняться до истечения кванта времени. В этом случае система переведет поток в состояние готовности, поместив в очередь готовых на выполнение потоков. Если во время выполнения поток инициирует ожидание возникновения некоторого события, то операционная система заблокирует поток и освободит процессор для первого потока из очереди готовых. При возникновении ожидаемого события система переведет поток из состояния блокировки в состояние готовности. Кстати говоря, запрос на следующее сообщение (вызов функции GetMessage) как раз и блокирует поток в ожидании его прихода.

Примечание

Квантом называется промежуток времени, выделяемый операционной системой для каждой единицы выполнения, в нашем случае — для потока. Величина кванта может быть фиксированной или настраиваемой. Слишком малое значение кванта приводит к увеличению накладных расходов, затрачиваемых операционной системой на переключение. Слишком большое значение приводит к значительным задержкам реакции на внешние события или сообщения.

Наличие состояния блокировки существенно разгружает систему. Поясним этот тезис на примере. Пусть процесс состоит из двух потоков, один из которых считывает данные из файла в память, а другой будет обрабатывать эти данные по окончании чтения. Предположим, что и тому и другому потоку доступна переменная, нулевое значение которой говорит о том, что данные еще не считаны, а единичное — что операция чтения завершена. При этом код второго потока будет выглядеть примерно так:

```
while (чтение не закончено == 0);
...
```

Реализация такого подхода называется *активным ожиданием*. С помощью утилиты System Monitor (Системный монитор), входящей в состав Windows, можно увидеть, что процессор в момент выполнения этого цикла занят на 100%. Если же поток заблокирует себя в ожидании завершения процесса чтения, то он вообще не будет использовать центральный процессор, т. е. накладные расходы будут минимальны.

Важный момент в реализации многозадачности — организация очереди готовых к выполнению потоков. Простейший способ — так называемый принцип FIFO (первый пришедший обслуживается первым). Реализация именно этого принципа существенно ограничила бы возможности Windows. В Win32 многозадачность приоритетная, т. е. в очередь готовых потоки попадают в соответствии с приоритетами. При появлении потока с более высоким приоритетом, чем у выполняющегося, последний немедленно освобождает процессор и заменяется этим высокоприоритетным потоком.

Что же будет с потоками, имеющими более низкий приоритет, — они так и не получат процессор для работы? В принципе — да. Такая ситуация называется *перегрузкой* (starvation). Вот тут как нельзя кстати механизм блокировки, который исключает перегрузку. Но все же, если создаваемому приложению требуются потоки с более высоким приоритетом, чем нормальный (об этом чуть ниже), следует очень тщательно отнестись к его написанию. Но давайте обо всем по порядку.

Классы процессов и приоритеты потоков

Системы Windows поддерживают 32 приоритета (от 0 до 31), чем больше номер, тем выше приоритет. Интересно, что Windows назначает приоритеты

не только потокам, чего и следовало бы ожидать, но и процессам, для которых правильнее будет говорить о *классах*. Класс процесса определяет базовый приоритет для всех создаваемых потоков. Таких классов четыре: Idle, Normal, High, Realtime. Для установки класса процесса можно использовать, например, функцию API ::SetPriorityClass:

BOOL ::**SetPriorityClass**(HANDLE hProcess, DWORD dwPriorityClass) —

hProcess — дескриптор процесса; dwPriorityClass — класс процесса.

В табл. 29.1 перечислены классы и соответствующие им значения второго параметра функции SetPriorityClass.

Таблица 29.1. *Классы процессов*

Класс	Базовый уровень приоритета	Значение параметра
Idle	4	IDLE_PRIORITY_CLASS
Normal	7–9	NORMAL_PRIORITY_CLASS
High	13	HIGH_PRIORITY_CLASS
Realtime	24	REALTIME_PRIORITY_CLASS

Рассмотрим краткую характеристику классов.

Класс Idle

По умолчанию потоки процесса класса Idle создаются с приоритетом 4. Такие потоки хороши для задач, которым процессор требуется время от времени, например для мониторинга системы, фоновой печати или для каких-нибудь несрочных действий.

Класс Normal

Все пользовательские процессы по умолчанию относятся именно к этому классу, если только порождающий процесс не относится к классу Idle. В этом случае новый процесс также получает класс приоритета Idle.

В табл. 29.1 базовый уровень приоритета для класса Normal указан в промежутке от 7 до 9. Конкретное значение зависит от ряда причин, указанных ниже. Для простоты будем считать, что базовый уровень процессов этого класса равен 8.

Процессы этого класса самые многочисленные, и система относится к ним несколько иначе, чем к процессам всех других классов. В соответствии с терминологией, принятой в теории операционных систем, такие процессы возможно назвать *интерактивными*, т. е. их работа практически постоянно требует действий пользователя, например, в графических или текстовых ре-

дакторах. Процессы класса Normal делятся на процессы *переднего плана* (foreground) и *фоновые* (background). И Windows 95/98, и Windows NT различают эти два типа процессов: в момент перевода процесса на передний план приоритеты всех его потоков увеличиваются на 1 в Windows 95/98 или на 2 в Windows NT. Сделано это для того, чтобы активный в данный момент процесс быстрее реагировал на действия пользователя. Это один из примеров динамического изменения приоритетов.

Класс High

Создавать процессы, относящиеся к классу High и базовый уровень приоритета которых равен 13, следует с большой осторожностью. Обычно это процессы, которые большую часть времени ожидают какого-либо события. В Windows NT таким процессом является Task Manager, ожидающий нажатия клавиш <Ctrl>+<Esc>, чтобы активизироваться и позволить, например, исключить зависший процесс из списка выполняющихся. Естественно, что класс процесса программы Task Manager и определяется ее назначением.

Класс Realtime

Базовый уровень приоритета процессов класса Realtime равен 24. Если процессы предыдущего класса рекомендуется создавать только при большой необходимости, то процессы класса Realtime вообще практически никогда не стоит создавать, кроме двух случаев: во-первых, процессами этого класса могут быть лишь программы, напрямую общающиеся с аппаратурой, во-вторых, если приложение выполняет операцию, которую ни в коем случае нельзя прерывать. Что касается последнего, то таких случаев не так уж и много.

Относительные приоритеты потоков

По умолчанию создаваемый поток получает фактический приоритет в соответствии с классом своего процесса. После создания потока его приоритет может изменять не только операционная система, но и приложение. В то же время можно назначать только относительный приоритет потока, задаваемый в соответствии с классом процесса. Для установки относительного приоритета потока можно использовать, например, функцию SetThreadPriority.

BOOL ::**SetThreadPriority**(HANDLE hThread, int nPriority) —

hThread — дескриптор потока; nPriority — относительный приоритет потока.

В табл. 29.2 перечислены возможные значения параметра nPriority функции SetThreadPriority.

Таблица 29.2. *Относительные приоритеты потоков*

Относительный приоритет потока	Значение параметра	Описание
Idle (простаивающий)	THREAD_PRIORITY_IDLE	Приоритет потока равен 1 для классов Idle, Normal и High и 16 для процессов Realtime
Lowest (самый низкий)	THREAD_PRIORITY_LOWEST	Приоритет потока на 2 меньше базового уровня приоритета класса процесса
Below normal (ниже нормального)	THREAD_PRIORITY_BELOW_NORMAL	Приоритет потока на 1 меньше базового уровня приоритета класса процесса
Normal (нормальный)	THREAD_PRIORITY_NORMAL	Приоритет потока равен базовому уровню приоритета класса процесса
Above normal (выше нормального)	THREAD_PRIORITY_ABOVE_NORMAL	Приоритет потока на 1 больше базового уровня приоритета класса процесса
Highest (наивысший)	THREAD_PRIORITY_HIGHEST	Приоритет потока на 2 больше базового уровня приоритета класса процесса
Time critical (критичный по времени)	THREAD_PRIORITY_TIME_CRITICAL	Приоритет потока равен 15 для классов Idle, Normal и High и 31 для процессов Realtime

Примечание

Самый низкий (нулевой) приоритет присваивается только одному системному потоку, задача которого заключается в обнулении страниц файла подкачки. Ни один другой поток не может иметь такой приоритет.

Потоки процесса, относящегося к классу Realtime, не могут иметь приоритет менее 16, а остальные — более 15.

Ранее уже упоминалось о том, что система может динамически изменять приоритеты некоторых потоков. Для уменьшения времени реакции приложения, например, на действия пользователя, операционная система может повысить текущий приоритет потока. Такой приоритет называется *динами-*

ческим приоритетом потока (thread's dynamic priority) и может принимать значения не больше 15. После нажатия клавиши приоритет потока на определенное время возрастает, чтобы приложение как можно быстрее обработало соответствующее сообщение. Затем постепенно приоритет потока вернется к своему первоначальному уровню — *базовому уровню приоритета потока* (thread's base priority level).

Приоритеты потоков процессов класса Realtime динамически никогда не меняются. Это позволяет разделять все процессы (и потоки) на обычные — с приоритетами от 0 до 15 включительно, и реального времени — с приоритетами от 16 до 31 включительно.

Давайте посмотрим, как система определяет фактический приоритет потока, комбинируя класс приоритета процесса с относительным приоритетом потока (табл. 29.3).

Таблица 29.3. *Фактический приоритет потока*

	Класс процесса					
Относительный приоритет потока	**Idle**	**Normal в фоновом режиме**	**Normal в режиме (+1)**	**Normal в режиме (+2)**	**High**	**Realtime**
Time critical (критичный по времени)	15	15	15	15	15	31
Highest (наивысший)	6	9	10	11	15	26
Above normal (выше нормального)	5	8	9	10	14	25
Normal (нормальный)	4	7	8	9	13	24
Below normal (ниже нормального)	3	6	7	8	12	23
Lowest (самый низкий)	2	5	6	7	11	22
Idle (простаивающий)	1	1	1	1	1	16

Система поддерживает два вида потоков: *интерфейсные потоки*, например, главный поток процесса, и *рабочие потоки*, реализуемые единственной функцией, завершение которой приводит и к завершению потока.

Каждый поток процесса может прекратить свое существование в трех случаях:

1. Поток самостоятельно завершается.
2. Поток процесса может быть принудительно завершен любым другим потоком этого же процесса. При определенных условиях вообще любой поток может вмешаться в работу всех потоков в системе. Необходимо лишь знать "имя" и, для Windows NT, иметь достаточно прав на это.
3. По завершении процесса все потоки, связанные с ним, будут удалены, т. е. завершены операционной системой, поэтому можно не беспокоиться, что через некоторое время работы система деградирует настолько, что придется перезагружать компьютер.

В заключение следует сказать, что большинство создаваемых потоком объектов считается собственностью процесса, которому принадлежит данный поток. Все блоки памяти и объекты GDI, созданные потоком, принадлежат процессу. Большая же часть объектов модуля User, кроме значков, курсоров и оконных классов, принадлежит потоку. Принадлежность объекта процессу в целом делает его доступным всем потокам процесса. Многозадачность системы, в свою очередь, дает возможность одновременного обращения к такому "разделяемому" объекту несколькими потоками. (Даже в однопроцессорной системе, где допустимо лишь последовательное переключение между потоками, возможна ситуация, когда один процесс частично изменяет, например, блок памяти, затем происходит переключение на другой поток, который читает этот блок памяти целиком, где части данных не соответствуют друг другу.) Далее мы коснемся вопроса синхронизации действий потоков.

Взаимодействие программ

Любая многозадачная система будет ущербна, если она позволяет только одновременное выполнение нескольких программ. Конечно и это удобно, но именно взаимодействие открывает и перед разработчиками, и перед пользователями поистине широчайшие возможности использования компьютера в различных областях. Многопотоковые системы, которыми являются Windows 95/98 и Windows NT, кроме того, просто немыслимы без возможности взаимодействия отдельных потоков и в рамках одного и того же процесса.

Для взаимодействия могут быть использованы уже имеющиеся средства, такие как механизм сообщений, DDE и OLE. Однако назначение последних двух механизмов прежде всего в организации обмена данными между приложениями, а организация взаимодействия на их основе скорее вторична, да и к тому же реализована на очень высоком уровне. Кроме того, оба механизма, особенно OLE, требуют существенных затрат ресурсов и довольно сложны в использовании, что может быть неприемлемо для ряда задач.

Далее мы рассмотрим то, что принципиально отличает Win32 API в области организации и поддержки взаимодействия программ.

Синхронизация потоков

Так как выполняются собственно не процессы, а потоки, есть смысл говорить лишь о взаимодействии и синхронизации именно потоков, принадлежащих как одному процессу, так и разным. Только в одном случае можно упомянуть о взаимодействии именно процессов — когда один процесс ожидает завершения другого. При этом дескриптор процесса является объектом синхронизации, о чем будет подробно рассказано ниже.

В этом разделе мы кратко остановимся на механизмах, позволяющих синхронизировать действия потоков (принадлежащих как одному, так и различным процессам).

Система Windows в настоящее время поддерживает широкий набор средств, обеспечивающих синхронизацию. В любой многозадачной системе встает проблема синхронизации, если система позволяет одновременный доступ к объектам. Вся теория синхронизации была разработана в 60-х—начале 70-х годов Деккером (Dekker), Дейкстра (Dijkstra) и др.

Механизм синхронизации, реализованный в Windows, достаточно прост и гибок в использовании, требует минимальных затрат ресурсов вычислительной системы и совершенно необходим в работе! Механизм синхронизации включает в себя критические секции, защищенный доступ к переменным, набор ожидающих (блокирующих) функций, объекты синхронизации, асинхронный ввод/вывод.

Критические секции

Критической секцией называется участок кода, в котором поток получает доступ к разделяемому ресурсу (обычно некоторая переменная), доступному другим потокам. Критическую секцию имеет смысл создавать, если она будет использоваться не менее двух раз — тогда, когда доступ к одному и тому же ресурсу одновременно требуется разным потокам.

Объект "критическая секция" обеспечивает синхронизацию так же, как и объект "взаимоисключение" (см. ниже), однако он может использоваться лишь в рамках одного процесса. Использование критических секций обеспечивает немного более быстрый и более эффективный механизм синхронизации. Как и взаимоисключением, объектом "критическая секция" одновременно владеет только один поток, что обеспечивает защиту разделяемого ресурса.

Win32 API поддерживает специальный тип данных для критических секций — CRITICAL_SECTION, две функции для их создания и удаления и две функции для их использования.

CRITICAL_SECTION section

Для использования критической секции достаточно создать переменную типа CRITICAL_SECTION, а перед использованием проинициализировать ее с помощью функции InitializeCriticalSection. Для запроса входа в критическую секцию поток должен вызывать функцию EnterCriticalSection, а по завершении — функцию LeaveCriticalSection. Если другой процесс находится в своем критическом участке кода, то вызов функции EnterCriticalSection вызовет блокировку потока до тех пор, пока другой процесс не покинет свой критический участок.

Важно знать, что повторные запросы на вход в критическую секцию не вызовут самоблокировку потока и достаточно однократного вызова LeaveCriticalSection для выхода из критической секции. С другой стороны, вход в критическую секцию окажет влияние только на потоки, также запросившие вход в свою критическую секцию, остальные потоки продолжают нормально выполняться.

Любой поток может, используя вызов функции DeleteCriticalSection, освободить соответствующий системный ресурс, захваченный при инициализации объекта, после чего объект "критическая секция" уже не может быть использован для синхронизации.

Ожидающие функции

Win32 API имеет целый ряд функций, позволяющих потоку блокировать себя в ожидании некоторых событий. Их общее название — *ожидающие функции* (имена всех таких функций начинаются с префикса Wait). Они переводят поток в состояние блокировки до того, как либо произойдет некоторое событие, либо истечет заданный промежуток времени (тайм-аут). Событиями выступают один или несколько синхронизирующих объектов.

В момент вызова одной из ожидающих функций проверяется состояние указанных синхронизирующих объектов. Таких состояний два: "установлен" (signaled) и, соответственно, "сброшен" (nonsignaled). Если ни один из объектов не установлен и тайм-аут не истек, поток переводится в состояние блокировки, при этом накладные расходы операционной системы, в данном случае — расходы процессорного времени на ожидание возникновения события (событий) для разблокировки потока, минимальны.

Ожидающих функций пять: WaitForSingleObject, WaitForSingleObjectEx, WaitForMultipleObjects, WaitForMultipleObjectsEx и MsgWaitForMultipleObjects. В большинстве случаев, если соответствующая функция завершается в связи с тем, что объект был установлен, он опять переводится в состояние "сброшен":

1. Счетчик семафора уменьшается на единицу, и если он станет равным нулю, семафор будет в состоянии "сброшен".
2. Объекты "взаимоисключение", автоматически сбрасываемые события и уведомления об изменении файлов переводятся в состояние "сброшен".

3. Состояние остальных объектов синхронизации не меняется.

Функции WaitForSingleObject, WaitForSingleObjectEx предназначены для ожидания установки одного синхронизирующего объекта или окончания тайм-аута.

Функции WaitForMultipleObjects, WaitForMultipleObjectsEx и MsgWaitFor MultipleObjects предназначены для ожидания установки одного или сразу всех синхронизирующих объектов, заданных массивом дескрипторов, или окончания тайм-аута. Если любая из этих функций ожидает установки сразу всех синхронизирующих объектов, то состояние указанных объектов не сбрасывается до тех пор, пока все объекты не будут установлены. Такая реализация снимает проблему возникновения *тупиковой ситуации* (deadlock), когда несколько потоков ожидают друг от друга освобождения ресурсов.

Функции WaitForSingleObjectsEx и WaitForMultipleObjectsEx отличаются от других, т. к. они могут осуществлять возобновление потока при завершении асинхронного ввода/вывода, подробно на котором мы останавливаться не будем.

Функция MsgWaitForMultipleObjects в основном аналогична функции WaitForMultipleObjects, за исключением того, что поток может возобновить выполнение при появлении в его очереди сообщения заданного типа, например, от клавиатуры или мыши.

В заключение нельзя не сказать о функции WaitCommEvent, предназначенной для осуществления обмена данными через последовательные порты. Данная функция по сути схожа с вышеперечисленными, за исключением того, что в качестве объекта синхронизации выступает сигнал от последовательного порта.

Объекты синхронизации

Итак, механизм синхронизации описан. Теперь необходимо сказать о том, какие же объекты предоставляет операционная система для синхронизации. Вот они.

- *Процессы* как синхронизирующие объекты используются для сигнализации о их завершении
- *Потоки* как синхронизирующие объекты используются для сигнализации о их завершении
- *Файлы*, коммуникационные устройства и каналы обмена и передачи данных могут выступать в качестве синхронизирующих объектов в режиме асинхронного ввода/вывода
- *Уведомления* об изменении файлов используются для обновления списков директориев, окон редакторов и т. д.

- *Консольный* ввод используется для уведомления о наличии данных во входном буфере консоли
- *Взаимоисключения* (mutexes) позволяют координировать взаимное исключение доступа к разделяемому ресурсу
- *События* (events) используются для уведомления одного или нескольких потоков о наступлении некоторого события
- *Семафоры* (semaphores) поддерживают счетчики, значения которых меняются от нуля до задаваемых максимальных значений

Из списка видно, что набор объектов синхронизации достаточно широк и разнообразен. Важность использования первых пяти объектов поясним на примере. Ожидание одним потоком завершения другого при помощи объекта "событие" слишком громоздко, к тому же установка соответствующего события перед завершением потока еще не означает, что поток завершился. Гораздо проще ожидать завершение потока, используя его дескриптор. В равной степени это относится и к процессам.

Далее подробно рассмотрим объекты синхронизации, назначение которых только в синхронизации и состоит:

- Взаимоисключения
- События
- Семафоры

Взаимоисключения, события и семафоры могут быть как безымянными, так и именованными. Первые служат для синхронизации потоков одного процесса, вторые — для синхронизации потоков различных процессов. В качестве имени используется строка символов. Доступ к именованному объекту синхронизации из другого процесса осуществляется только по имени, другого способа (например, по дескриптору) не существует.

Взаимоисключения

Объект "взаимоисключение" позволяет координировать взаимное исключение доступа к разделяемому ресурсу. Объект имеет два состояния — "установлен" и "сброшен".

Объект "взаимоисключение" установлен в момент времени, когда обеспечение доступа к разделяемому ресурсу не требуется, говорят, что такой объект *не принадлежит* ни одному потоку, и, наоборот, объект сброшен, когда один из потоков уже осуществляет доступ к нему или *владеет* объектом. Таким образом доступ разрешается, когда другие потоки уже освободили разделяемый ресурс. Как упоминалось выше, разрешение на доступ (т. е. завершение одной из ожидающих функций) одновременно сбрасывает состояние объекта, приостанавливая таким образом все последующие запросы. По завершении работы с разделяемым ресурсом следует освободить объект "взаимоисключение", т. е. опять перевести его в состояние "установлен".

Для объявления переменной, именующей объект в приложении, используется тип HANDLE, т. к. необходимо лишь место для хранения дескриптора создаваемого впоследствии объекта. Для работы с объектом "взаимоисключение" Win32 API предоставляет функции: для создания — CreateMutex, для доступа к именованному объекту — OpenMutex и для сигнализации об окончании процесса взаимоисключения — ReleaseMutex.

Для того чтобы объявить существующий объект "взаимоисключение" принадлежащим потоку, следует вызвать одну из ожидающих функций. По завершении работы с описываемым объектом достаточно освободить дескриптор, вызвав функцию CloseHandle.

Поток, которому принадлежит в данный момент объект "взаимоисключение", может и далее использовать его в ожидающих функциях, не вызывая блокировку самого себя.

События

Событие (event) применяется для уведомления одного или нескольких ожидающих (блокированных) потоков.

Существует два типа событий. Сброс событий первого типа осуществляется вручную функцией ResetEvent. События этого типа используются для уведомления нескольких потоков. События второго типа — *автоматически сбрасываемые события* — сбрасываются ожидающими функциями.

Для объявления переменной, именующей объект, в приложении используется тип HANDLE, т. к. необходимо лишь место для хранения дескриптора создаваемого впоследствии объекта. Для работы с объектом "событие" Win32 API предоставляет функцию для создания — CreateEvent, функцию для доступа к поименованному объекту — OpenEvent, две функции для установки события — SetEvent и PulseEvent и функцию для сброса события — ResetEvent. Для ожидания возникновения события используются ожидающие функции. По завершении работы с описываемым объектом достаточно освободить дескриптор, вызвав функцию CloseHandle.

Особо следует сказать о функции PulseEvent. Эта функция устанавливает событие и затем переводит его в сброшенное состояние после возобновления заблокированных потоков. Для объектов с ручным сбросом все ожидающие потоки разблокируются. Для автоматически сбрасываемых — разблокируется только один поток, даже если в состоянии ожидания или блокировки находятся несколько потоков. Если ожидающих потоков нет, функция PulseEvent сбрасывает событие и завершается.

Семафоры

Семафор (semaphore) — это объект синхронизации, реализованный в виде счетчика, принимающего значения от нулевого до задаваемого максималь-

ного. Семафоры обычно применяются для ограничения количества потоков, одновременно обращающихся за набором разделяемых ресурсов. Например, программа поддерживает пять ресурсов одного типа, которые кратковременно могут быть выделены потокам. Как только все пять ресурсов будут выделены, следующий поток, запрашивающий ресурс данного типа, будет приостановлен до высвобождения хотя бы одного ресурса.

В установленном состоянии семафор находится в том случае, если значение его счетчика больше нуля, в противном случае он "сброшен". Ожидающие функции, использующие семафор в качестве объекта синхронизации, уменьшают значение счетчика. Поток может последовательно запрашивать семафор, используя ожидающую функцию. Но использование функции WaitForMultipleObjects (или подобной), у которой в качестве массива объектов задан массив дескрипторов одного и того же семафора, уменьшит значение счетчика последнего только на единицу.

Win32 API поддерживает следующие функции для использования семафоров: CreateSemaphore — создает семафор с заданными начальным значением счетчика и максимальным значением, которое он может получить; OpenSemaphore — осуществляет доступ к именованному семафору; ReleaseSemaphore — увеличивает значение счетчика на заданную величину (значение счетчика не может быть меньше нуля и больше максимального значения); по завершении работы с описываемым объектом достаточно освободить дескриптор, вызвав функцию CloseHandle.

Но вернемся к библиотеке.

Библиотека MFC и многозадачность

Библиотека классов MFC обеспечивает разработчиков полным набором средств создания процессов и потоков. В принципе, это вполне естественно, т. к. основной класс приложения CWinApp, являясь производным от класса CWinThread, представляет основной поток процесса приложения.

В этой главе мы рассмотрим ряд классов и приемов их использования, позволяющих эффективно создавать многопотоковые приложения, которые уже сейчас активно применяются для решения самых различных задач.

Каждый поток для приложения, созданного на базе классов библиотеки MFC, представлен объектом класса CWinThread. Причем сами объекты обычно создавать нет необходимости. Вместо вызова напрямую конструктора класса CWinThread следует воспользоваться вспомогательной функцией библиотеки ::AfxBeginThread, которая самостоятельно и создаст объект класса.

Библиотека MFC определяет два типа потоков:

- *Интерфейсные потоки* (user-interface threads)
- Так называемые *рабочие потоки* (worker threads)

Потоки первого типа предоставляют возможность взаимодействия с ними при помощи механизма сообщений и имеют в своем составе функции, аналогичные тем, которые создаются для объекта-приложения, т. е. основного его потока. Потоки второго типа, напротив, сообщения обрабатывать не могут, не имея для этого соответствующих средств обслуживания очереди сообщений, и применяются обычно для реализации строго ограниченных по времени действий, например, для сохранения данных, вычисления и т. д. Интересно, что собственно Windows не делает никаких различий между этими двумя видами потоков. Просто система должна знать адрес, с которого начинается код потока. Другое дело, что для одних потоков класс CWinThread обеспечивает механизм обслуживания сообщений и всю мощь поддержки компилятора, а для других нет. Это, впрочем, не накладывает никаких ограничений на самостоятельную реализацию фантазий программиста.

Многопотоковые приложения предназначены не только для того, чтобы выполнять независимые задания одновременно. Более того, обычно потоки одного приложения так или иначе взаимосвязаны, т. к. работают с одними данными (такие данные называются разделяемыми). Понятно, что одновременный доступ к некоторой переменной в лучшем случае приведет к возникновению ошибки в одном потоке, в худшем — сбой будет общий. Именно поэтому использование нескольких потоков в рамках одного приложения заставляет учитывать особенности работы с разделяемыми данными. Далее мы рассмотрим классы, которые предоставляет библиотека MFC для реализации корректной работы многих потоков в рамках одного приложения или даже для организации взаимодействия между потоками различных процессов.

Класс и функции для поддержки многопотоковых приложений

Все, что нужно для работы с потоком, библиотека MFC группирует в классе CWinThread. Причем объекты этого класса могут представлять как интерфейсные потоки, так и рабочие.

В большинстве случаев, а для рабочих потоков — всегда, создание потока следует производить при помощи универсальной функции ::AfxBeginThread.

Для завершения потока в любой момент его выполнения библиотека предоставляет функцию ::AfxEndThread. Однако при помощи этой функции можно завершить только текущий поток, т. е. корректное завершение потока может быть осуществлено только им самим. Другое дело, что о необходимости завершения потоку можно тем или иным образом сообщить (при помощи сообщения) или просигнализировать (при помощи одного из объектов синхронизации). Извне поток можно прервать только при помощи функции Win32 API ::TerminateThread.

Классы и функции для поддержки взаимодействия потоков и процессов

Естественно, что взаимодействие потоков, как в рамках одного процесса, так и между потоками различных процессов, можно было бы осуществлять при помощи механизма сообщений. Однако это не всегда возможно даже для интерфейсных потоков, а тем более для рабочих, не имеющих средств для обработки сообщений.

В ряде случаев требуется организовать более надежное взаимодействие (синхронизацию) для реализации взаимоисключения, т. е. разграничения доступа к разделяемым объектам и данным, а также для сигнализации о возникновении тех или других событий.

Выше мы подробно описали объекты синхронизации, которые поддерживают операционные системы Windows 95/98 и Windows NT. Библиотека MFC, в свою очередь, предлагает классы-аналоги не только объектов синхронизации, но и ожидающих функций. Приведем таблицу соответствия классов библиотеки объектам Windows или функциям Win32 API.

Класс	Объект Windows или функция Win32 API	Примечание
CSyncObject		Базовый класс для всех классов синхронизирующих объектов
CCriticalSection	Критическая секция	
CMutex	Взаимоисключение	
CEvent	Событие	
CSemaphore	Семафор	
—	Процессы, потоки, файлы, коммуникационные устройства, каналы передачи данных, уведомления об изменении файлов, консоли	Естественно, что для большинства этих объектов есть соответствующие классы, однако они не могут быть использованы в качестве именно классов синхронизации
CSingleLock	::WaitForSingleObject и ::WaitForSingleObjectEx	Ожидание единичного события
CMultiLock	::WaitForMultipleObjects, ::WaitForMultipleObjectsEx и ::MsgWaitForMultipleObjects	Ожидание нескольких событий одновременно или по одному
—	::WaitCommEvent	Ожидание события (сигнала) от последовательного порта

Реализация ряда классов библиотеки исключает полный отказ от Win32 API. Так например, ожидать завершения потоков невозможно при помощи классов CSingleLock и CMultiLock. Однако в большинстве других случаев переход к использованию классов библиотеки MFC, обеспечивающих синхронизацию и взаимодействие, вполне оправдан.

Глава 30

Создание потоков

Начнем обзор функций и классов, обеспечивающих реализацию многопоточных приложений, обзором основной функции библиотеки, которая позволяет создавать потоки обоих типов — ::AfxBeginThread. Эта функция создает новый объект CWinThread и вызовом функции этого класса CreateThread создает собственно поток. То есть приложение может создавать новые потоки и таким способом.

CWinThread*::**AfxBeginThread**(CRuntimeClass* pThreadClass,
int nPriority = THREAD_PRIORITY_NORMAL,
UINT nStackSize = 0, DWORD dwCreateFlags = 0,
LPSECURITY_ATTRIBUTESlpSecurityAttrs = NULL)

и

CWinThread*::**AfxBeginThread**(AFX_THREADPROC pfnThreadProc,
LPVOID pParam, int nPriority = THREAD_PRIORITY_NORMAL,
UINT nStackSize = 0, DWORD dwCreateFlags = 0,
LPSECURITY_ATTRIBUTES lpSecurityAttrs = NULL) —

первая версия функции создает интерфейсные потоки. Параметр pThreadClass описывает класс, производный от CWinThread, например так:

```
RUNTIME_CLASS(CUIWinThread)
```

Вторая версия функции создает рабочие потоки, код которых сосредоточен в единственной функции — функции управления (controlling function). Параметр pfnThreadProc как раз и является указателем на такую функцию. Естественно, этот параметр не может быть равен NULL. Функция же должна быть описана следующим образом:

```
UINT ThreadControllingFunction(LPVOID pParam);
```

Параметр pParam задает значение, которое передается в управляющую функцию рабочего потока.

Остальные параметры обеих версий функции аналогичны по назначению. Параметр nPriority задает относительный приоритет создаваемого потока. Значения, которые может принимать этот параметр:

THREAD_PRIORITY_TIME_CRITICAL	Критичный по времени
THREAD_PRIORITY_HIGHEST	Наивысший

THREAD_PRIORITY_ABOVE_NORMAL	Выше нормального
THREAD_PRIORITY_NORMAL	Нормальный
THREAD_PRIORITY_BELOW_NORMAL	Ниже нормального
THREAD_PRIORITY_LOWEST	Самый низкий
THREAD_PRIORITY_IDLE	Простаивающий

Примечание

Обратите внимание, что приоритет потока зависит от базового приоритета, который определяется классом приоритета процесса.

Эти мнемоники определяют непоследовательные значения в Win32 API, и, следовательно, числовые значения использовать не следует. Интересно отметить: в описании этой функции сказано, что значение параметра nPriority, равное 0, задает приоритет создаваемого потока равным приоритету потока, в рамках которого и происходит создание. А поскольку значение THREAD_PRIORITY_NORMAL как раз и равно 0, этот факт может привести к неожиданным результатам. Хотя текущая реализация на самом деле создает все-таки потоки именно с заданным, включая THREAD_PRIORITY_NORMAL, приоритетом, возможно, для большей уверенности следует создавать новые потоки следующим образом:

```
// Главный поток имеет, например, наивысший относительный приоритет
// (THREAD_PRIORITY_HIGHEST) и он же является текущим
//
// Необходимо создать новый поток
// с нормальным (равным базовому) приоритетом
//
// Объект-приложение является производным от класса CWinThread,
// следовательно, можем воспользоваться функциями этого класса
// для сохранения текущего значения относительного приоритета
// и изменения его на нормальное
DWORD dwCurrentPriority = AfxGetApp()->GetThreadPriority();
AfxGetApp()->SetThreadPriority(THREAD_PRIORITY_NORMAL);
// Создаем поток
CWinThread *pThread = AfxBeginThread(
  RUNTIME_CLASS(CUIWinThread), THREAD_PRIORITY_NORMAL);
// Проверяем относительный приоритет (для наглядности)
DWORD dwPriority = pThread->GetThreadPriority();
ASSERT(dwPriority == THREAD_PRIORITY_NORMAL);
// Восстанавливаем относительный приоритет главного потока
// сохраненным ранее значением
AfxGetApp()->SetThreadPriority(dwCurrentPriority);
```

Параметр nStackSize задает размер стека (в байтах) для нового потока. Значение 0 обеспечивает создание стека того же размера, что и у создающего потока. В принципе, и в Windows 95/98 и в Windows NT размер стека настраивается (увеличивается) по мере необходимости, так что задание начального размера стека в большинстве случаев необязательно. Параметр dwCreateFlags позволяет задавать начальное состояние нового потока и может принимать одно из следующих значений:

CREATE_SUSPENDED	Поток изначально будет приостановлен со значением счетчика приостановки 1. Для запуска потока необходимо выполнить функцию ResumeThread
0	Поток начнет немедленно выполняться сразу же после создания. Правильнее сказать, что поток будет готов к выполнению, как только он окажется первым в очереди готовых потоков своего приоритета, а очереди с потоками более высоких приоритетов будут пусты

Параметр lpSecurityAttrs указывает на структуру SECURITY_ATTRIBUTES, которая содержит атрибуты защиты потока. Значение этого параметра, равное NULL, обеспечивает новый поток теми же атрибутами защиты, что и у создающего потока. Структура содержит указатель на собственно дескриптор защиты (SECURITY_DESCRIPTOR), поле lpSecurityDescriptor, доступ к которому возможен посредством специальных функций и который содержит в том числе следующую важную информацию:

- Дескриптор защиты родителя (SID)
- Дескриптор защиты группы (SID)
- Список дискретного разграничения доступа (discretionary access control list)

Примечание

Дескриптор защиты используется только в рамках Windows NT.

Кроме дескриптора защиты структура SECURITY_ATTRIBUTES в поле bInheritHandle позволяет указать, возможно ли наследование объекта, для которого структура будет использована.

Примечание

Практически любая система позволяет одному процессу создавать другие. При этом создающие процессы называются *родительскими*, а создаваемые — *дочерними*. Однако такое наименование имеет смысл только в том случае, если дочерние и родительские процессы связаны не только временем и местом создания. В Windows 95/98 и Windows NT дочерние процессы наследуют как минимум класс приоритета, а в Windows NT еще и атрибуты защиты, хотя и то и другое может назначаться по умолчанию. Но кроме "характера" хорошо бы еще от родителя получить и "счет в банке". Дочерние процессы могут наследовать

разного рода ресурсы, описываемые дескрипторами. Это действительно удобно, например, унаследовав открытый файл, с ним можно работать — читать и писать — сразу.

Проблемы возникают только в случае, если родитель прекратил свое существование. Что делать с ресурсами, первоначально выделенными родителю? Некоторые системы просто завершают вместе с родительским все его дочерние процессы. Способ простой, но явно уж очень жестокий. Тем не менее целый ряд ресурсов просто не может больше существовать после завершения процесса — он их забирает с собой "в могилу". К такого рода ресурсам следует отнести потоки — они завершаются вместе с завершением процесса, что вполне естественно. А вот открытые файлы продолжают существовать и далее до тех пор, пока не будет завершен последний дочерний процесс, наследовавший доступ к файлам. Реализуется это достаточно просто. С каждым ресурсом связан счетчик использования. При завершении процесса система уменьшает счетчики у всех ресурсов, выделенных ему во время работы, и удаляет только те из них, счетчик которых стал равен 0.

Пример создания наследуемых потоков

Следующий пример иллюстрирует создание наследуемых потоков и дочерних процессов с использованием атрибутов защиты (дескрипторы защиты в данном примере не используются). Суть примера в том, что главный родительский процесс (Parent) по команде пользователя создает наследуемый рабочий поток, затем создает дочерний процесс (Child), которому в качестве командной строки передает числовой дескриптор нового потока. Для того чтобы дочерний процесс получил доступ к любому объекту, получаемому "по наследству", он должен быть создан с флагом наследуемости (пятый параметр функции CreateProcess).

```
// Функция обработки команды создания нового потока и дочернего процесса
void CParentDlg::OnChild()
{
  // Счетчик процессов
  static int s_nCount = 0;
  // Атрибуты наследуемого потока
  SECURITY_ATTRIBUTES sa =
  {
    sizeof(SECURITY_ATTRIBUTES),
    NULL, TRUE
  };
  // Создаем новый рабочий поток
  CWinThread *pThread = AfxBeginThread(
    (AFX_THREADPROC)InheritedThread, 0,
```

```
    THREAD_PRIORITY_NORMAL, 0, 0, &sa);
  // Структуры, необходимые для создания нового процесса
  PROCESS_INFORMATION pi;
  STARTUPINFO si;
  // Заполняем структуру STARTUPINFO
  GetStartupInfo(&si);
  // Формируем командную строку для создания/запуска нового процесса
  // (объект класса CWinThread содержит дескриптор потока)
  CString csCommand;
  csCommand.Format("Child.exe %d", pThread->m_hThread);
  // Получаем указатель на сформированную строку
  LPTSTR lpCmdLine =
    csCommand.GetBuffer(csCommand.GetLength());
  // Создаем дочерний процесс
  CreateProcess(
    NULL,        // Имя модуля передадим во втором параметре
    lpCmdLine,   // Командная строка процесса
    NULL,        // Используем атрибуты защиты процесса по умолчанию
    NULL,        // Используем атрибуты защиты потока по умолчанию
    TRUE,        // Дескрипторы наследуются
    CREATE_DEFAULT_ERROR_MODE,  // Стандартная обработка ошибок
    NULL,        // Переменные окружения не нужны
    NULL,        // Текущий каталог не используется
    &si,         // Передаем копию собственной структуры STARTUPINFO
    &pi          // Получаем дескриптор и другую информацию о процессе
   );
  // Освобождаем буфер
  csCommand.ReleaseBuffer();
  // Обновляем информацию о потоках и дочерних процессах
  CString csMessage;
  csMessage.Format("Number of threads\n"
    "and child processes:\t%d", ++s_nCount);
  GetDlgItem(IDC_MSG)->SetWindowText(csMessage);
}
```

На рис. 30.1 представлены окна родительского и дочернего процессов. Дочерний процесс время от времени по сообщению от таймера запрашивает код завершения наследованного потока. Пока родительский процесс в работе — соответствующий поток существует.

Рис. 30.1. Оба процесса в работе

Код управляющей функции рабочего наследуемого потока родительского процесса следующий:

```
// Управляющая функция потока
UINT InheritedThread(LPVOID pParam)
{
  // Поток завершится только при завершении процесса —
  // самостоятельно поток не завершится никогда
  for (;;)
    Sleep(10); // Для экономии использования процессора
  return 0;
}
```

Чтобы дочерний процесс отображал состояние наследованного потока, необходимо выполнить следующий код.

```
// Функция обработки сообщения от таймера
void CChildDlg::OnTimer(UINT nIDEvent)
{
  // Получаем код возврата наследованного потока,
  // дескриптор которого хранится в переменной m_hThread
  DWORD dwExitCode;
  GetExitCodeThread(m_hThread, &dwExitCode);

  // В зависимости от кода возврата отображаем
  // соответствующую диагностику
  if (dwExitCode == STILL_ACTIVE)
    GetDlgItem(IDC_MSG)->SetWindowText("Thread is ALIVE");
  else
    GetDlgItem(IDC_MSG)->SetWindowText("Thread is DEAD");

  CDialog::OnTimer(nIDEvent);
}
```

После завершения родительского процесса, естественно, завершился и его рабочий поток, что и отображено на рис. 30.2

Рис. 30.2. Родительский процесс завершен, включая все его потоки

В завершение приведем функцию InitInstance класса приложения дочернего процесса, в которой происходит обработка командной строки, откуда процесс получает дескриптор наследуемого потока.

```
BOOL CChildApp::InitInstance()
{
  ...
  // Преобразуем содержимое командной строки в число, а затем
  // в дескриптор — это и будет дескриптор наследуемого потока
  CChildDlg dlg((HANDLE)atoi(m_lpCmdLine));
  m_pMainWnd = &dlg;
  int nResponse = dlg.DoModal();
...
  return FALSE;
}
```

Создание интерфейсных потоков

Интерфейсные потоки (user-interface threads) используются при необходимости обработки сообщений независимо от работы других потоков приложения. Изначально каждое MFC-приложение уже имеет в своем составе главный интерфейсный поток. Такой поток обычно создается при помощи средства генерации приложений AppWizard на базе класса CWinApp. Собственно состав этого класса, базирующегося в свою очередь на классе CWinThread, полностью соответствует составу любого другого класса, обеспечивающего создание пользовательского интерфейсного потока.

Вот что необходимо сделать для создания своего интерфейсного потока:

- Создать класс на базе класса CWinThread. При этом следует описать и определить этот класс с использованием макросов DECLARE_DYNCREATE и IMPLEMENT_DYNCREATE
- Перегрузить (реализовать) часть функций, некоторые в обязательном порядке, другие в зависимости от решаемых задач

Примечание

Как и большинство классов, реализуемых на базе классов библиотеки MFC, класс потока, конечно, следует создавать при помощи средства ClassWizard. Однако класс потока может быть использован не только для интерфейсных, но и для рабочих потоков, которые в случае создания своего класса на базе CWinThread не будут обрабатывать сообщения не потому, что созданный класс изначально не имеет средств для этого. Просто все необходимые действия потока следует реализовать в функциях InitInstance или Run (см. пример Graphics из следующей главы).

По большому счету это приемлемая реализация, т. к. в рамках класса гораздо лучше группировать необходимые значения вместо того, чтобы иметь большое количество глобальных или автоматических переменных. Кроме того, при такой реализации становится возможным использовать то, что уже реализовано в классе CWinThread.

Однако в этом случае при создании своего класса следует использовать другие макросы, а именно DECLARE_DYNAMIC и IMPLEMENT_DYNAMIC.

Итак, для того чтобы создать интерфейсный поток, необходимо перегрузить функцию InitInstance, в которой следует произвести необходимые инициализирующие действия. Все остальные виртуальные функции базового класса могут быть перегружены при необходимости:

- ExitInstance — вызывается по завершении работы потока, код ее завершения определяет код завершения потока
- Run — основная функция потока, где реализован стандартный цикл обслуживания сообщений
- OnIdle — вызывается в момент простоя, т. е. отсутствия сообщений в очереди сообщений потока
- PreTranslateMessage — фильтрует сообщения до их передачи функциям TranslateMessage и DispatchMessage
- ProcessWndProcException — обрабатывает ряд исключительных ситуаций, связанных с ошибками обработки некоторых сообщений

К сожалению, чтобы обслуживать сообщения от Windows, связанные, в том числе, и с действиями пользователя, недостаточно выполнить перечисленные выше действия. Необходимо обеспечить получение сообщения потоком, т. е. обеспечить путь, по которому сообщение от системы попадет в очередь сообщений потока. Но об этом несколько позже, а сейчас самое время подробнее познакомиться с классом CWinThread.

Класс CWinThread

Объект класса CWinThread в MFC-приложении представляет собой отдельный поток. Работа именно с этим классом позволяет в полной мере использовать возможности библиотеки, перегружая виртуальные функ-

ции привычным для однопотоковых приложений способом. Кроме того, только потоки, созданные при помощи класса CWinThread, имеют возможность доступа к любым функциям и классам библиотеки MFC. Связано это с тем, что локальная область потока (thread-local storage, TLS) управляется как раз объектом класса CWinThread.

Класс самого приложения CWinApp является производным от CWinThread, т. к. представляет главный поток приложения — интерфейсный поток.

Рабочие потоки, созданные при помощи функции ::AfxBeginThread, так же как и интерфейсные, являются объектами класса CWinThread, что позволяет получать доступ к элементам, содержащим дескриптор и идентификатор потока:

HANDLE CWinThread::**m_hThread**

DWORD CWinThread::**m_nThreadID**

Первый элемент содержит дескриптор потока, а второй — его идентификатор. Эти элементы класса содержат действительные значения, естественно, только если поток существует.

Примечание

Следует отметить, что далеко не все функции Win32 API, предназначенные для работы с потоками, реализованы в виде компонентов класса CWinThread. Так что достаточно часто приходится использовать дескриптор потока напрямую.

Объект класса CWinThread обычно полностью зависит от связанного с ним потока. Это значит, что объект автоматически разрушается (удаляется) при корректном завершении потока. Однако в ряде случаев бывает необходимо самостоятельно удалить объект уже после завершения потока, например для того, чтобы получить код завершения потока. Для управления методом удаления объекта класса CWinThread служит член класса m_bAutoDelete, значение которого по умолчанию равно TRUE:

BOOL CWinThread::**m_bAutoDelete**

Если значение этого элемента равно FALSE, следует самостоятельно разрушить объект класса, когда необходимость в нем отпадет.

Кроме того, и рабочие и интерфейсные потоки могут использовать функции получения и назначения относительного приоритета потока. Для этого служат следующие две функции:

int CWinThread::**GetThreadPriority**( )

и

BOOL CWinThread::**SetThreadPriority**(int nPriority) —

параметр nPriority задает новый относительный приоритет потока. (Возможные значения см. в описании функции ::AfxBeginThread.)

Естественно, что наличия объекта класса CWinThread для вызова этих функций недостаточно — их следует вызывать только после того, как поток создан.

Как уже упоминалось, библиотека MFC предоставляет еще одну возможность создания нового потока, помимо вызова функции AfxBeginThread, а именно создание объекта класса CWinThread или производного от него, а затем вызов функции CreateThread для создания собственно потока:

BOOL CWinThread::**CreateThread**(
DWORD dwCreateFlags = 0, UINT nStackSize = 0,
LPSECURITY_ATTRIBUTES lpSecurityAttrs=NULL) —

параметры этой функции аналогичны параметрам функции ::AfxBeginThread.

Примечание

Кроме конструктора по умолчанию, класс CWinThread имеет конструктор, позволяющий создавать рабочие потоки, которому передаются в качестве параметров указатель на управляющую функцию и значение параметра для нее. (Подробнее см. раздел, посвященный созданию рабочих потоков.)

Для приостановки и возобновления потока предназначены функции SuspendThread и ResumeThread соответственно:

DWORD CWinThread::**SuspendThread**( )

и

DWORD CWinThread::**ResumeThread**( ) —

возвращают предыдущее значение счетчика приостановки в случае успешного выполнения и 0xFFFFFFFF ((DWORD) –1) в противном случае. Первая функция может быть вызвана не только для работающего потока, но и для потока ранее приостановленного. В обоих случаях функция просто увеличивает счетчик приостановки — операционная система приостанавливает поток, как только значение счетчика станет равным 1.

Функция ResumeThread, в свою очередь, лишь уменьшает значение счетчика. Если результат работы этой функции равен 0, это значит, что поток и до выполнения функции не был приостановлен. Значение 1 говорит о том, что до выполнения функции поток был приостановлен, но теперь продолжил свое выполнение. Любое другое значение говорит о том, что несмотря на выполнение функции ResumeThread и, следовательно, уменьшение счетчика приостановки, поток все равно продолжает быть приостановленным.

Таким образом, для гарантированного возобновления потока следует выполнить такой код:

```
while (pThread->ResumeThread() > 1);
```

Каждый поток начинает свое выполнение с вызова функции InitInstance, которая позволяет производному классу от класса CWinThread проводить необходимую инициализацию в момент, когда поток уже существует. В конструкторе класса такие действия не всегда возможны именно по причине отсутствия собственно потока:

virtual BOOL CWinThread::**InitInstance**( ) —

в этой функции следует проводить необходимые проверки условий, невыполнение которых исключает дальнейшее функционирование потока. Так, если результат функции равен FALSE, библиотечная функция, вызвавшая функцию InitInstance, корректно завершит поток.

Для своего класса *интерфейсного* потока необходимо в обязательном порядке перегрузить эту функцию, т. к. функция базового класса возвращает значение FALSE, что исключает дальнейшее выполнение потока. Для *рабочих* потоков переопределять функцию нет необходимости, т. к. она просто не вызывается. Все, что должен делать поток, включая начальную инициализацию, должно производиться в управляющей функции.

Перед завершением функции Run, о которой скажем чуть позже, а также в случае, если функция InitInstance вернет значение FALSE, библиотекой вызывается функция ExitInstance:

virtual int CWinThread::**ExitInstance**( )

В принципе нет необходимости переписывать эту функцию в своем классе. Однако функция ExitInstance базового класса возвращает код, переданный потоку при помощи функции ::PostQuitMessage (после завершения функции Run), или первый параметр последнего (текущего) сообщения. Вот часть исходного кода класса CWinThread из библиотеки:

```
int CWinThread::ExitInstance()
{
  ASSERT_VALID(this);
  ASSERT(AfxGetApp() != this);

  int nResult = m_msgCur.wParam;  // returns the value from
                                  // PostQuitMessage
  return nResult;
}
```

Естественно, что это не всегда может устроить разработчика, тем более что функция ::PostQuitMessage может вызываться отнюдь не из созданного им кода, а самой библиотекой. Так, очень часто в качестве кода завершения используется код клавиши, нажатие которой привело к завершению. А по умолчанию считается, что любой код, отличный от 0, сообщает о возникновении ошибки. (Кстати, код завершения потока определяется при помощи функции Win32 API ::GetExitCodeThread.)

Примечание

Несмотря на то что реализация функции ExitInstance класса CWinThread по сути "ничего не делает", в документации настоятельно рекомендуется не только не вызывать самостоятельно эту функцию, но и в обязательном порядке вызы-

вать базовую в своей реализации. Последнее связано с тем, что именно в этой функции следует выполнять явное освобождение ресурсов (если необходимо). В будущем, возможно, что-нибудь подобное будет делать и библиотека MFC, включая удаление объекта класса потока (в зависимости от значения элемента m_bAutoDelete), что на сегодняшний день не реализовано — пока происходит в функции AfxEndThread.

Основная функция каждого интерфейсного потока:

virtual int CWinThread::**Run**( )

Для интерфейсных потоков надобности в ее собственной реализации нет — инициализация производится в функции InitInstance, деинициализация — в ExitInstance. Значение, которое возвращает функция, является кодом завершения потока. Библиотечная реализация функции перед завершением просто вызывает функцию ExitInstance, так что по умолчанию функция Run просто возвращает то значение, которое возвращает функция ExitInstance.

Чтобы лучше разобраться в том, что делает библиотечная реализация функции, перечислим ее основные действия:

- Обеспечивает цикл выборки сообщений, включая их трансляцию при помощи вызова функции-члена PreTranslateMessage и дальнейшую обработку при помощи функции Win32 API DispathMessage
- Завершает цикл обработки сообщений в момент прихода сообщения WM_QUIT
- В случае отсутствия сообщений в очереди сообщений выполняет действия, допустимые в режиме простаивания (функция OnIdle), при их необходимости

Перегружать функцию Run для рабочих потоков вообще нет смысла — она просто не вызывается.

virtual BOOL CWinThread::**OnIdle**(LONG lCount) —

вызывается в цикле обработки сообщений, когда очередь сообщений потока пуста. Функция, выполняемая по умолчанию, освобождает временные объекты, т. е. объекты классов, созданные на основе уже существующих объектов Windows, например, при помощи функции классов контекстов устройства или графических объектов FromHandle, и неиспользуемые динамически подключаемые библиотеки, освобождая тем самым память. В перегруженной функции следует обязательно сначала вызвать базовую версию функции с текущим значением параметра lCount. Параметр lCount увеличивается перед каждым вызовом функции, т. е. всякий раз, когда очередь сообщений приложения пуста, и сбрасывается в 0 при обработке нового сообщения. Этот параметр можно использовать для определения различных режимов обработки или относительного времени "простоя" потока.

Функция должна вернуть нулевое значение в случае, если больше нет необходимости в ее повторном выполнении до прихода следующего сообщения. В этом случае функция будет вызвана опять, как только очередь сообщений станет пуста.

Так как во время выполнения функции OnIdle поток не сможет обрабатывать сообщения, нельзя в рамках этой функции выполнять длительные по времени действия.

Следующая функция вызывается до обработки поступившего сообщения:

virtual BOOL CWinThread::**PreTranslateMessage**(MSG* pMsg) —

позволяет до того, как сообщение будет обработано системными функциями, обработать его самостоятельно или отфильтровать часть сообщений. Возвращает значение TRUE, если сообщение, описанное по указателю pMsg, полностью обработано и, следовательно, не требует дальнейшей обработки системными средствами. Естественно, что эта функция вызывается только для интерфейсных потоков со стандартным циклом обработки сообщений, полностью аналогичным реализованному в функции Run класса CWinThread.

В любом приложении может возникнуть ошибка и не всегда она связана с ошибками, допущенными при программировании — например, может не хватить ресурсов. Для обработки такого рода ошибок класс CWinThread предусматривает следующую функцию:

virtual LRESULT CWinThread::**ProcessWndProcException**(
CException* e, const MSG* pMsg)

По умолчанию обработка производится только при возникновении следующих двух ситуаций:

- Ошибка при обработке сообщения WM_CREATE, связанного с невозможностью создания окна
- Обнуление области отсечения при обработке сообщения WM_PAINT для предотвращения генерации еще одного такого сообщения

Функция получает не только указатель на объект-исключение (параметр e), но и сопровождающее его сообщение (параметр pMsg). На возвращаемое значение библиотекой накладывается следующее ограничение — функция должна возвращать значение −1, если возникло исключение при обработке сообщения WM_CREATE, и 0 во всех остальных случаях.

Эта функция вызывается только для интерфейсных потоков при возникновении исключения всякий раз, когда не производится его обработка выше по коду в обработчиках соответствующих сообщений.

Создание интерфейсных потоков (продолжение)

Для того чтобы создать интерфейсный поток, необходимо выполнить не только вышеперечисленные действия, т. е. создать объект класса, производного от CWinThread и перегрузить часть его функций, но и еще обеспечить:

- Передачу командных сообщений из главного потока
- Корректное завершение текущего потока до завершения основного

В следующем примере это как раз и реализовано.

CUIThreadApp — класс всего приложения

В функции InitInstance создается основное окно, имеющее заголовок и меню для возможности задания скорости прокрутки "бегущей строки" (рис. 30.3).

Рис. 30.3. Приложение "Бегущая строка" за работой

```
// Файл UIThread.cpp
BOOL CUIThreadApp::InitInstance()
{
  ...
  CUIThreadFrame *pWnd = new CUIThreadFrame;
  m_pMainWnd = pWnd;
  pWnd->Create(NULL, _T("UIThread — Бегущая строка"),
    WS_OVERLAPPEDWINDOW, CFrameWnd::rectDefault,
    NULL, IDR_UITHREAD);
  pWnd->CenterWindow();
  pWnd->ShowWindow(m_nCmdShow);
  pWnd->UpdateWindow();
  return TRUE;
}
```

На этом роль класса приложения заканчивается. Все остальное, включая обработку команд, берут на себя остальные классы и библиотека.

CUIThreadFrame — класс основного окна

При создании основного окна создается окно-редактор (класс CEditWnd), занимающее большую часть основного окна. Кроме того, создается новый поток "бегущая строка" (класс CLineThread). Класс CUIThreadFrame отвечает за обработку команд меню. Часть команд в виде сообщений пересылается другому потоку в окно "бегущая строка". Функции класса синхронизируют завершение приложения так, чтобы до завершения основного потока завер-

шился дополнительный с освобождением всех захваченных ресурсов. Класс также отвечает за изменение размеров своих дочерних окон.

Следует отметить, что функции класса CUIThreadFrame выполняются в рамках основного потока приложения.

```
// Файл UIThreadFrm.h
class CUIThreadFrame: public CFrameWnd
{
public:
  CUIThreadFrame();
public:
  // Окно-редактор
  CEditWnd m_wndEdit;
  // Указатель на объект-поток
  CLineThread *m_pLineThread;
public:
  // ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CUIThreadFrame)
  public:
  virtual BOOL Create(LPCTSTR lpszClassName,
    LPCTSTR lpszWindowName, DWORD dwStyle,
    const RECT& rect, CWnd* pParentWnd, UINT nID,
    CCreateContext* pContext = NULL);
  virtual BOOL DestroyWindow();
  //}}AFX_VIRTUAL
protected:
  // Обработка командных сообщений
  virtual BOOL OnCmdMsg(UINT nID, int nCode, void* pExtra,
    AFX_CMDHANDLERINFO* pHandlerInfo);
  // message handlers
  //{{AFX_MSG(CUIThreadFrame)
  afx_msg void OnSize(UINT nType, int cx, int cy);
  //}}AFX_MSG
  afx_msg LRESULT OnPrepareToClose(WPARAM wParam = 0,
    LPARAM lParam = 0);
  DECLARE_MESSAGE_MAP()
};
```

В принципе нет необходимости хранить указатель на объект класса потока m_pLineThread — он нигде после создания потока не используется. Однако этот указатель может пригодиться при реализации метода синхронизации

завершения дополнительного потока, основанного на ожидании завершения потока, если использовать в качестве объекта синхронизации его дескриптор. В этом случае необходимо взять на себя удаление объекта класса CLineThread, установив элемент класса m_bAutoDelete в значение FALSE.

```
// Файл UIThreadFrm.cpp
BEGIN_MESSAGE_MAP(CUIThreadFrame, CFrameWnd)
  //{{AFX_MSG_MAP(CUIThreadFrame)
  ON_WM_SIZE()
  //}}AFX_MSG_MAP
  ON_MESSAGE(WM_USER_PREPARE_TO_CLOSE, OnPrepareToClose)
END_MESSAGE_MAP()
BOOL CUIThreadFrame::Create(LPCTSTR lpszClassName,
                            LPCTSTR lpszWindowName,
                            DWORD dwStyle,
                            const RECT& rect,
                            CWnd* pParentWnd,
                            UINT nID,
                            CCreateContext* pContext)
{
  if (!CFrameWnd::Create(lpszClassName, lpszWindowName,
    dwStyle, rect, pParentWnd, MAKEINTRESOURCE(nID), 0, pContext))
    return FALSE;
  CRect clientRect;
  GetClientRect(&clientRect);
  clientRect.top += 20;
  // Создаем окно-редактор
  m_wndEdit.Create(ES_MULTILINE | ES_AUTOHSCROLL | WS_BORDER |
    ES_AUTOVSCROLL | WS_CHILD | WS_VISIBLE,
    clientRect, this, IDC_LINE_WND+1);

  // Создаем новый поток, при этом будет создано и окно "бегущая строка"
  m_pLineThread = new CLineThread(m_hWnd);
  m_pLineThread->CreateThread();

  // Устанавливаем фокус на окно-редактор
  m_wndEdit.SetFocus();
  return TRUE;
}
//
```

```
// Функция перенаправления командных сообщений
//
BOOL CUIThreadFrame::OnCmdMsg(UINT nID, int nCode, void* pExtra,
    AFX_CMDHANDLERINFO* pHandlerInfo)
{
  // По умолчанию эта функция служит диспетчером команд меню
  // и команд инициализации, перенаправляя их в обработчики
  // ON_COMMAND и ON_UPDATE_COMMAND_UI.
  // Причем все это происходит в основном потоке приложения.
  // Однако объект класса CLineWnd находится в другом потоке.
  // Вызов напрямую функции CLineWnd::OnCmdMsg возможен,
  // но в этом случае обработка сообщений будет происходить
  // в этом основном потоке. Самый лучший способ - передать
  // сообщение окну, собрав в специальной структуре необходимые
  // данные. Это самый надежный и правильный способ,
  // исключающий вызов напрямую функций другого потока.
  CWnd* pLineWnd = (CLineWnd*)GetDlgItem(IDC_LINE_WND);
  if (pLineWnd == NULL)
    return FALSE; // Окно CLineWnd еще не создано
  // Создаем объект CCmdMsg в стеке, поскольку функция SendMessage
  // осуществляет синхронную передачу сообщений, т.о. она завершится
  // только по завершении обработки сообщения
  CCmdMsg cmdmsg;
  cmdmsg.m_nID = nID;
  cmdmsg.m_nCode = nCode;
  cmdmsg.m_pExtra = pExtra;
  cmdmsg.m_pHandlerInfo = pHandlerInfo;
  return pLineWnd->SendMessage(WM_USER_ONCMDMSG,
    GetCurrentMessage()->wParam, (LPARAM)&cmdmsg);
}
```

При удалении главного окна приложения будет удалено и окно CLineWnd, которое является его дочерним окном, пусть и выполняемом в другом потоке. В результате удаления окна CLineWnd будет удален объект класса CLineThread и соответствующий поток (см. комментарии к классу CLineThread).

```
BOOL CUIThreadFrame::DestroyWindow()
{
  OnPrepareToClose();
  return CFrameWnd::DestroyWindow();
```

```
}
LRESULT CUIThreadFrame::OnPrepareToClose(WPARAM, LPARAM)
{
  // Отправляем сообщение окну CLineWnd на завершение и ждем,
  // пока это сообщение не будет обработано.
  // Ожидаем сигнала о завершении соответствующего потока
  CWnd* pLineWnd = (CLineWnd*)GetDlgItem(IDC_LINE_WND);
  pLineWnd->SendMessage(WM_USER_PREPARE_TO_CLOSE);
  WaitForSingleObject(
    CLineThread::m_hEventLineThreadKilled, INFINITE);
  return 0;
}
```

Обратите внимание, что на момент вызова функции WaitForSingleObject объект потока фактически может не существовать, что и обусловило использование именно статического элемента-события (CLineThread ::m_hEventLineThreadKilled). Такая методика, кстати, не исключает одновременное создание нескольких потоков на базе класса CLineThread даже при том, что синхронизирующий объект один! Каждому окну перед ожиданием завершения соответствующего потока посылается сообщение о необходимости завершения. Следовательно, все, что требуется сделать при количестве окон большем одного, — это после завершения очередного потока сбрасывать объект-событие при помощи функции ResetEvent.

Собственно наличие механизма корректного завершения потока и соответственно освобождения памяти, занимаемой объектом класса CLineThread, никак не сказывается на работе приложения. При отсутствии этого механизма только в режиме отладки можно будет обнаружить сообщения о неосвобожденном участке памяти, выделенном под объект класса CLineThread. Причем это может происходить далеко не всегда — этот объект создан таким образом, что при завершении потока происходит автоматическое удаление и объекта класса. Однако не факт, что главный поток приложения и соответственно процесс в целом не успеют завершиться до освобождения памяти.

```
void CUIThreadFrame::OnSize(UINT nType, int cx, int cy)
{
  CFrameWnd::OnSize(nType, cx, cy);
  CWnd* pLineWnd = GetDlgItem(IDC_LINE_WND);
  if (pLineWnd == NULL)
    return; // Окно CLineWnd еще не создано
  CRect rect;
  GetClientRect(&rect);
  pLineWnd->SetWindowPos(NULL, rect.top, rect.left,
```

```
    rect.Width(), 20,
    SWP_NOACTIVATE | SWP_NOMOVE | SWP_NOZORDER);
  m_wndEdit.SetWindowPos(NULL, rect.top+20, rect.left,
    rect.Width(), rect.Height()-20,
    SWP_NOACTIVATE | SWP_NOMOVE | SWP_NOZORDER);
}
```

Примечание

Для простоты мы не стали вычислять размеры, требуемые для вывода одной строки, предполагая, что 20 пикселей будет достаточно. На самом деле требуются более точные значения.

CLineThread — класс дополнительного потока

Основная задача этого класса — в создании окна "бегущая строка" и организации его перерисовки в момент простоя (функция OnIdle).

```
// Файл LineThread.h
class CLineThread: public CWinThread
{
  DECLARE_DYNCREATE(CLineThread)
protected:
  CLineThread();            // protected constructor used by
                            // dynamic creation
public:
  CLineThread(HWND hwndParent);
  void operator delete(void* p);
public:
  static HANDLE m_hEventLineThreadKilled;
protected:
  HWND m_hwndParent;
  CLineWnd m_wndLine;
public:
  // ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CLineThread)
  public:
  virtual BOOL InitInstance();
  virtual int ExitInstance();
  virtual BOOL OnIdle(LONG lCount);
  //}}AFX_VIRTUAL
```

```
protected:
  virtual ~CLineThread();
  // Generated message map functions
  //{{AFX_MSG(CLineThread)
    // NOTE — the ClassWizard will add and remove
    // member functions here.
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()
};
// Файл LineThread.cpp
HANDLE CLineThread::m_hEventLineThreadKilled;
IMPLEMENT_DYNCREATE(CLineThread, CWinThread)
```

Многопотоковые приложения отличаются от обычных MFC-приложений еще и тем, как происходит доступ к MFC-объектам из разных потоков. Эта тема отдельного раздела этой главы, сейчас отметим только, что в конструктор класса CLineThread передается не указатель на объект класса CUIThreadFrame, а дескриптор связанного с ним окна.

```
CLineThread::CLineThread(HWND hwndParent): m_hwndParent(hwndParent)
{
}
```

Сигнализация о завершении потока происходит в самый последний момент, когда поток фактически уже прекратил свое существование, а объект класса CLineThread уже "удаляется".

```
void CLineThread::operator delete(void* p)
{
  CWinThread::operator delete(p);
  SetEvent(m_hEventLineThreadKilled);
}
BOOL CLineThread::InitInstance()
{
  CWnd* pParent = CWnd::FromHandle(m_hwndParent);
  CRect rect;
  pParent->GetClientRect(&rect);
  rect.bottom = rect.top + 20;
  BOOL bReturn = m_wndLine.Create(_T("LineMTChildWnd"),
    WS_CHILD | WS_VISIBLE, rect, pParent);
  // Для того чтобы интерфейсный поток завершался автоматически сразу
  // после уничтожения окна, следует связать поток с этим окном
if (bReturn)
```

```
    m_pMainWnd = &m_wndLine;
  return bReturn;
}
int CLineThread::ExitInstance()
{
  return CWinThread::ExitInstance();
}
BOOL CLineThread::OnIdle(LONG lCount)
{
  if (::IsWindow(m_wndLine.m_hWnd))
  {
    // Для управления скоростью "бегущей строки" используем значение
    // параметра lCount. Игнорируем его нулевое значение,
    // т.к. в противном случае строка будет перемещаться при большом
    // количестве обычных сообщений, например, от мыши - после каждого
    // такого сообщения функция OnIdle может вызываться,
    // причем параметр lCount при этом будет равен 0
    if (!((lCount+1) % m_wndLine.m_Speed))
      m_wndLine.Redraw();
    return TRUE;
  }
  return CWinThread::OnIdle(lCount);
}
```

Код функции OnIdle имеет один существенный недостаток, который никак не сказывается на самом приложении, но тем не менее негативно влияет на всю систему в целом. С точки зрения приложения, а точнее второго его потока, процесс перемещения строки текста в окне использует то время, когда самому потоку процессор не нужен, т. к. нет сообщений. Однако в результате такой организации, поток постоянно выполняет либо функцию OnIdle, либо обслуживает сообщения. Альтернативой такой реализации является создание таймера для окна класса CLineWnd, обработчик сообщений от которого и должен производить перерисовку. В Windows 95/98 и Windows NT до истечения заданного интервала времени приложение может продолжать выполнять какой-либо код, не занимаясь при этом опросом часов или операционной системы, которая также не использует механизм активного ожидания. Таким образом, поток ожидает поступления сообщения, в том числе и от таймера, и не использует процессор.

CLineWnd — класс окна "бегущая строка"

Функции объекта этого класса выполняются в дополнительном потоке, хотя сам он является дочерним окном главного окна приложения. Функции

класса служат диспетчером команд, обрабатывают команды меню и команды обновления дочернего окна. Также класс неявно отвечает за завершение потока, в котором выполняются его функции.

```
// Файл LineWnd.h
#define IDC_LINE_WND 1
// Структура CCmdMsg используется для передачи параметров
// командных сообщений между окнами различных потоков
struct CCmdMsg
{
  int m_nID;
  int m_nCode;
  void* m_pExtra;
  AFX_CMDHANDLERINFO* m_pHandlerInfo;
};
#define WM_USER_ONCMDMSG (WM_USER + 0x1000)
#define WM_USER_PREPARE_TO_CLOSE (WM_USER + 0x1001)
```

Далее идет описание самого обычного класса окна. Действительно, ему нет никакого дела до того, в каком потоке он будет выполняться — все действия по организации такого выполнения берут на себя классы CUITreadFrame и CLineThread. Конечно все это верно с учетом того, что без обработчиков сообщений WM_USER_ONCMDMSG и WM_USER_PREPARE_TO_CLOSE (особенно первого) в *отдельном* потоке ничего работать не будет.

```
// CLineWnd window
class CLineWnd: public CWnd
{
  DECLARE_DYNAMIC(CLineWnd)
public:
  CLineWnd();
  BOOL Create(LPCTSTR szTitle, LONG style,
    const RECT& rect, CWnd* pParent);
public:
  CBrush m_Brush;
  int m_cxClient;
  int m_cyClient;
  int m_Speed;
  enum { SlowSpeed = 5000, NormalSpeed = 2500, FastSpeed = 1000 };
public:
  void Redraw();
  // ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CLineWnd)
```

```
  //}}AFX_VIRTUAL
public:
  virtual ~CLineWnd();
protected:
  //{{AFX_MSG(CLineWnd)
  afx_msg void OnSize(UINT nType, int cx, int cy);
  afx_msg void OnPaint();
  afx_msg void OnSpeedLow();
  afx_msg void OnSpeedNormal();
  afx_msg void OnSpeedHigh();
  afx_msg void OnUpdateSpeedSlow(CCmdUI* pCmdUI);
  afx_msg void OnUpdateSpeedNormal(CCmdUI* pCmdUI);
  afx_msg void OnUpdateSpeedFast(CCmdUI* pCmdUI);
  //}}AFX_MSG
  afx_msg LRESULT OnDelegatedCmdMsg(WPARAM, LPARAM);
  afx_msg LRESULT OnPrepareToClose(WPARAM wParam = 0,
    LPARAM lParam = 0);
  DECLARE_MESSAGE_MAP()
};
```

Обратите внимание, что обработчики командных сообщений организованы как обычно — их вызовом будет заниматься функция OnCmdMsg, которая, находясь в основном потоке, будет вызываться библиотекой, а в нашем случае — функцией OnDelegatedCmdMsg.

```
// Файл LineWnd.cpp
IMPLEMENT_DYNAMIC(CLineWnd, CWnd)
// CLineWnd
CLineWnd::CLineWnd(): m_Speed(NormalSpeed)
{
}
BEGIN_MESSAGE_MAP(CLineWnd, CWnd)
  //{{AFX_MSG_MAP(CLineWnd)
  ON_WM_SIZE()
  ON_WM_PAINT()
  ON_COMMAND(ID_SPEED_SLOW, OnSpeedSlow)
  ON_COMMAND(ID_SPEED_NORMAL, OnSpeedNormal)
  ON_COMMAND(ID_SPEED_FAST, OnSpeedFast)
  ON_UPDATE_COMMAND_UI(ID_SPEED_SLOW, OnUpdateSpeedSlow)
  ON_UPDATE_COMMAND_UI(ID_SPEED_NORMAL, OnUpdateSpeedNormal)
  ON_UPDATE_COMMAND_UI(ID_SPEED_FAST, OnUpdateSpeedFast)
  //}}AFX_MSG_MAP
```

```
  ON_MESSAGE(WM_USER_ONCMDMSG, OnDelegatedCmdMsg)
  ON_MESSAGE(WM_USER_PREPARE_TO_CLOSE, OnPrepareToClose)
END_MESSAGE_MAP()
```

Алгоритм следующей функции в данном случае не важен. Главное — обратите внимание на то, что данные для "бегущей строки" копируются во временную переменную из разделяемого между потоками объекта, доступ к которому должен быть монопольным. И это достигается использованием объекта "взаимоисключение" (mutex). Тот же метод реализован в классе CEditWnd.

```
void CLineWnd::Redraw()
{
  static int nOffset_1 = 0;
  // Организация взаимоисключающего доступа к элементу
  // CUIThreadApp::m_Text, который содержит строку текста
  ((CUIThreadApp *)AfxGetApp())->m_TextLock.Lock();
  CString csText = ((CUIThreadApp *)AfxGetApp())->m_Text;
((CUIThreadApp *)AfxGetApp())->m_TextLock.Unlock();
  CClientDC dc(this);
  dc.SetBkColor(RGB(192,192,192));
  CRect rect(0, 0, m_cxClient, m_cyClient);
  dc.DrawText(csText, &rect,
    DT_SINGLELINE | DT_VCENTER | DT_NOPREFIX | DT_CALCRECT);
  int nWidth = rect.Width();
  rect.OffsetRect(nOffset_1, 0);
  dc.DrawText(csText, &rect,
    DT_SINGLELINE | DT_VCENTER | DT_NOPREFIX);
  int nOffset_2 = nWidth + nOffset_1 + 50;
  if (nOffset_2 — nOffset_1 < m_cxClient)
    nOffset_2 = m_cxClient + nOffset_1;
  rect.OffsetRect(nOffset_2 — nOffset_1, 0);
  dc.DrawText(csText, &rect,
    DT_SINGLELINE | DT_VCENTER | DT_NOPREFIX);
  dc.FillRect(CRect(nOffset_1+nWidth, 0, nOffset_2,
    m_cyClient), &m_Brush);
  if (nOffset_2+nWidth < m_cxClient)
    dc.FillRect(CRect(nOffset_2+nWidth, 0, m_cxClient,
    m_cyClient), &m_Brush);
  nOffset_1--;
  if (nOffset_2 == 0)
    nOffset_1 = nOffset_2;
}
```

```
BOOL CLineWnd::Create(LPCTSTR szTitle, LONG style,
  const RECT& rect, CWnd* parent)
{
  m_Brush.CreateSolidBrush(RGB(192,192,192));
  // Регистрируем класс окна, т.к. оно имеет отличную от
  // принятой кисть и не имеет значка
  LPCTSTR lpszLineClass =
    AfxRegisterWndClass(CS_HREDRAW | CS_VREDRAW | CS_SAVEBITS,
      LoadCursor(NULL, IDC_ARROW), m_Brush, NULL);
  return CWnd::Create(lpszLineClass, szTitle, style,
    rect, parent,  IDC_LINE_WND);
}
LRESULT CLineWnd::OnDelegatedCmdMsg(WPARAM, LPARAM lParam)
{
  CCmdMsg* pcmdMsg = (CCmdMsg*)lParam;
  return CWnd::OnCmdMsg(pcmdMsg->m_nID, pcmdMsg->m_nCode,
    pcmdMsg->m_pExtra,  pcmdMsg->m_pHandlerInfo);
}
LRESULT CLineWnd::OnPrepareToClose(WPARAM, LPARAM)
{
  DestroyWindow();
  return 0;
}
void CLineWnd::OnSize(UINT nType, int cx, int cy)
{
  CWnd::OnSize(nType, cx, cy);
  // Запоминаем размер клиентской области окна, чтобы каждый раз
  // при перерисовке не запрашивать размер заново
  m_cxClient = cx;
  m_cyClient = cy;
}
void CLineWnd::OnPaint()
{
  CPaintDC dc(this); // device context for painting
CRect rect(0, 0, m_cxClient, m_cyClient);
  dc.FillRect(&rect, &m_Brush);
  Redraw();
}
void CLineWnd::OnSpeedSlow()
```

```
{
  m_Speed = SlowSpeed;
}
void CLineWnd::OnSpeedNormal()
{
  m_Speed = NormalSpeed;
}
void CLineWnd::OnSpeedFast()
{
  m_Speed = FastSpeed;
}
void CLineWnd::OnUpdateSpeedSlow(CCmdUI* pCmdUI)
{
  pCmdUI->SetCheck(m_Speed == SlowSpeed);
}
void CLineWnd::OnUpdateSpeedNormal(CCmdUI* pCmdUI)
{
  pCmdUI->SetCheck(m_Speed == NormalSpeed);
}
void CLineWnd::OnUpdateSpeedFast(CCmdUI* pCmdUI)
{
  pCmdUI->SetCheck(m_Speed == FastSpeed);
}
```

CEditWnd — класс окна редактора

Класс занимается тем, что раз в секунду обновляет содержимое специального элемента класса CUIThreadApp, в который записывает содержимое редактора. Именно через этот элемент и осуществляется обмен между потоками — он содержит информацию для "бегущей строки" (рис. 30.4).

Рис. 30.4. Приложение в процессе редактирования. Текст изменился, однако содержимое "бегущей строки" еще об этом "не знает"

```
// Файл EditWnd.h
class CEditWnd: public CEdit
{
public:
  CEditWnd();
  // ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CEditWnd)
  //}}AFX_VIRTUAL
public:
  virtual ~CEditWnd();
protected:
  //{{AFX_MSG(CEditWnd)
  afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
  afx_msg void OnTimer(UINT nIDEvent);
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()
};
// Файл EditWnd.cpp
BEGIN_MESSAGE_MAP(CEditWnd, CEdit)
  //{{AFX_MSG_MAP(CEditWnd)
  ON_WM_CREATE()
  ON_WM_TIMER()
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

Для того чтобы обновлять разделяемый между потоками объект, создаем таймер и обработчик сообщения от него, в котором и происходит собственно обновление. Именно поэтому содержимое "бегущей строки" может измениться не сразу — объект обмена содержит устаревшие и еще не обновленные данные.

```
int CEditWnd::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
  if (CEdit::OnCreate(lpCreateStruct) == -1)
    return -1;
SetTimer(1, 1000, NULL);
return 0;
}
void CEditWnd::OnTimer(UINT nIDEvent)
{
  if (nIDEvent == 1)
  {
```

```
    ((CUIThreadApp *)AfxGetApp())->m_TextLock.Lock();
    GetWindowText(((CUIThreadApp *)AfxGetApp())->m_Text);
    ((CUIThreadApp *)AfxGetApp())->m_TextLock.Unlock();
  }
CEdit::OnTimer(nIDEvent);
}
```

Создание рабочих потоков

Рабочие потоки (worker threads) обычно создаются для выполнения каких-либо фоновых действий, завершения которых не требуется ожидать, прерывая работу приложения. Примером таких действий может служить фоновое сохранение файлов, печать и вычисления.

В отличие от интерфейсных потоков, рабочие создавать существенно проще, что не уменьшает их значение. Например, добавление в MS Word 97 возможности сохранения файлов в фоновом режиме позволяет не отвлекаться от работы с большими документами во время их автосохранения.

Для создания рабочего потока требуется создать:

- Управляющую функцию
- Собственно поток (одним из двух способов, описанных ниже)

Создание рабочего потока возможно с использованием уже описанной функции ::AfxBeginThread, которой в качестве первого параметра передается указатель на управляющую функцию, т. е. просто ее имя, либо вызов с использованием специального конструктора класса CWinThread, а затем вызова функции CreateThread для созданного объекта.

Первый способ более надежен прежде всего потому, что функция ::AfxBeginThread осуществляет все возможные проверки и при возникновении ошибки уничтожает все созданные во время ее работы объекты. Хотя при этом она создает новый поток как раз вторым описанным способом. Вот описание конструктора класса CWinThread, предназначенного для создания рабочих потоков:

CWinThread::**CWinThread**(AFX_THREADPROC pfnThreadProc, LPVOID pParam) —

параметры задают управляющую функцию (pfnThreadProc) и значение параметра для нее (pParam). Все остальные параметры, необходимые для запуска потока, задаются в функции AfxBeginThread (первый способ) или в функции CreateThread класса CWinThread (второй способ).

Управляющая функция и ее параметр

Управляющая функция — это практически весь код, который составляет поток. Кроме нее в рамках созданного потока выполняется код библиотеч-

ных функций, осуществляющих все необходимые действия для правильного функционирования потока. Но важно понимать, что все дополнительные действия производятся только до и после выполнения управляющей функции, так что с момента ее вызова и до ее завершения поток находится полностью во власти разработчика. Завершение функции приводит и к завершению потока.

Объявление управляющей функции должно быть таким:

```
UINT ThreadControllingFunction(LPVOID pParam);
```

В качестве параметра этой функции используется 32-битное значение. Этот параметр имеет то же значение, что задается при создании потока в конструкторе или функции создания потока. При этом интерпретация значения зависит от разработчика, это может быть число, указатель на любой тип данных (массив, структуру, класс) или может вообще игнорироваться. Этот параметр может не только передавать какое-либо значение в управляющую функцию, но также служить "мостом" для передачи данных из потока, например, если фактически параметр является указателем на объект. Естественно, что в этом случае поток должен тем или иным способом сигнализировать о том, что данные готовы.

По завершении функции код завершения потока устанавливается равным величине возвращаемого значения функции. Обычно принято возвращать значение 0 в случае успешного завершения потока и ненулевое значение в случае возникновения той или иной ошибки. Естественно, что смысл кода завершения потока зависит полностью от нужд разработчика, например, код завершения может сообщать о количестве захваченных ресурсов и т. п.

Пример управляющей функции

```
// Фоновые вычисления
UINT ThreadProc(LPVOID pParam)
{
  CDataObject* pObject = (CDataObject*)pParam;
  if (pObject == NULL ||
    !pObject->IsKindOf(RUNTIME_CLASS(CDataObject)))
  return -1; // Неверный параметр
  // Основные вычисления
  ...
  return 0; // Все в порядке
}
// Другая функция приложения
{
  ...
```

```
  CDataObject *pObject = new CDataObject;
  AfxBeginThread(ThreadProc, pObject);
  ...
}
```

Завершение потоков

Как уже упоминалось, каждый рабочий процесс завершается вместе с управляющей функцией. Таким образом, завершение действия, ради которого создавался поток, приводит к его *нормальному* завершению. Однако может оказаться так, что во время выполнения потока возникает необходимость в завершении его еще до окончания выполнения им предусмотренных действий, т. е. в *преждевременном* завершении. Как и в первом случае, если инициатором завершения потока является сам поток, то можно говорить о корректном завершении. Но не всякий поток можно создать таким образом, чтобы он самостоятельно определял возникновение необходимости преждевременного завершения. В принципе, если такая необходимость возникает в связи с завершением процесса в целом, а действия, выполняемые потоком, можно безболезненно прервать, то беспокоиться не стоит — поток обязательно будет завершен, а все захваченные им ресурсы освобождены.

Нормальное завершение

Итак, для рабочих потоков нормальным завершением является завершение управляющей функции, возвращаемое значение которой и будет кодом завершения. Хорошей практикой является установка кода возврата в 0 в случае успешного выполнения предусмотренных действий. Однако определение смысла кода возврата возлагается полностью на разработчика.

Для интерфейсных потоков аналогом нормального завершения служит посылка сообщения WM_QUIT главному окну потока. Это сообщение следует посылать при помощи вызова функции Win32 API ::PostQuitMessage, а в качестве значения ее единственного параметра следует указывать как раз код завершения потока.

Преждевременное завершение

Выполнение преждевременного завершения для потоков обоих типов в любом месте кода, выполняющегося в *рамках* потока, который требуется завершить, осуществляется простым вызовом функции ::AfxEndThread:

void::**AfxEndThread**(UINT nExitCode) —

параметр nExitCode определяет код завершения потока. Функция не только завершает поток, но и освобождает стек потока, связанные с потоком библиотеки динамической компоновки (DLL) и удаляет, по умолчанию (см. описание элемента CWinThread::m_bAutoDelete), объект класса CWinThread.

Примечание

Использование этой функции для преждевременного завершения главного потока и, соответственно, процесса приведет к ошибке времени выполнения, так что функцией ::AfxEndThread можно пользоваться только для завершения дополнительных потоков.

Для завершения потока, как нормального, так и преждевременного, из другого потока следует предусмотреть метод взаимодействия между потоками. Причем для интерфейсных потоков это сделать существенно проще — достаточно послать сообщение либо его главному окну, либо непосредственно самому потоку при помощи функции CWnd::PostMessage, либо функции Win32 API ::PostMessage или ::PostThreadMessage. Необходимо знать только соответственно указатель, дескриптор или идентификатор.

Далее приведены некоторые варианты использования этих функций, т. к. они нас интересуют только с точки зрения завершения потоков.

```
m_pThreadMainWnd->PostMessage(WM_QUIT, nDesireExitCode, 0);
```

или

```
::PostMessage(m_hThreadMainWnd, WM_QUIT, nDesireExitCode, 0);
```

или

```
::PostThreadMessage(m_nThreadID, WM_QUIT, nDesireExitCode, 0);
```

Примечание

Использование одного этого метода не обеспечивает не только какой-либо синхронизации завершения потока с работой остальных потоков приложения, но и вообще не гарантирует собственно завершение потока! Это может быть связано с тем, что поток в результате ошибки не обрабатывает сообщения, а также, в случае использования последней функции ::PostThreadMessage, с тем, что на момент ее вызова поток еще не успел вызвать ни одной функции из модулей USER или GDI, в результате чего для потока не была создана очередь сообщений, и, как следствие, сообщение WM_QUIT не будет поставлено в нее (функция ::PostThreadMessage вернет значение FALSE — ошибка выполнения).

И последнее, что необходимо отметить. Для интерфейсных потоков преждевременное завершение скорее всего приведет к появлению в отладчике сообщения о невозможности вызова в деструкторе класса CWnd функций OnDestroy или PostNcDestroy производного класса, т. к. в деструкторе базового класса VTBL (виртуальная таблица) не указывает на объект производного класса:

```
Warning: calling DestroyWindow in CWnd::~CWnd;
  OnDestroy or PostNcDestroy in derived class will not be called
```

Конечно в этом нет ничего "страшного" — захваченная память будет освобождена, однако если в этих функциях предусмотрены дополнительные действия, то они не будут произведены, что рано или поздно скажется на выполнении всего процесса в целом.

Экстренное завершение

К сожалению, не всегда есть возможность корректно завершить поток. Он может войти в бесконечный цикл или в результате ошибки не сработает реализованный способ взаимодействия. В такой ситуации может понадобиться *экстренное* завершение потока. Происходит это самым безжалостным образом, после такого завершения нет стандартной возможности определить, в какой момент был остановлен поток, не происходит освобождение памяти, включая стек потока, связанные с потоком библиотеки динамической компоновки (DLL) не уведомляются о его завершении и соответственно могут зря расходовать системные ресурсы и ресурсы процесса.

Более того, экстренное завершение потока может привести к куда более серьезным последствиям:

- если поток выполняет определенный код ядра операционной системы (kernel32) в момент завершения, системный статус процесса, к которому принадлежит (или принадлежал) поток, может оказаться неопределенным;
- если поток во время завершения манипулировал глобальным статусом совместно используемой библиотеки динамической компоновки (DLL), статус последней может быть безвозвратно удален, что непременно скажется на других процессах и потоках, которые также используют эту библиотеку;
- если поток в момент завершения находится в своей критической секции, объект "критическая секция" не будет освобожден никогда, что может привести к возникновению тупика — другие потоки, ожидающие освобождения этого объекта, так никогда и не дождутся его освобождения.

Приведенных примеров вполне достаточно, чтобы сформулировать правило, следуя которому, можно более или менее гарантировать отсутствие серьезных сбоев при выполнении экстренного завершения или прерывания потоков.

Правило

Прерывание потока допустимо только в случае, если *точно* известно, *что* поток делает, и при этом *точно* известно, *какой* код выполняет поток в момент его экстренного завершения.

Таким образом, при невыполнении этого правила, возможно, достаточно будет просто приостановить поток при помощи функции Win32 API ::SuspendThread или функции библиотеки MFC CWinThread::SuspendThread.

Собственно экстренное завершение потока осуществляется функцией Win32 API:

BOOL ::**TerminateThread**(HANDLE hThread, DWORD dwExitCode) —

параметрами этой функции являются дескриптор потока, который требуется прервать, а также значение желаемого кода завершения потока. Причем дескриптор потока должен иметь доступ THREAD_TERMINATE.

Сам поток не может "защитить" себя от функции TerminateThread иначе, чем контролируя доступ к своим дескрипторам. Но так как функция Win32 API, создающая поток, CreateThread или функция CreateProcess, которая, естественно, создает вместе с процессом его главный поток, возвращают дескрипторы изначально имеющие доступ THREAD_TERMINATE, любой поток или процесс, имеющие дескриптор другого потока, полученный в результате выполнения этих функций, может его прервать.

Экстренное завершение последнего потока процесса приводит также к экстренному завершению самого процесса со всеми вытекающими из этого последствиями.

Состояние потока после осуществления экстренного завершения переводится в состояние "установлен", что позволяет продолжить свое выполнение потокам, ожидающим завершения прерванного потока. При этом значение его статуса меняется со STILL_ACTIVE на значение параметра dwExitCode.

Экстренное завершение потока необязательно приводит к его удалению из системы. Объект "поток" удаляется, когда освобождается его последний дескриптор.

Код завершения

Для получения кода завершения потока, рабочего или интерфейсного, следует вызвать функцию Win32 API :

BOOL ::**GetExitCodeThread**(HANDLE hThread, LPDWORD lpExitCode) —

помещает в переменную, заданную указателем lpExitCode, код завершения потока, заданного дескриптором hThread. Для получения кода завершения дескриптор должен иметь доступ THREAD_QUERY_INFORMATION.

Если поток все еще существует, код завершения будет равен STILL_ACTIVE. Кроме кода завершения собственно потока, функция может сообщить код завершения процесса в случае, если завершение потока привело к завершению процесса, т. к. код завершения главного потока совпадает с кодом завершения процесса.

Для потоков, созданных на базе класса CWinThread, получение кода завершения требует дополнительных подготовительных действий, т. к. сам объект этого класса хранит дескриптор потока в элементе m_hThread, а по умолчанию завершение потока приводит к удалению объекта этого класса и, следо-

вательно, доступ к этому элементу невозможен. Поэтому рекомендуется выполнить одно из двух следующих действий сразу же после создания потока:

- После создания потока установить значение элемента класса m_bAutoDelete равным FALSE, что исключит удаление объекта после завершения потока. Таким образом, до явного удаления объекта класса CWinThread всегда можно использовать элемент m_hThread для попытки получения кода завершения потока. Используя этот механизм, следует не забыть о необходимости самостоятельно освободить память, занимаемую объектом, как только необходимость в информации о потоке будет не нужна, а сам поток завершен.
- После создания потока следует отдельно сохранить копию дескриптора потока. Правильнее это сделать при помощи функции Win32 API ::DuplicateHandle. (В примере, посвященном наследованию, для простоты передачи дескриптора мы обошлись просто копированием значений.)

Оба эти подхода требуют, чтобы в момент изменения элемента m_bAutoDelete или копирования дескриптора поток еще не завершился. Для этого можно порекомендовать создавать поток, изначально приостановленный (флаг при создании CREATE_SUSPENDED), и только после выполнения вышеперечисленных действий следует возобновить поток при помощи функции ResumeThread.

Права доступа к потоку

Функция ::DuplicateHandle позволяет создавать копии дескрипторов, имеющих ограниченные права доступа. Здесь, наверное, необходимо пояснить, что многие объекты Windows, описываемые дескрипторами, могут иметь ограничения по доступу к ним. Так как собственно объекты являются некоторыми структурами данных, а доступ к ним осуществляется исключительно системой по дескрипторам, то вполне допустимо говорить о правах доступа и самих дескрипторов.

Для дескрипторов потоков Windows предусматривает следующие права доступа:

SYNCHRONIZE	(Только для Windows NT) позволяет использовать дескриптор для ожидания завершения потоков в любой из ожидающих функций
THREAD_ALL_ACCESS	Дескриптор позволяет осуществлять любой вид доступа к потоку
THREAD_DIRECT_IMPERSONATION	Для внутреннего использования
THREAD_GET_CONTEXT	Дескриптор позволяет получать контекст потока
THREAD_IMPERSONATE	Для внутреннего использования

THREAD_QUERY_INFORMATION	Дескриптор позволяет получать код завершения, приоритет и элементы таблицы дескрипторов потока по заданному селектору
THREAD_SET_CONTEXT	Дескриптор позволяет устанавливать контекст потока
THREAD_SET_INFORMATION	Дескриптор позволяет назначать относительный приоритет потоку
THREAD_SET_THREAD_TOKEN	Для внутреннего использования
THREAD_SUSPEND_RESUME	Дескриптор позволяет приостанавливать и возобновлять поток
THREAD_TERMINATE	Дескриптор позволяет экстренно завершать поток

Глава 31

Взаимодействие потоков

Взаимодействие потоков, относящихся к одному или к различным процессам, требует специальных приемов, с помощью которых достигаются две основных цели:

- Возможность синхронизировать деятельность потоков путем посылки ими друг другу определенных "сигналов"
- Осуществление корректного доступа к разделяемым данным

Взаимодействие должно быть реализовано достаточно просто и не требовать чрезмерных накладных расходов. В этой главе мы не будет рассматривать возможность обмена информацией между различными процессами. Поэтому нас будет интересовать взаимоисключение доступа прежде всего к совместно используемым участкам памяти одного процесса, а также работа с разделяемыми объектами Windows, которые использует процесс во время работы. Если синхронизация потоков (посылка сигналов) в принципе не требует организации взаимоисключения доступа, то как раз работа с разделяемыми объектами Windows в явном или неявном виде обязательно использует такую синхронизацию. И последнее, что осталось отметить — это то, как сама операционная система и классы библиотеки MFC вносят определенную специфику в рассматриваемые вопросы.

Взаимоисключение доступа

Довольно часто приходится сталкиваться с такой ситуацией — дверью вагона метро одновременно пытаются воспользоваться как выходящие, так и входящие, в результате налицо неопределенность — люди и входят, и выходят одновременно (пытаются), происходит потеря времени и нервных клеток. Однако электроника подчиняется более строгим правилам, исключающим такого рода неопределенность. При обращении одного процессора к ячейке памяти блокируется доступ к ней для другого. Причем все это реализуется на аппаратном уровне. Но программы работают с существенно боль-

шими по объему данными, которые занимают более одной ячейки памяти. Теоретически может возникнуть ситуация, когда в одну часть структуры осуществляет запись один процессор, а в другую — другой.

Параллельное выполнение и обработка данных возможны и на однопроцессорной машине. Это может привести к тому, что один поток осуществляет запись в массив, в то время как другой производит чтение из того же массива. В общем случае результат чтения не определен: прочитаны могут только старые данные, только новые или частично те и другие.

Самое интересное, что проблемы могут возникнуть при одновременном доступе к данным, занимающим не только несколько ячеек, но даже всего один байт памяти. Одна операция на языке высокого уровня может быть представлена целым набором команд процессора, которые только и являются неделимыми, т. е. именно они не могут быть прерваны, после же выполнения инструкции процессор вполне может быть переключен на выполнение другого потока. В этом смысле практически каждая операция языка высокого уровня может продвигаться к завершению выполнения совместно с другими операциями других потоков своего же или другого процесса.

Вот пример кода, иллюстрирующий ситуацию, когда из-за отсутствия взаимоисключающего доступа может возникнуть ошибка.

```
int nCount;
...
nCount += nCount;
```

Последняя строчка примера в результате компиляции заменяется следующими двумя строчками кода, представленными на ассемблере:

```
1: mov eax,dword ptr [nCount]
2: add dword ptr [nCount],eax
```

Переключение сразу после выполнения первой строки кода на другой поток, в котором происходит изменение переменной nCount, приведет к тому, что результат удвоения значения этой переменной будет неверным, т. е. при параллельных вычислениях дважды два не всегда четыре! Отсюда:

Правило

Доступ на изменение разделяемых данных более чем в одном потоке должен быть *взаимоисключающим*.

Однако вполне допустимым является продолжение выполнения потока в цикле, условием для которого может служить значение некоторой глобальной переменной без реализации взаимоисключающего доступа, например так:

```
BOOL g_bEnough = FALSE;
```

Первый поток:

```
...
g_bEnough = TRUE;
...
```

Второй поток:

```
...
while (g_bEnough == FALSE)
{
...
}
...
```

Организация доступа к объектам классов MFC

Сам язык программирования C++ (да и C тоже) не имеет в своем составе средств параллельного программирования, создание классов или структур для их использования в параллельных вычислениях не возможно без специальных средств, требующих существенных накладных расходов. Тем не менее, их можно использовать в том случае, если доступ к данным класса осуществляется только при помощи функций-членов, то есть применения взаимоисключения в "явном" виде может и не быть. Достаточно, чтобы эти функции использовали механизм взаимоисключения.

Когда речь идет о собственных классах, то вся ответственность, естественно, лежит на разработчиках. Если же приложение пользуется классами, например, библиотеки MFC, то сделать уже ничего нельзя — либо взаимоисключающий доступ реализован, либо нет.

Из соображений эффективности классы библиотеки MFC не используют взаимоисключение. Из этого следует, что возможно использование различных объектов какого-либо класса в нескольких потоках, но работа с одним и тем же объектом требует использования механизма синхронизации доступа. Однако для защиты глобальных структур данных библиотека MFC использует механизм критических секций.

Кроме ограничения использования объектов библиотеки MFC в многопотоковых приложениях, вообще нельзя работать с классами этой библиотеки в потоках, созданных иными средствами, нежели описаны здесь. То есть для создания потоков нельзя пользоваться функциями Win32 API. Связано это с тем, что при создании потока с помощью функции CreateThread класса CWinThread осуществляется инициализация внутренних переменных, необходимых для правильной работы многопотоковых MFC-приложений.

Списки дескрипторов объектов Windows

Когда речь идет о вспомогательных классах, то контроль совместного использования одних и тех же объектов в различных потоках исключается. Однако для объектов классов, инкапсулирующих объекты Windows, такой контроль проводится и вот каким способом.

Каждое приложение имеет два списка, содержащих указатели на объекты классов библиотеки MFC, которые ассоциированы с конкретными объектами Windows. В одном списке хранятся указатели на постоянные объекты, а во втором — на временные. Мы уже сталкивались с термином "временный объект", когда обсуждали контексты устройств. Такие объекты создаются как бы из объектов Windows, а не от классов C++, при помощи функции FromHandle. Перечислим, какие объекты Windows, описывающие их дескрипторы и связанные с ними классы, хранятся в списках:

Объект	Дескриптор	Класс
Окно	HWND	CWnd и производные от него
Контекст устройства	HDC	CDC и производные от него
Меню	HMENU	CMenu
Карандаш	HPEN	CGdiObject и CPen
Кисть	HBRUSH	CGdiObject и CBrush
Шрифт	HFONT	CGdiObject и CFont
Битовый массив	HBITMAP	CGdiObject и CBitmap
Палитра	HPALETTE	CGdiObject и CPalette
Регион	HRGN	CGdiObject и CRgn
Список изображений	HIMAGELIST	CImageList
Сокет	SOCKET	CSocket

При чем тут объекты, списки? На самом деле для многопотоковых приложений в принципе неверным является утверждение, что списки объектов относятся ко всему приложению. Эти списки создаются и поддерживаются отдельно для каждого потока, созданного при помощи класса CWinThread, и располагаются в локальном пространстве потока (thread local storage, TLS). Сделано это для дополнительной защиты и независимости потоков друг от друга. А попытки нарушения могут быть предприняты в самый неожиданный момент из самого неожиданного места в программе, а именно из недр самой библиотеки MFC, например, в том случае, если в какую-либо функцию передается указатель на объект, созданный в другом потоке. Если посмотреть исходные тексты библиотеки, то можно увидеть, что примерно треть кода — это

всевозможные проверки. Для объектов классов, связанных с объектами Windows, эти проверки включают проверку потока, в котором идет обращение к объекту, по наличию указателя на него в одном из списков потока.

Примечание

Такая проверка проводится только для отладочной версии приложения.

Дело, конечно, не в проверках. Как уже упоминалось, в многопоточных приложениях не следует использовать один и тот же объект класса в различных приложениях. Всегда определять и контролировать такое использование код библиотеки не в состоянии. Но когда такая проверка не требует больших накладных расходов, особенно при отладке, она производится.

Что нельзя делать, теперь мы знаем, а что же можно делать, вернее, как обойти указанное ограничение? Таких приемов несколько.

- Во-первых, можно передать дескриптор объекта вместо указателя на объект соответствующего класса. Впоследствии на его основе можно создать либо постоянный объект при помощи функции Attach (после окончания использования объекта класса его следует "отключить" от объекта Windows (функция Detach), иначе может произойти ошибка времени исполнения — объект Windows остается в списке), либо временный при помощи функции FromHandle.
- Во-вторых, можно опять же на основании переданного дескриптора вызывать функции Win32 API, использующие дескриптор для определения объекта. Использование этого метода может нарушить работу приложения, если соответствующий объект библиотеки MFC другого потока исключает прямое использование дескриптора. Это, например, относится к контекстам устройств.
- В-третьих, для окон, а именно они составляют львиную долю всех длительно используемых в работе объектов, можно создать пользовательские сообщения, которые после посылки автоматически будут оказываться в очереди именно того потока, во время выполнения которого соответствующее окно было создано. В большинстве случаев для посылки сообщения следует использовать функцию Win32 API ::PostMessage. Естественно, что класс окна должен иметь функции обработки таких сообщений.

Graphics — пример приложения

Приложение Graphics создает 10 потоков, каждый из которых имеет отличающийся относительный приоритет. Каждый поток случайным образом определяет новое место для небольшого объекта-квадратика в рамках главного окна приложения. Это приложение демонстрирует следующие рассмотренные нами особенности создания многопотоковых приложений:

- Создание потоков, по форме являющихся интерфейсными, а по сути рабочими, т. к. они не обслуживают сообщения, — все действия производятся в цикле функции Run
- Отключение возможности уничтожения объекта класса, производного от класса CWinThread, чтобы объект класса был доступен перед завершением приложения
- Передача дескриптора контекста устройства в качестве параметра конструктору класса потока и создание нового объекта CDC на основе этого дескриптора
- Осуществление взаимоисключения доступа при реализации графических операций, т. к. контекст устройства, участвующий в процессе рисования, не является автоматически перераспределяемым ресурсом
- Использование обычной переменной для индикации необходимости завершения созданных потоков без реализации взаимоисключения

Рис. 31.1. Главное окно приложения Graphics

```
// ФАЙЛ Graphics.cpp
// Количество создаваемых потоков
#define THREAD_NUMBER  10

// Возможные относительные приоритеты
int cs_nPrior[] = {
  THREAD_PRIORITY_IDLE,
  THREAD_PRIORITY_LOWEST,
  THREAD_PRIORITY_BELOW_NORMAL,
  THREAD_PRIORITY_NORMAL,
  THREAD_PRIORITY_ABOVE_NORMAL,
  THREAD_PRIORITY_HIGHEST,
  THREAD_PRIORITY_TIME_CRITICAL };
CThread *cs_pThread[THREAD_NUMBER];
```

```
BOOL CGraphicsApp::InitInstance()
{
  ...
 // Создание главного окна приложения
  CGraphicsFrame *pWnd = new CGraphicsFrame;
  m_pMainWnd = pWnd;
  int nSizeX = (int)(GetSystemMetrics(SM_CXMAXIMIZED) * 0.3);
  int nSizeY = (int)(GetSystemMetrics(SM_CYMAXIMIZED) * 0.4);
  DWORD dwStyle = (WS_OVERLAPPEDWINDOW & ~WS_THICKFRAME);
  dwStyle &= ~ WS_MAXIMIZEBOX;
  pWnd->Create(NULL, _T("Graphics -- Multithreading"),
    dwStyle, CRect(0, 0, nSizeX, nSizeY), NULL,
    MAKEINTRESOURCE(IDR_GRAPHICS));
  pWnd->CenterWindow();
  pWnd->ShowWindow(m_nCmdShow);
  pWnd->UpdateWindow();
  // Инициализируем переменную, по которой проверяется
  // возможность продолжения выполнения потоков
  m_nDone = FALSE;

  CDC *pDC = pWnd->GetDC();

  // Количество относительных приоритетов
  int nPrior = sizeof(cs_nPrior) / sizeof(int);

  for (int i = 0; i < THREAD_NUMBER; i++)  {
    COLORREF cr = RGB(12 * i, 12 * i, 12 * i);

    // Создаем новый объект потока, передавая указатель
    // на объект окна (нужно для интерфейсных потоков,
    // в нашем случае этот объект не используется, кроме как
    // в главном потоке), дескриптор контекста устройства и цвет
    cs_pThread[i] = new CThread(pWnd, pDC->GetSafeHdc(), cr);
    if (cs_pThread[i] == NULL)
      break;
    ASSERT_VALID(cs_pThread[i]);
    // Для исключения автоудаления объекта потока
    cs_pThread[i]->m_bAutoDelete = FALSE;
```

```
    // При создании потока определяем его как изначально
    // приостановленный, чтобы перед началом его работы
    // настроить относительный приоритет
    if (!cs_pThread[i]->CreateThread(CREATE_SUSPENDED))    {
      delete cs_pThread[i];
      cs_pThread[i] = NULL;
      break;
    }
    VERIFY(cs_pThread[i]->SetThreadPriority(
      cs_nPrior[(i * nPrior) / THREAD_NUMBER]));
    // Возобновляем поток
    cs_pThread[i]->ResumeThread();
  }
  return TRUE;
}
void CGraphicsApp::CancelThreads()
{
  // Для того чтобы потоки могли продолжить выполнение и обнаружить
  // необходимость завершения, их необходимо возобновить; если они не
  // были приостановлены, ничего не произойдет
  OnResume();
  // Теперь все потоки завершатся
  m_nDone = TRUE;
  // Чтобы исключить преждевременное завершение
  // главного потока, дождемся завершения всех остальных
  for (int i = 0; i < THREAD_NUMBER; i++)   {
    if (cs_pThread[i] != NULL)    {
      WaitForSingleObject(cs_pThread[i]->m_hThread, INFINITE);
      delete cs_pThread[i];
    }
  }
}
```

Обычно для приостановки потоков следует использовать функцию SuspendThread, однако в примере для этого используется критическая секция, т. к. все дополнительные потоки приложения осуществляют графический вывод только тогда, когда контекст устройства никем больше не используется. Это дает возможность блокировать все потоки, имитируя такой доступ.

```
// Блокируем и возобновляем потоки путем входа или
// выхода из критической секции соответственно
```

```
void CGraphicsApp::OnStop()
{
  CThread::m_csGDILock.Lock();
}
void CGraphicsApp::OnResume()
{
  CThread::m_csGDILock.Unlock();
}
// Файл GraphicsFrm.cpp
BEGIN_MESSAGE_MAP(CGraphicsFrame, CFrameWnd)
  //{{AFX_MSG_MAP(CGraphicsFrame)
  ON_WM_PAINT()
  ON_WM_CLOSE()
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()
void CGraphicsFrame::OnPaint()
{
  CPaintDC dc(this);
  CRect rect;
  GetClientRect(&rect);
  CBrush brush(::GetSysColor(COLOR_WINDOW));
  dc.FillRect(&rect, &brush);
}
void CGraphicsFrame::OnClose()
{
  // Чтобы контекст устройства в котором происходит рисование потоками,
  // существовал, потоки следуют завершить до того, как будет закрыто
  // главное окно приложения. Если завершать потоки в функции
  // ExitInstance объекта-приложения, возможно нарушение процесса
  // рисования, т.к. дескриптор контекста в этом случае будет
  // неверным, и вывод будет осуществлен "прямо" на экран
  ((CGraphicsApp *)AfxGetApp())->CancelThreads();

  CFrameWnd::OnClose();
}
```

Класс CThread на основе переданного и сохраненного дескриптора контекста устройства создает соответствующий класс библиотеки MFC. Именно так и следует делать, однако разработчик должен гарантировать, что дескриптор будет действительным все время, пока он используется в созданных потоках!

```
// Файл Thread.h
class CThread: public CWinThread
{
...
public:
  static CCriticalSection m_csGDILock;
private:
  CRect m_Rect;
  HDC m_hDC;
  CPoint m_pt, m_offset;
  CBrush m_Brush;
public:
  //{{AFX_VIRTUAL(CThread)
  public:
  virtual BOOL InitInstance();
  virtual int ExitInstance();
  virtual int Run();
  //}}AFX_VIRTUAL
...
};
// Файл Thread.cpp
// Критическая секция для взаимоисключения доступа к контексту устройства
CCriticalSection CThread::m_csGDILock;
IMPLEMENT_DYNAMIC(CThread, CWinThread)
```

Конструктор осуществляет все необходимые инициализации, но т. к. код конструктора выполняется в рамках главного потока приложения, в конструкторе нельзя (не имеет смысла) выполнять действия, которые следует совершать в рамках создаваемого потока. Именно поэтому назначение "зерна" для генератора случайных чисел производится в функции Run, а не в конструкторе класса. Кстати, обратите внимание, что это назначение (вызов функции srand) производится для каждого потока в отдельности. Дело в том, что, в отличие от библиотеки классов MFC, стандартная библиотека runtime, используемая для многопотоковых приложений, создана таким образом, что все ее глобальные данные копируются при создании для каждого нового потока. При этом изменение этих данных оказывает влияние только на тот поток, которому они принадлежат.

```
CThread::CThread(CWnd* pWnd, HDC hDC, COLORREF &cr)
{
  m_pMainWnd = pWnd;
```

```
  m_pMainWnd->GetClientRect(&m_Rect);
  m_hDC = hDC;
  m_pt.x = m_Rect.Width() / 2;
  m_pt.y = m_Rect.Height() / 2;
  m_offset.x = 10;
  m_offset.y = 10;
  m_Brush.CreateSolidBrush(cr);
}
int CThread::Run()
{
  CDC dc;
  dc.Attach(m_hDC);
  CBrush brush(::GetSysColor(COLOR_WINDOW));
  // Назначаем "зерно" для генератора случайных чисел.
  // Это нужно делать для каждого потока в отдельности,
  // причем только тогда, когда именно он выполняется
  srand((unsigned)this);
  // Работаем, пока не изменится проверяемое значение
  while (!((CGraphicsApp *)AfxGetApp())->m_nDone)  {
    CPoint pt = m_pt;
    // Определяем новое положение
    for (;;)    {
      pt += m_offset;
      if (m_Rect.PtInRect(pt) && (rand() % 1000) < 900)      {
        if (m_offset.x < 0)
          m_offset.x -= 1;
        else if (m_offset.x > 0)
          m_offset.x += 1;
        if (m_offset.y < 0)
          m_offset.y -= 1;
        else if (m_offset.y > 0)
          m_offset.y += 1;
        break;
      }
      pt = m_pt;
      CPoint offset = m_offset;
      while (offset == m_offset ||
        offset.x == 0 && offset.y == 0 ||
```

```
        offset.x == -m_offset.x && offset.y == -m_offset.y)      {
        offset.x = ((rand() % 11) - 5);
        offset.y = ((rand() % 11) - 5);
      }
      m_offset = offset;
    }
    // Блокируем доступ к контексту устройства или ожидаем
    // возможность доступа к контексту устройства и получив его
    // блокируем доступ для других потоков
    CThread::m_csGDILock.Lock();
    {
      CRect rect(m_pt.x-4, m_pt.y-4, m_pt.x+5, m_pt.y+5);
      dc.FillRect(&rect, &brush);
      m_pt = pt;
      CBrush *pBrush = dc.SelectObject(&m_Brush);
      dc.Rectangle(m_pt.x-4, m_pt.y-4, m_pt.x+5, m_pt.y+5);
      dc.SelectObject(pBrush);
      // Windows группирует вызовы функций рисования для каждого из
      // потоков, а так как мы разделяем контекст устройства,
      // необходимо перед разрешением доступа к контексту других
      // потоков выполнить все накопленные действия
      GdiFlush();
    }
    CThread::m_csGDILock.Unlock();
    // Небольшая задержка - пусть другие поработают
    Sleep(50);
  }
  // Необходимо отсоединить дескриптор от объекта класса,
  // т.к. не только этот класс связан с контекстом
  dc.Detach();
  return 0;
}
```

Попытка задания различных приоритетов для потоков, судя по работе приложения, никак не сказывается на скорости их выполнения. Скорее всего, это связано с тем, что в результате реализации взаимоисключения происходит синхронизация всех потоков, так что их скорость определяется скоростью самого медленного потока — приоритетность не является определяющим фактором для выбора следующего потока после выхода текущего из критической секции.

Использование классов объектов синхронизации

Windows предоставляет для использования четыре основных типа объектов синхронизации: критические секции, взаимоисключения, события и семафоры. Несмотря на их количество, все они должны быть реализованы совершенно одинаково и представляют собой совокупность защищенной переменной и списка. На самом деле так реализуются классические семафоры.

Примечание

На самом деле как такового объекта "критическая секция" нет. Критическая секция представлена в программе структурой, одно поле которой содержит дескриптор семафора. Таким образом, создание критической секции просто приводит к созданию семафора.

Семафоры (речь идет в данном случае не об объектах Windows) бывают двух типов: двоичные и считающие. *Двоичные* семафоры могут принимать только два значения (отсюда и название), а *считающие* используют защищенную переменную в качестве счетчика.

Список используется для хранения информации о потоках, ожидающих появления ненулевого значения защищенной переменной. Вообще говоря, список может быть организован различными способами, т. е. поддерживать различные методики организации элементов, например, в соответствии с методикой FIFO (первый пришедший обслуживается первым), что является наиболее справедливой методикой реагирования на события. Судя по всему, Windows никак не упорядочивает элементы списка — первым активизируется тот поток, находящийся в списке, который первым должен получить следующий квант времени. При этом по не совсем понятным причинам таким потоком может быть поток, имеющий не самый высокий приоритет. Видимо, для улучшения пропускной способности и отсутствия бесконечного откладывания по истечении кванта времени высокоприоритетные потоки дают возможность поработать остальным потокам, другое дело, что последние будут практически сразу же вытеснены.

Примечание

Эти выводы получены в результате анализа работы довольно сложной тестовой программы, обзор которой выходит за рамки данной книги. Тестовые испытания проводились для Windows 95/98 — возможно, Windows NT ведет себя несколько иначе.

В следующей таблице приведены объекты синхронизации и соответствующие им классы библиотеки MFC.

Объекты	Класс	Тип	Назначение
Критические секции	CCriticalSection	Двоичный семафор	Взаимоисключение в рамках одного процесса
Взаимоисключения	CMutex	Двоичный семафор	Межзадачное взаимоисключение
События	CEvent	Двоичный семафор	Сигнализация, в том числе межзадачная
Семафоры	CSemaphore	Считающий семафор	Ограничение доступа, возможно использование и для сигнализации, в том числе между различными процессами

Примечание

Мы все время используем термин "процесс", однако взаимодействие между процессами будем определять как "межзадачное".

Объект синхронизации "событие" все-таки несколько отличается от обычных семафоров из-за того, что он позволяет регулировать процесс изменения своего состояния, а также может активизировать сразу все потоки, ожидающие смены его состояния.

Создание класса разделяемых объектов

Все классы, объекты которых создаются и используются в многопотоковых приложениях, можно разделить на две большие группы:

- Классы, управляющие и синхронизирующие работу нескольких потоков
- Классы, используемые для создания нескольких объектов, в различных потоках, но при этом использующих общие ресурсы, что требует для всех этих объектов класса самосинхронизации

Классы синхронизирующих объектов

Объекты, синхронизирующие работу нескольких потоков, содержат функции, которые могут вызываться из различных потоков для доступа к данным объекта. Естественно, в этом случае просто необходимо осуществлять взаимоисключение такого доступа. Характерным примером могут служить приложения типа "читатели и писатели". В таких приложениях осуществляется параллельный доступ к некоторому буферу, который может и должен быть реализован как класс, синхронизирующий потоки-читатели и потоки-писатели.

Вот реализация класса кольцевого буфера, решающая задачи синхронизации.

```
// Первый вариант
class CBuffer{
public:
  CBuffer(): m_nHead(0), m_nTail(0) {}
  TCHAR ReadChar();
  void WriteChar(TCHAR cChar);
private:
  TCHAR m_Buffer[BUF_LEN];
  int m_nHead;  // "Куда" писать
  int m_nTail;  // "Откуда" читать
// Объект "критическая секция" для организации взаимоисключения
CCriticalSection m_Lock;
};
TCHAR CBuffer::ReadChar()
{
  TCHAR cChar;
  // Вход в критическую секцию
  m_Lock.Lock();
  if (m_nTail == m_nHead)        // Работа с разделяемыми
    cChar = (TCHAR)-1;           // данными
  else
  {
    cChar = m_Buffer[m_nTail++]; // Возможно, запись в буфер и
    m_nTail %= BUF_LEN;          // не требует взаимоисключения
  }
  // Выход из критической секции
  m_Lock.Unlock();
  return cChar;
}
void CBuffer::WriteChar(TCHAR cChar)
{
  // Вход в критическую секцию
  m_Lock.Lock();
  m_Buffer[m_nHead++] = cChar;   // Работа с разделяемыми
  m_nHead %= BUF_LEN;            // данными
  // Выход из критической секции
  m_nLock.Unlock();
}
```

Обратите внимание, что по сути в данной конкретной реализации защищается не столько сам буфер, сколько элемент m_nHead, который при выполнении записи в буфер изменяется, а при чтении из буфера его значение проверяется, для того чтобы можно было определить, есть ли символы в буфере (не пуст ли он?). В данной реализации предполагается, что буфер не будет переполнен, хотя взаимоисключение и не обеспечивает от этого никакой защиты.

Собственно защита на все случаи жизни от переполнения буфера может быть только одна — динамическое изменение (увеличение) размера буфера. Если, однако, условия использования кольцевого буфера гарантируют обязательное и практически немедленное извлечение данных из буфера, то можно процесс записи в буфер при его заполнении слегка притормозить. Опять же это можно делать только тогда, когда нет угрозы потери данных потока-писателя.

Примечание

Исчерпание системных ресурсов, в данном случае памяти, нами, естественно, не рассматривается. Динамическое увеличение размера кольцевого буфера будет приведено в примере Terminal, в котором рассматривается работа с последовательным портом.

```
// Второй вариант
class CBuffer{
public:
  CBuffer(): m_nHead(0), m_nTail(0) {}
  TCHAR ReadChar();
  void WriteChar(TCHAR cChar);
private:
  TCHAR m_Buffer[BUF_LEN];
  int m_nHead;  // "Куда" писать
  int m_nTail;  // "Откуда" читать
// Объект "критическая секция" для организации взаимоисключения
  CCriticalSection m_Lock;
// Объект "событие" для сигнализации о
// наличии свободного места в буфере
 CEvent m_Free;
};
TCHAR CBuffer::ReadChar()
{
  TCHAR cChar;
  // Вход в критическую секцию
```

```
  m_Lock.Lock();
  if (m_nTail == m_nHead)        // Работа с разделяемыми
    cChar = (TCHAR)-1;           // данными
  else  {
    cChar = m_Buffer[m_nTail++]; // Теперь переменная m_nTail также
    m_nTail %= BUF_LEN;          // является разделяемой
  }
  // Сигнализируем о том, что в буфере есть хотя бы одно свободное место
  m_Free.SetEvent();
  // Выход из критической секции
  m_Lock.Unlock();
  return cChar;
}
void CBuffer::WriteChar(TCHAR cChar)
{
  // Вход в критическую секцию
  m_Lock.Lock();
  // Заранее проверяем, не будет ли буфер полностью заполнен после
  // дозаписи в него; это необходимо проделать, т.к. в противном
  // случае невозможно будет отличить заполненный буфер от пустого -
  // и в том и в другом случае m_nHead = m_nTail
  if (((m_nHead+1) % BUF_LEN) == m_nTail)  {
    // Буфер полон!!!
    // Выходим из критической секции
    m_Lock.Unlock();
    // Ожидаем освобождения хотя бы одного элемента, т.е. будет
    // достаточно одного вызова функции ReadChar
    m_Free.Lock();
    // Опять входим в критическую секцию
    m_Lock.Lock();
  }
  // Сбрасываем сигнал об освобождении элемента буфера,
  // иначе в следующий раз, когда буфер будет полон,
  // все равно поступит сигнал об освобождении элемента,
  // т.к. каждый раз при чтении компонент класса m_Free
  // устанавливается (см. функцию ReadChar)
  m_Free.ResetEvent();
  m_Buffer[m_nHead++] = cChar;    // Работа с разделяемыми
  m_nHead %= BUF_LEN;             // данными
  // Выход из критической секции
```

```
  m_Lock.Unlock();
}
```

В случае возникновения перспективы переполнения буфера функция записи дает возможность другому потоку осуществить чтение из массива и, естественно, освободить элемент массива. Для этого выполняется выход из критической секции, но функция не завершается, а ожидает этого самого освобождения и только затем производит запись в буфер. Если при отсутствии места в буфере просто завершить функцию, проинформировав вызывающую функцию о невозможности записи, то для исключения потери данных придется либо разрабатывать механизм диагностики освобождения массива, который будет скорее всего довольно громоздким и запутанным, либо осуществлять повторный вызов функции записи до тех пор, пока запись не будет действительно осуществлена. Последний вариант, как минимум, будет отнимать вычислительные ресурсы, так что пусть уж лучше операционная система временно заблокирует поток без потери общей производительности.

Вышеприведенные примеры можно улучшить за счет использования возможностей языка C++. В частности, при выходе из области видимости для объектов классов вызываются деструкторы. Этот факт позволяет использовать классы, обеспечивающие синхронизацию (аналоги ожидающих функций), так что выход, например, из критической секции будет производиться автоматически без лишних усилий. Вот как может выглядеть функция записи в кольцевой буфер в свете последних высказываний:

```
void CBuffer::WriteChar(TCHAR cChar)
{
  // Создание объекта, обеспечивающего синхронизацию,
  // с одновременным ожиданием возможности входа в критическую секцию
  CSingleLock Lock(&m_Lock, TRUE);
  m_Buffer[m_nHead++] = cChar;   // Работа с разделяемыми
  m_nHead %= BUF_LEN;            // данными
  // Выход из критической секции будет осуществлен сразу
  // после завершения функции при выходе из области видимости,
  // при вызове деструктора класса CSingleLock (естественно,
  // возможен и явный вызов функции Unlock класса CSingleLock)
}
```

Примечание

Следует иметь в виду, что фигурные скобки, определяющие область видимости, можно располагать в любом месте функции. Для этого не требуются ни операторы условия, ни операторы цикла.

В отладочной версии приложения использование класса CSingleLock увеличивает время выполнения соответствующих участков кода на два-три порядка за

счет большого количества довольно громоздких проверок значений и типов, например: ASSERT(pObject->IsKindOf(RUNTIME_CLASS(CSyncObject))), что может существенно сказываться на поведении приложения — ряд операций может выполняться как бы рывками. (Приложение-пример этой главы — терминальная программа в отладочной версии как раз так себя и ведет.) Однако это практически не сказывается на рабочей версии примера.

Классы самосинхронизирующихся объектов

Классы, требующие самосинхронизации доступа, должны иметь общий для них синхронизирующий объект. Естественно, что не следует для синхронизации доступа использовать внешний объект или, тем более, элемент другого класса. Наилучшим вариантом будет создание статического, т. е. одного на все объекты класса, элемента, ответственного за синхронизацию. Кстати сказать, мы уже приводили пример такой реализации. В приложении Graphics класс дополнительных потоков имеет в своем составе статический элемент "критическая секция":

```
static CCriticalSection m_csGDILock;
```

Этот объект используется всеми экземплярами (объектами) класса для организации исключительного доступа к контексту устройства при графическом выводе.

Использование классов объектов синхронизации

В начале раздела мы сказали, что все объекты синхронизации имеют одну и ту же природу. Зачем же Windows поддерживает сразу несколько объектов синхронизации? Ответ прост. Каждый из объектов синхронизации используется для решения различных задач, многообразие таких объектов позволяет упростить их использование.

Отметим сразу, что объекты синхронизации используются прежде всего для того, чтобы синхронизировать доступ к разделяемым ресурсам. Даже если на первый взгляд такого доступа и нет, необходимо помнить, что ресурс — понятие достаточно широкое, например, если поток просто получает возможность продолжить выполнение, это приводит как раз к получению доступа к вычислительным ресурсам.

Вот несколько правил, следуя которым при решении конкретной задачи можно осуществить наиболее приемлемый выбор объекта синхронизации из тех, что предлагает Windows:

- Если приложение ожидает возникновения некоторого события, в том числе, перед тем как получить доступ к некоторым данным, следует использовать объект "событие" (event), возможно при помощи класса библиотеки MFC CEvent. Возможно — потому, что в ряде случаев приходится использовать функции Win32 API, которые ничего про этот класс "не знают" и работают только с дескрипторами объекта "событие". Это преж-

де всего относится к асинхронному вводу/выводу и работе с коммуникационными устройствами.

Примечание

Если для решения большинства задач можно использовать семафоры как универсальный объект синхронизации, то для реализации асинхронного ввода/вывода и работы с коммуникационными устройствами Windows предусматривает только объекты "событие".

- Если к некоторому ресурсу может быть осуществлен одновременный доступ сразу нескольких потоков, но при этом все-таки необходимо их число ограничить, следует использовать объект "семафор" или класс библиотеки MFC CSemaphore.
- Если к некоторому ресурсу, в отличие от предыдущего случая, нельзя разрешить доступ более, чем одному потоку, то для межзадачной синхронизации следует использовать объект "взаимоисключение" или класс библиотеки MFC CMutex, а для взаимоисключения в рамках одного процесса — объект "критическая секция" или класс CCriticalSection.

Что касается того, как использовать объекты синхронизации, то библиотека классов MFC предоставляет более широкие возможности, нежели Win32 API, прежде всего, обеспечивая большую гибкость и варианты использования. Несмотря на то что все равно так или иначе для синхронизации используются ожидающие функции, использование классов библиотеки позволяет практически на равных применять как функции-члены непосредственно классов синхронизации, так и объекты класса CSingleLock. Естественно, что при одновременном использовании нескольких объектов синхронизации можно применять только класс CMultiLock.

Классы объектов синхронизации

В основе классов (кроме базового для всех них) лежат объекты ядра операционной системы, с помощью которых перед тем, как заблокировать себя, поток сообщает системе, какое событие должно произойти, чтобы его исполнение можно было возобновить. Для использования всех классов объектов синхронизации в проект необходимо включить заголовочный файл AFXMT.H. Проще и надежнее это сделать в файле STDAFX.H.

Класс CSyncObject — базовый класс объектов синхронизации

Класс CSyncObject является базовым абстрактным классом для всех классов, обеспечивающих синхронизацию. Он предоставляет базовые функции, которые позволяют

использовать большинство возможностей соответствующих объектов синхронизации. Одни и те же наименования функций в производных классах позволяют использовать схожий синтаксис для решения различных задач. Единственным исключением является класс CEvent, который несколько иначе интерпретирует эти функции, что вносит некоторую путаницу.

Объекты описываемого класса никак не связаны с объектами синхронизации Windows. Хотя именно этот класс содержит элемент, сохраняющий дескриптор взаимоисключений, событий и семафоров. Этот элемент является общедоступным (public), но тем не менее не представлен в документации:

HANDLE CSyncObject::**m_hObject**

Однако для того чтобы этот дескриптор можно было бы использовать в ожидающих функциях, вовсе не следует использовать его непосредственно. Доступ же к дескриптору бывает необходим, например, при реализации асинхронного ввода/вывода, когда один из объектов синхронизации не является объектом класса.

Класс CSyncObject предусматривает функцию преобразования объекта к дескриптору, что упрощает программирование:

operator CSyncObject::**HANDLE**( )

Вот пример использования такого преобразования, которое происходит неявно.

```
// Объявление объекта "событие"
  CEvent m_Event;
  // Структура и объект "событие" для реализации
  // асинхронного ввода/вывода
  OVERLAPPED o;
  o.hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
  // Массив дескрипторов, где второй элемент
  // неявно относится к типу HANDLE
  HANDLE Handles[] = { o.hEvent, m_Event };
  // Начало чтения
  ReadFile(hFile, szBuffer, 100, &dwBytes, &o);
  // Ожидание завершения чтения, либо возникновения события m_Event
  WaitForMultipleObjects(2, Handles, FALSE, INFINITE);
```

К сожалению, класс CSyncObject не содержит функций типа Attach и Detach, позволяющих конструировать объекты соответствующих классов на основе существующих объектов Windows. Хотя это вполне объяснимо, однако отсутствие такой возможности исключает использование классов объектов синхронизации еще и тогда, когда необходимо произвести дублирование объекта при помощи функции ::DuplicateHandle — работать с копией дескриптора приходится при помощи функций Win32 API.

Поскольку все объекты, кроме критических секций, могут быть поименованы, класс CSyncObject в качестве параметра принимает возможное имя нового объекта. Однако его можно использовать только в отладочной версии для того, чтобы сбрасывать в дамп заданное имя. Естественно, что реальные объекты синхронизации создаются соответствующими конструкторами производных классов.

CSyncObject::**CSyncObject**(LPCTSTR pstrName)

По завершении использования объекта синхронизации обязательно следует его уничтожить (освободить). В Win32 API для этого предусмотрена функция ::CloseHandle, которая вызывается в деструкторе класса CSyncObject, и нам об объекте синхронизации беспокоиться уже не следует.

Все описанное выше используется автоматически без участия разработчика и приведено лишь для ознакомления. Следующие две функции, напротив, чрезвычайно важны, их применение требует глубокого понимания того, что же они на самом деле делают.

virtual BOOL CSyncObject::**Lock**(DWORD dwTimeout = INFINITE) —

осуществляет блокирование вызвавшего ее потока до того момента, пока связанный с объектом класса объект синхронизации не будет установлен (signaled) или не истечет заданный параметром dwTimeout в *миллисекундах* промежуток времени. По умолчанию этот промежуток принимает значение INFINITE, что исключает завершение функции до возникновения ожидаемого события. Кстати, все производные классы, кроме CCriticalSection, не имеют своих функций блокировки.

Функция возвращает значение FALSE в случае истечения заданного кванта времени. К сожалению, это же значение возвращается и в результате возникновения ошибки.

В отличие от рассмотренной, функция освобождения класса CSyncObject "ничего не делает". Она имеет две версии:

virtual BOOL CSyncObject::**Unlock**(LONG lCount, LPLONG lpPrevCount = NULL)

и

virtual BOOL CSyncObject::**Unlock**( ) —

параметры первой версии функции классом CSyncObject не используются, и функция всегда возвращает TRUE. Вторая версия является чисто абстрактной и для всех производных классов требуется ее переопределение.

Надо сказать, что не все классы выполняют действия по освобождению в соответствующих функциях своих классов. Например, класс CEvent "ничего не делает" в функции Unlock, просто возвращая значения TRUE. Это вполне объяснимо потому, что основной смысл класса CEvent и объекта "событие" в сигнализации, а остальных — в ограничении доступа. Именно поэтому освобождение для остальных классов заключается в переводе объекта синхронизации в установленное состояние. Кроме того, первая версия функции позволяет семафорам изменять (увеличивать) содержимое счетчика более чем на единицу.

Примечание

Вообще говоря, наименование и назначение этих функций может вызвать некоторую путаницу, особенно если до этого в основном использовались функции Win32 API. Действительно, ожидание "установки" некоторого объекта требует этой самой установки. Наиболее соответствует этому правилу объект "событие". Для установки используется функция SetEvent. С другой стороны, связка блокирование-освобождение скорее соответствует не потоку, а разделяемому ресурсу — функция Lock блокирует к нему доступ из других потоков, а функция Unlock разрешает такой доступ. Именно поэтому эти две функции должны использоваться совместно.

К сожалению, это верно не для всех классов, причем если результат действия функции Lock полностью совпадает для всех четырех классов, то функция Unlock для класса CEvent вообще "ничего не делает" (об этом позже). При этом, чтобы хоть как-то использовать объекты этого класса, требуется пользоваться специфичными для этого класса функциями, при том, что все остальные классы практически идентичны и работа с ними полностью унифицирована.

И последнее. Для объектов-взаимоисключений используются понятия "захвачен" (сброшен) и "освобожден" (установлен).

Критическая секция (класс CCriticalSection)

Класс CCriticalSection обеспечивает представление в приложении объекта синхронизации "критическая секция", который позволяет организовать взаимоисключающий доступ к некоторому ресурсу или участку кода в рамках одного приложения. Этот объект не может быть именован и именно из-за этого не может использоваться для межзадачной синхронизации. (Для межзадачной синхронизации доступа следует воспользоваться классом CMutex.)

Среди всех объектов синхронизации наиболее выделяется объект "критическая секция", не только тем, что он описывается структурой, одно поле которой является дескриптором семафора, используемого для организации взаимоисключения, но и тем, что все другие объекты могут использоваться для межзадачной синхронизации. Достигается это тем, что эти объекты могут быть поименованы — при помощи символьного имени они могут быть найдены различными процессами для дальнейшего совместного использования.

Организация класса полностью соответствует схеме блокирован-освобожден (lock-unlock), предоставляемой базовым классом CSyncObject. Участок кода, доступ к которому должен быть регламентирован, следует помещать между последовательными вызовами функций Lock и Unlock этого класса.

Обратите внимание, что нельзя использовать ни объект класса CCriticalSection совместно с классом CMultiLock, ни объект "критическая секция" в функциях Win32 API ::WaitForMultipleObjects или ::MsgForMultipleObjetcs.

Для создания объекта класса CCriticalSection используется конструктор без параметров:

CCriticalSection::**CCriticalSection**( )

Функция Lock блокирует участок кода. В дополнение к версии без параметров функция Lock имеет вторую версию, а версии функции, где значение задержки задано по умолчанию, напротив, нет.

BOOL CCriticalSection::**Lock**( )

и

BOOL CCriticalSection::**Lock**(DWORD dwTimeout)

Никаких временных ограничений на ожидание освобождения критической секции в Win32 API не предусмотрено. Чтобы исключить конфликт наименования функций, в библиотеку классов MFC и были включены обе эти версии функций. Таким образом исключается вызов функции Lock базового класса. Необходимо помнить, что несмотря на наличие параметра dwTimeout, создаваемые приложения не должны делать никаких предположений о возможности задания задержки. Если все-таки необходимо отслеживать время, следует вместо критической секции использовать объект "взаимоисключение" и класс CMutex.

Для завершения исключительного доступа предназначена функция Unlock, а удаление самого объекта "критическая секция" выполняется деструктором.

Взаимоисключение (класс CMutex)

Класс CMutex обеспечивает представление в приложении объекта синхронизации "взаимоисключение" (mutex — от mutually exclusion), который позволяет организовать взаимоисключающий доступ к некоторому ресурсу или участку кода как в рамках одного приложения, так и в случае нескольких процессов.

Правила использования класса CMutex практически полностью совпадают с правилами использования класса CCriticalSection. Отличие заключается лишь в том, что функция Lock этим классом не перегружается, т. к. объект синхронизации, лежащий в основе класса, может использоваться в ожидающих функциях без ограничений.

Перегруженная функция Unlock обеспечивает освобождение объекта "взаимоисключение" путем перевода его состояния в "установлен".

Наибольший интерес представляет конструктор класса.

CMutex::**CMutex**(BOOL bInitiallyOwn = FALSE, LPCTSTR lpszName = NULL, LPSECURITY_ATTRIBUTES lpsaAttribute = NULL) —

конструктор позволяет создавать не только поименованные объекты (параметр lpszName), но также задавать начальное состояние объекта (параметр bInitiallyOwn), и атрибуты защиты (параметр lpsaAttribute).

Значение параметра bInitiallyOwn, равное TRUE, позволяет потоку захватить объект "взаимоисключение", т. е. состояние объекта изначально "сброшен".

Атрибуты защиты имеют тот же смысл и то же назначение, что и для потоков (см. предыдущую главу).

В случае невозможности создать объект "взаимоисключение" конструктор генерирует исключение типа CResourceException. Чаще всего невозможность создания является следствием того, что в системе уже есть поименованный объект с тем же именем, которое задано параметром lpszName, однако этот объект другого типа — событие либо семафор.

Для того чтобы создавать объект класса CMutex, имеющий отличные от принятых по умолчанию параметры и который является при этом элементом какого-либо класса, следует в конструкторе этого класса вызвать конструктор объекта класса CMutex:

```
CMercuryClass::CMercuryClass(): m_Mutex(FALSE, _T("Mercury")) {}
```

Событие (класс CEvent)

Класс CEvent обеспечивает представление в приложении объекта синхронизации "событие", который позволяет организовать синхронизацию выполнения действий различных потоков как в рамках одного приложения, так и в случае нескольких процессов.

Использование класса CEvent в наибольшей степени отличается от использования других классов синхронизирующих объектов. Только функция Lock выполняет обычные действия и потому не перегружается.

По вполне понятным причинам перегруженная функция Unlock, напротив, "ничего не делает", т. к. описываемый объект синхронизации выполняет роль индикатора события и поэтому нельзя допустить его автоматическую генерацию, что реализовано для остальных классов. Если вы встретите описание этой функции в документации по библиотеке классов MFC, не делайте вывода о том, что результатом ее выполнения является освобождение объекта (в терминах синхронизирующих объектов это означает перевод объекта в состояние "установлен"). Это не так, поскольку иначе нарушало бы всю концепцию событий. Чтобы сомнений в сказанном не осталось, приведем полный текст этой функции:

```
BOOL CEvent::Unlock()
{
  return TRUE;
}
```

Конструктор класса CEvent отличается от конструктора класса CMutex только возможностью указания типа создаваемого объекта:

CEvent::**CEvent**(BOOL bInitiallyOwn = FALSE,
BOOL bManualReset = FALSE,LPCTSTR lpszName = NULL,
LPSECURITY_ATTRIBUTES lpsaAttribute = NULL)

Для этого используется параметр bManualReset. Объекты "событие" могут быть двух видов. Во-первых, объекты автоматического сбрасывания (bManualReset принимает значение FALSE), когда перевод в сброшенное состояние осуществляется в результате завершения одной из ожидающих функций в связи с установкой объекта "событие". Во-вторых, ручного сбрасывания (bManualReset принимает значение TRUE), когда только явный вызов функции ResetEvent (Win32 API или класса CEvent) переводит объект в состояние "сброшен". В последнем случае все потоки, дождавшиеся возникновения соответствующего события, получат возможность продолжить свою работу.

Если параметр bInitialyOwn имеет значение равное TRUE, созданный объект "событие" изначально находится в состоянии "установлен".

Для перевода объекта "событие" в состояние "установлен" следует использовать функцию SetEvent:

BOOL CEvent::**SetEvent**( )

После выполнения данной функции для объекта ручного сброса его состояние останется без изменений до тех пор, пока не будет выполнена функция ResetEvent, и все потоки, ожидающие событие, продолжат свое выполнение. Напротив, для объекта автоматического сброса перевод его в состояние "установлен" позволит продолжить свое выполнение лишь одному ожидающему потоку. Все остальные будут продолжать ждать возникновения следующего события. Если ни один поток не ожидает появления события, то его состояние для объектов "событие" обоих типов не изменится до выполнения функции ResetEvent или ожидающей функции.

Чтобы исключить ситуацию, когда только один из ожидающих потоков сможет продолжить выполнение в результате возникновения события и в то же время обеспечить автоматический сброс объекта "событие", следует использовать функцию:

BOOL CEvent::**PulseEvent**( )

Во время выполнения этой функции объект "событие" будет переведен сначала в состояние "установлен", а затем "сброшен", независимо от того, существует ли хотя бы один поток, ожидающий это событие. Для объектов автоматического сброса это единственное отличие — в любом случае только один ожидающий поток сможет продолжить свое выполнение. Для объектов ручного сброса все несколько иначе. Да, по завершении выполнения этой функции объект "событие" будет находиться в состоянии "сброшен", однако в момент установки состояния свое выполнение смогут продолжить все потоки, уже ожидавшие к этому моменту появления события.

Для сброса состояния объекта "событие" используется следующая функция

BOOL CEvent::**ResetEvent**( )

Она может оказаться полезной не только для объектов ручного сброса, но и для объектов сброса автоматического в том случае, если нет уверенности, что хотя бы один поток ожидал связанного с объектом события, и есть необходимость перевести объект в состояние "сброшен".

Семафор (класс CSemaphore)

Класс CSemaphore обеспечивает представление в приложении объекта синхронизации "семафор", который позволяет ограничить количество потоков, конкурирующих за доступ к некоторому ресурсу как в рамках одного приложения, так и в случае нескольких процессов.

Семафоры предназначены прежде всего для контроля доступа к набору разделяемых ресурсов. Текущее значение счетчика показывает число свободных на данный момент ресурсов. При завершении одной из ожидающих функций значение счетчика уменьшается на единицу. Как только значение счетчика становится равным 0 (состояние объекта "семафор" устанавливается в "сброшен"), все потоки, запрашивающие ресурс данного типа, переводятся в состояние ожидания освобождения хотя бы одного ресурса (тогда состояние объекта "семафор" переводится в "установлен") или истечения заданного промежутка времени. Максимальное количество ресурсов задается в конструкторе.

CSemaphore::**CSemaphore**(LONG lInitialCount = 1,
LONG lMaxCount = 1, LPCTSTR pstrName = NULL,
LPSECURITY_ATTRIBUTES lpsaAttributes = NULL) —

первые два параметра определяют максимальное количество ресурсов (параметр lMaxCount) и количество ресурсов, свободных в данный момент (параметр lInitialCount). Назначение остальных параметров аналогично их назначению для классов CMutex и CEvent.

Для увеличения значения счетчика используется функция Unlock:

virtual BOOL CSemaphore::**Unlock**( )

и

virtual BOOL CSemaphore::**Unlock**(LONG lCount,
LPLONG lprevCount = NULL) —

первая версия функции увеличивает значение счетчика на единицу, а вторая — на значение, заданное параметром lCount, при этом, задав в качестве второго параметра указатель на переменную типа LONG, можно получить предыдущее значение счетчика. Обе версии функции возвращают значение FALSE в результате возникновения ошибки или если значение счетчика может превысить максимальное возможное. При этом значение счетчика не изменится.

Большое количество исключений из правил, диктуемых набором и назначением функций-членов базового класса объектов синхронизации, требует некоторой систематизации. Для этого приведем таблицу, в которой представлены основные характеристики всех классов объектов синхронизации.

Класс	CCriticalSection	CMutex	CEvent	CSemaphore
Ожидание	Lock	Базовая	Базовая	Базовая
Установка объекта	Unlock	Unlock	SetEvent, PulseEvent	Unlock
Сброс объекта	—	—	ResetEvent	—
Wait-функции	Нет	Да	Да	Да
Класс CSingleLock	Да	Да	Да	Да
Класс CMultiLock	Нет	Да	Да	Да
Именование	Нет	Да	Да	Да

Классы, обеспечивающие синхронизацию

Для реализации синхронизации недостаточно только самих объектов синхронизации. Для этого Win32 API предоставляет набор ожидающих функций (критические секции используются несколько иначе). Все они работают с дескрипторами, описывающими соответствующие объекты синхронизации. Конечно, классы объектов синхронизации имеют возможность получать доступ к дескрипторам, но библиотека MFC обеспечивает разработчиков специальными классами, которые выполняют те же действия, что и ожидающие функции, кроме того, их использование имеет дополнительные преимущества:

- Использование классов обеспечивает обязательное выполнение некоторых действий, требуемых для реализации синхронизации. Поскольку для эффективного использования объекты описываемых классов должны создаваться в стеке, то выход из области действия приводит к выполнению деструктора, в котором производятся действия по смене состояния объектов синхронизации.
- Эти классы позволяют определять текущее состояние объекта синхронизации (хотя эта информация не всегда верна, что практически исключает ее использование).

Перед подробным описанием этих классов следует сказать, что класс CSingleLock в приложении может совсем не использоваться, в то время как

класс CMultiLock незаменим при необходимости ожидания установки сразу нескольких объектов синхронизации.

Класс CSingleLock

CSingleLock

Класс CSingleLock обеспечивает механизм синхронизации для одного объекта любого из классов синхронизации: CCriticalSection, CMutex, CEvent или CSemaphore. Для создания объекта класса используется конструктор:

CSingleLock::**CSingleLock**(CSyncObject* pObject, BOOL bInitialLock = FALSE) —

первый параметр задает через указатель объект одного из классов синхронизации, а второй позволяет немедленно заблокировать поток в ожидании того момента, когда соответствующий объект синхронизации будет установлен (в этом случае значение параметра bInitialLock должно быть равным TRUE, а ожидание не будет ограничено по времени).

Для задания временного ограничения ожидания следует использовать функцию Lock:

BOOL CSingleLock::**Lock**(DWORD dwTimeOut = INFINITE) —

параметр dwTimeOut задает время в миллисекундах, по истечении которого, если ожидаемое событие (установка объекта синхронизации) не произойдет, функция завершится, вернув значение FALSE. Значение этого параметра равное INFINITE приведет к отключению временного ограничения.

Завершение блокировки потока, а по сути — перевод синхронизирующего объекта в состояние "установлен" (кроме объекта класса CEvent), реализуется функцией Unlock, которая полностью аналогична по своему назначению, использованию и параметрам функции *Unlock* класса *CSemaphore*. Эта же функция (ее версия без параметров) вызывается в деструкторе.

Проверить текущее состояние объекта синхронизации можно при помощи функции IsLocked:

BOOL CSingleLock::**IsLocked**( )

К сожалению, возвращаемое ею значение актуально сразу же после выполнения функции Lock или Unlock, т. к. контроль состояния объекта синхронизации по сути *не производится*, и функция возвращает результат последнего вызова одной из этих функций. Собственно текущее состояние объекта синхронизации проконтролировать невозможно — оно может меняться асинхронно — в этом суть многозадачности. В принципе только для объектов критических секций и взаимоисключений, а также семафоров с максимальным значением счетчика 1 после выполнения функции Lock можно гарантировать достоверность полученного значения, но в этом случае состояние и так известно — объект находится в состоянии "сброшен".

Класс CMultiLock

CMultiLock

Класс CMultiLock обеспечивает механизм синхронизации сразу для нескольких объектов любых классов синхрони-

зации, кроме класса CCriticalSection. Количество таких объектов ограничено значением MAXIMUM_WAIT_OBJECTS, которое в настоящий момент определено как 64. Обычно в качестве объектов синхронизации используются объекты "событие".

Для создания объекта класса необходимо создать массив указателей на объекты классов синхронизации, а затем вызвать следующий конструктор:

CMultiLock::**CMultiLock**(CSyncObject* ppObjects [Sn], DWORD dwCount, BOOL bInitialLock = FALSE) —

параметр ppObjects определяет массив указателей на объекты синхронизирующих классов, размерность которого определяется параметром dwCount. Параметр bInitialLock используется аналогично конструктору класса CSyncLock. При его значении равном TRUE в конструкторе вызывается функция блокировки Lock со значениями параметров, задаваемыми по умолчанию.

DWORD CMultiLock::**Lock**(DWORD dwTimeOut = INFINITE, BOOL bWaitForAll = TRUE, DWORD dwWakeMask = 0) —

параметр dwTimeOut задает время в миллисекундах, по истечении которого, если ожидаемое событие (установка хотя бы одного или всех объектов синхронизации в зависимости от значения параметра bWaitForAll или возникновение заданного параметром dwWakeMask события) не произойдет, функция завершится, вернув значение WAIT_TIMEOUT. Значение этого параметра, равное INFINITE, приведет к отключению временного ограничения.

Параметр bWaitForAll определяет, следует ли ожидать установки всех объектов синхронизации для завершения функции (TRUE) или достаточно хотя бы одного (FALSE). При значении данного параметра FALSE и установки более чем одного объекта синхронизации одновременно, возможно достоверное определение изменения состояния только того объекта, указатель на который располагается ближе всех к началу массива указателей.

Параметр dwWakeMask определяет дополнительные условия завершения функции блокировки. Под дополнительными условиями понимается ряд событий-сообщений, которые могут оказаться в очереди сообщений потока. Этот параметр может принимать любую комбинацию следующих значений:

QS_ALLINPUT	Любое сообщение
QS_HOTKEY	Сообщение WM_HOTKEY
QS_INPUT	Сообщение о вводе
QS_KEY	Сообщение о нажатии клавиши, в том числе системной: WM_KEYUP, WM_KEYDOWN, WM_SYSKEYUP или WM_SYSKEYDOWN
QS_MOUSE	Сообщение об изменении положения мыши или о нажатии одной из ее кнопок: WM_MOUSEMOVE или WM_LBUTTONUP, WM_RBUTTONDOWN и т. д.
QS_MOUSEBUTTON	Сообщение о нажатии одной из кнопок мыши: WM_LBUTTONUP, WM_RBUTTONDOWN и т. д.
QS_MOUSEMOVE	Сообщение об изменении положения мыши WM_MOUSEMOVE
QS_PAINT	Сообщение WM_PAINT

QS_POSTMESSAGE	Любое, кроме перечисленных сообщений, поступивших в очередь сообщений в результате выполнения функции PostMessage
QS_SENDMESSAGE	Любое, кроме перечисленных сообщений, поступивших в очередь сообщений в результате выполнения функции SendMessage
QS_TIMER	Сообщение WM_TIMER

В зависимости от возвращаемого значения приложение может определить, в результате какого события произошло завершение функции:

–1	Произошла ошибка
от WAIT_OBJECT_0 до WAIT_OBJECT_0 + (число объектов –1)	Если значение параметра bWaitForAll равно TRUE, все объекты были переведены в состояние "установлен". Иначе установленный объект определяется по индексу массива, который вычисляется по формуле: (значение функции – WAIT_OBJECT_0)
WAIT_OBJECT_0 + (число объектов)	Произошло событие (или события), заданное параметром dwWakeMask, т. е. в очередь сообщений пришло одно из сообщений
от WAIT_ABANDONED_0 до WAIT_ABANDONED_0 + (число объектов – 1)	Если значение параметра bWaitForAll равно TRUE, все объекты были переведены в состояние "установлен" и хотя бы один объект является объектом взаимоисключения (mutex). Иначе установленный объект "взаимоисключение" определяется по индексу массива, который вычисляется по формуле: (значение функции – WAIT_ABANDONED_0)
WAIT_TIMEOUT	Истек заданный параметром dwTimeOut временной интервал

Остальные функции класса Unlock и IsLocked аналогичны функциям с теми же именами класса CSingleLock. (Функция IsLocked позволяет проверять состояние конкретного объекта, заданного индексом массива.)

Предупреждение

Версия функции Unlock с двумя параметрами есть как в описании класса в заголовочном файле, так и в документации. А вот соответствующего кода нет. НЕТ! Если хорошей практикой является отсутствие даже самых незначительных предупреждающих сообщений, то наличие сообщений об ошибках просто не может допускаться. Мы старались при написании книги все описанные или упомянутые функции проверить в работе, но то, что осталось за кадром, совершенно не внушает доверия. Конечно, отсутствие функции будет определено при сборке программы, но если в библиотеке нет ряда функций (хотя бы одной!), то есть ли гарантия, что содержимое реализованных функций не содержит аналогичных грубых и недопустимых ошибок?

Terminal — многопотоковое приложение, взаимодействующее с последовательным портом

Приложение Terminal иллюстрирует многое из того, чему была посвящена данная часть книги. Основная задача, решаемая приложением, — организация взаимодействия с последовательным портом. Для простоты считается, что к последовательному порту подключен модем, и именно с ним происходит обмен информацией. Для работы с другими периферийными устройствами потребуется не только изменить настройки, но и реализовать соответствующий устройству механизм обмена, а также тот протокол, который это устройство использует. Эта же программа просто принимает и получает байты данных, которые интерпретируются как символы. Единственным, пожалуй, расширением является добавление к символу возврата каретки (код 0Dh [13]) символа перевода строки (код 0Ah [10]).

В основе приложения лежат следующие механизмы:

- Перехват кодов клавиш и перенаправление их в кольцевой буфер вместо окна редактора
- Инициализация последовательного порта с одновременным запуском трех дополнительных потоков
- Обслуживание выходного буфера с осуществлением записи в последовательный порт
- Обслуживание входного буфера с осуществлением обновления окна редактора
- Чтение из последовательного порта во входной буфер
- Реализация кольцевого буфера с динамически изменяемой (увеличивающейся) длиной, в виде кольцевой очереди, и способного работать как на чтение, так и на запись асинхронно

Класс блока диалога и функции потоков

Класс блока диалога управляет всей программой. Он занимается тем, что активизирует инициализацию одного из четырех последовательных портов, а также выполняет действия, связанные с завершением работы приложения, — завершением дополнительных потоков и освобождением захваченного последовательного порта.

```
// Файл TerminalDlg.h
class CTerminalDlg: public CDialog
{
  DECLARE_DYNAMIC(CTerminalDlg)
```

```
  ...
// Dialog Data
  //{{AFX_DATA(CTerminalDlg)
  enum { IDD = IDD_TERMINAL_DIALOG };
  CTermWnd m_Terminal;
  //}}AFX_DATA
  //{{AFX_MSG(CTerminalDlg)
  virtual BOOL OnInitDialog();
  afx_msg void OnPaint();
  afx_msg HCURSOR OnQueryDragIcon();
  afx_msg void OnClose();
  afx_msg void OnCom1();
  afx_msg void OnCom2();
  afx_msg void OnCom3();
  afx_msg void OnCom4();
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()
private:
  void Initialize(LPCTSTR lpCommPort);
  void UnInitialize();
public:
  CModem m_Modem;
  CWinThread *m_pOutThread;
  CWinThread *m_pInThread;
  CWinThread *m_pDrawThread;
  CEvent m_Quit;
};
```

Все объекты, используемые в работе, так или иначе связаны с объектом блока диалога. Один из элементов этого класса — объект класса CModem, в котором реализованы функции работы с модемом и с которым связаны входной и выходной буферы обмена.

```
// Файл TerminalDlg.cpp
// Объявление функций потоков
DWORD OutData(LPVOID lParam);
DWORD InData(LPVOID lParam);
DWORD DrawData(LPVOID lParam);
// Необходимо для реализации получения информации
// об объекте времени выполнения для контроля
// результата преобразования параметра функции потока
```

```
IMPLEMENT_DYNAMIC(CTerminalDlg, CDialog)
// В результате создается объект "событие", управляющий завершением
// потоков; этот объект является событием ручного сброса
CTerminalDlg::CTerminalDlg(CWnd* pParent /*=NULL*/)
 : CDialog(CTerminalDlg::IDD, pParent),
    m_Quit(FALSE, TRUE)
{
  ...
  m_pOutThread = NULL;
  m_pInThread = NULL;
  m_pDrawThread = NULL;
}
BEGIN_MESSAGE_MAP(CTerminalDlg, CDialog)
  //{{AFX_MSG_MAP(CTerminalDlg)
  ON_WM_PAINT()
  ON_WM_QUERYDRAGICON()
  ON_WM_CLOSE()
  ON_BN_CLICKED(IDC_COM1, OnCom1)
  ON_BN_CLICKED(IDC_COM2, OnCom2)
  ON_BN_CLICKED(IDC_COM3, OnCom3)
  ON_BN_CLICKED(IDC_COM4, OnCom4)
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

Основными функциями класса являются функции запуска процесса обмена с устройством и завершения этого обмена.

```
BOOL CTerminalDlg::OnInitDialog()
{
  ...
  // По умолчанию пытаемся проинициализировать
  // первый последовательный порт
  Initialize("COM1");
  ...
}
void CTerminalDlg::OnClose()
{
  UnInitialize();
  CDialog::OnClose();
}
```

```
// Функции-обработчики команд инициализации
// соответствующих последовательных портов
void CTerminalDlg::OnCom1()
{
  Initialize(_T("COM1"));
}
void CTerminalDlg::OnCom4()
{
  Initialize(_T("COM4"));
}
```

Для того чтобы инициализировать порт, необходимо проверить, не работали ли мы с другим последовательным портом, т. к. в этом случае его необходимо освободить и завершить работу дополнительных потоков. Для простоты вызывается функция деинициализации независимо от того, работали мы с другим портом или нет.

```
// Функция подготовки к обмену с устройством
void CTerminalDlg::Initialize(LPCTSTR lpCommPort)
{
  // Для начала завершим работу с портом,
  // с которым работали (или пытались работать) раньше
  UnInitialize();
  CString csTitle = _T("Terminal");
  // Пытаемся проинициализировать порт;
  // невозможность инициализации обуславливается двумя факторами:
  //   — порт отсутствует;
  //   — порт занят другим процессом.
  if (m_Modem.Initialize(lpCommPort))  {
    m_Terminal.EnableWindow(TRUE);
    GotoDlgCtrl(&m_Terminal);
    // Запускаем все дополнительные потоки, отключаем возможность
    // самоудаления описывающих их объектов класса CWinThread
    // для того, чтобы была возможность корректного завершения всего
    // приложения, т.е. завершения основного потока только
    // после того, как завершены дополнительные
    m_pOutThread =
      AfxBeginThread((AFX_THREADPROC)OutData, this);
    m_pOutThread->m_bAutoDelete = FALSE;
    m_pInThread =
      AfxBeginThread((AFX_THREADPROC)InData, this);
```

```
    m_pInThread->m_bAutoDelete = FALSE;
    m_pDrawThread =
      AfxBeginThread((AFX_THREADPROC)DrawData, this);
    m_pDrawThread->m_bAutoDelete = FALSE;
    csTitle.Format("%s -- %s", _T("Terminal"), lpCommPort);
  }
  SetWindowText(csTitle);
}
```

Пскольку для обмена мы используем сразу три потока, которые в случае окончания работы с последовательным портом необходимо завершить, для этого достаточно одного объекта "событие ручного сброса". После перевода его в установленное состояние все потоки со временем завершатся. Наша задача — реализовать потоки таким образом, чтобы они большую часть времени были заблокированы в ожидании, в том числе и сигнала о необходимости завершения.

```
// Функция завершения обмена с устройством
void CTerminalDlg::UnInitialize()
{
  m_Terminal.EnableWindow(FALSE);
  // Сигнализируем о необходимости завершения
  m_Quit.SetEvent();
  // Ждем завершения потоков только в том случае, если они
  // существуют; затем удаляем соответствующие объекты
  if (m_pOutThread)  {
    WaitForSingleObject(m_pOutThread->m_hThread, INFINITE);
    delete m_pOutThread;
    m_pOutThread = NULL;
  }
  if (m_pInThread)  {
    m_Modem.UnInitialize();
    WaitForSingleObject(m_pInThread->m_hThread, INFINITE);
    delete m_pInThread;
    m_pInThread = NULL;
  }
  if (m_pDrawThread)  {
    WaitForSingleObject(m_pDrawThread->m_hThread, INFINITE);
    delete m_pDrawThread;
    m_pDrawThread = NULL;
  }
```

```
  // Сбрасываем событие, сигнализировавшее
  // о необходимости завершить дополнительные потоки
  m_Quit.ResetEvent();
}
```

Все три потока выполняются таким образом, что постоянно отслеживают сигнал о необходимости завершения (помимо тех событий, для которых они созданы), а именно — поступление символов либо в соответствующий буфер, либо из последовательного порта. Таким образом, есть стопроцентная гарантия того, что в функции завершения работы потоков функции ожидания всегда завершатся и практически сразу же после вызова.

```
// Функция чтения из выходного буфера с записью
// прочитанных символов в последовательный порт
DWORD OutData(LPVOID lParam)
{
  CTerminalDlg *pWnd = (CTerminalDlg *)lParam;
  ASSERT(pWnd->IsKindOf(RUNTIME_CLASS(CTerminalDlg)));
  // Читаем только тогда, когда буфер не пуст и до тех пор,
  // пока не придет сигнал о необходимости завершения
  CSyncObject *pObject[] =
    { &pWnd->m_Modem.m_OutBuffer.m_NotEmpty, &pWnd->m_Quit };
  DWORD dwBytes;
  CHAR cChar;
  for (;;)  {
    CMultiLock Lock(pObject, 2);
    switch (Lock.Lock(INFINITE, FALSE))
    {
    case WAIT_OBJECT_0:
      if (pWnd->m_Modem.m_OutBuffer.ReadChar(cChar))
      {
        WriteFile(pWnd->m_Modem.m_hComm, &cChar,
          sizeof(CHAR), &dwBytes, NULL);
      }
      break;
      default:
      return 0;
    }
  }
  return 0;
}
```

Описание работы с последовательным портом и асинхронным вводом/выводом выходит за рамки нашей книги. Однако анализ примера позволит вам в общих чертах познакомиться с этими механизмами.

```
// Функция чтения из последовательного порта
// с записью прочитанных символов во входной буфер
DWORD InData(LPVOID lParam)
{
  BYTE szBuffer[4096];
  DWORD dwEvtMask;
  DWORD dwError;
  DWORD dwBytes;
  COMSTAT csStat;
  CTerminalDlg *pWnd = (CTerminalDlg *)lParam;
  ASSERT(pWnd->IsKindOf(RUNTIME_CLASS(CTerminalDlg)));
  CEvent& event = pWnd->m_Quit;
  OVERLAPPED o;
  o.hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
  // Читаем только тогда, когда на входе что-то есть, и до тех пор,
  // пока не придет сигнал о необходимости завершения
  HANDLE Handles[] = { o.hEvent, event };
  for (;;)  {
    ResetEvent(o.hEvent);
    // Асинхронная работа с устройством — информируем систему
    // о том, что ждем сигнал от порта
    if (!WaitCommEvent(pWnd->m_Modem.m_hComm, &dwEvtMask, &o))    {
      // Ждем либо сигнала от порта, либо сигнала
      // о необходимости завершения работы потока
      if (WaitForMultipleObjects(2, Handles, FALSE, INFINITE) ==
          WAIT_OBJECT_0+1)
        break;
    }
    if (dwEvtMask & EV_BREAK)
      break;
    else if (dwEvtMask & EV_RXCHAR)    {
      ClearCommError(pWnd->m_Modem.m_hComm,  &dwError, &csStat);
      ResetEvent(o.hEvent);
      // Асинхронное чтение — информируем систему
      // о том, что ждем сигнала о завершении процесса чтения
      if (!ReadFile(pWnd->m_Modem.m_hComm, szBuffer,
```

```
        csStat.cbInQue, &dwBytes, &o))      {
        // Ждем либо сигнала о завершении чтения, либо сигнала
        // о необходимости завершения работы потока
        if (WaitForMultipleObjects(2, Handles, FALSE, INFINITE) ==
          WAIT_OBJECT_0+1)
          break;
      }
      // Записываем полученные символы во входной буфер
      for (UINT i = 0; i < csStat.cbInQue; i++)
        pWnd->m_Modem.m_InBuffer.WriteChar(szBuffer[i]);
    }
  }
  return 0;
}
```

Во всех потоках мы с легкостью пользуемся указателями на объекты классов CTerminalDlg и CTermWnd. При этом мы не создаем новые объекты на основе дескрипторов окон. Однако это не противоречит тем правилам, которые были изложены в разделе "Организация доступа к объектам классов MFC", т. к. в этих потоках мы не используем ни одной функции этих классов, за исключением функции ReplaceSel, которая только отправляет окну соответствующее сообщение, что вполне допустимо.

```
// Функция чтения из входного буфера с последующим
// изменением окна редактора
DWORD DrawData(LPVOID lParam)
{
  CTerminalDlg *pWnd = (CTerminalDlg *)lParam;
  ASSERT(pWnd->IsKindOf(RUNTIME_CLASS(CTerminalDlg)));
  // Читаем только тогда, когда буфер не пуст, и до тех пор,
  // пока не придет сигнал о необходимости завершения
  CSyncObject *pObject[] =
    { &pWnd->m_Modem.m_InBuffer.m_NotEmpty, &pWnd->m_Quit };
  CHAR cChar;
  CString csString;
  for (;;)  {
    CMultiLock Lock(pObject, 2);
    switch (Lock.Lock(INFINITE, FALSE))    {
    case WAIT_OBJECT_0:
      csString.Empty();
      for (;;)      {
```

```
        if (!pWnd->m_Modem.m_InBuffer.ReadChar(cChar))
          break;
        csString += cChar;
      }
      pWnd->m_Terminal.ReplaceSel(csString);
      break;
    default:
      return 0;
    }
  }
  return 0;
}
```

Окно редактора

Этот класс создан исключительно для того, чтобы перехватить коды клавиш и исключить вывод символов на экран — они будут выведены в окно как результат эхо-ввода, который может поддерживать модем. Естественно, что при отсутствии модема в окне ничего отображено не будет, хотя все потоки будут работать — мы инициализируем последовательный порт, к которому вовсе не обязательно подключен модем.

```
// Файл TermWnd.cpp
BEGIN_MESSAGE_MAP(CTermWnd, CEdit)
  //{{AFX_MSG_MAP(CTermWnd)
  ON_WM_CHAR()
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()
void CTermWnd::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)
{
  // Записываем поступающие коды клавиш в выходной буфер,
  // кроме кода клавиши <Esc>
  if (nChar != 0x1B)  {
    CBuffer& buffer =
      ((CTerminalDlg *)GetParent())->m_Modem.m_OutBuffer;
    buffer.WriteChar((CHAR)nChar);
    // Если надо — добавляем код символа перевода строки
    if (nChar == 0x0D)
      buffer.WriteChar(0x0A);
  }
}
```

Рис. 31.2. Окно приложения Terminal

Работа с последовательным портом

Для упрощения программы класс CModem не содержит функций чтения и записи, он занимается инициализацией и освобождением последовательного порта и в качестве элементов содержит два буфера — входной и выходной.

```
// Файл Modem.h
class CModem: public CObject
{
public:
  CModem();
  BOOL Initialize(LPCTSTR lpCommPort);
  void UnInitialize();
public:
  CBuffer m_OutBuffer;
  CBuffer m_InBuffer;
  HANDLE m_hComm;
public:
  virtual ~CModem();
};
// Файл Modem.cpp
CModem::CModem()
{
  m_hComm = INVALID_HANDLE_VALUE;
}
CModem::~CModem()
```

```
{
  UnInitialize();
}
```

Обратите внимание на функцию открытия последовательного порта — Win32 API не поддерживает больше специальных функций работы с ними. Настройка самого порта в общем случае может быть существенно больше: необходимо настраивать внутренние буферы, задавать тайм-аут (и не один) и т. д. Но для данного примера такой инициализации вполне достаточно.

```
BOOL CModem::Initialize(LPCTSTR lpCommPort)
{
  // Инициализируем порт для асинхронного обмена
  m_hComm = CreateFile(lpCommPort,
    GENERIC_READ | GENERIC_WRITE, 0, NULL,
    OPEN_EXISTING, FILE_FLAG_OVERLAPPED, NULL);
  if (m_hComm == INVALID_HANDLE_VALUE)
    return FALSE;
  DCB dcb;
  // Получаем текущие параметры порта и изменяем часть из них
  GetCommState(m_hComm, &dcb);
  BuildCommDCB("baud=19200 parity=N data=8 stop=1", &dcb);
  SetCommState(m_hComm, &dcb);
  // Задаем маску событий на сигнал break и получение очередного символа
  SetCommMask(m_hComm, EV_BREAK | EV_RXCHAR);
  return TRUE;
}
void CModem::UnInitialize()
{
  if (m_hComm != INVALID_HANDLE_VALUE)  {
    EscapeCommFunction(m_hComm, SETBREAK);
    CloseHandle(m_hComm);
    m_hComm = INVALID_HANDLE_VALUE;
  }
}
```

Кольцевой буфер общего доступа

Класс кольцевого буфера реализован по всем правилам, изложенным в разделе "Создание класса разделяемых объектов". Объект этого класса сам заботится об организации взаимоисключающего доступа при помощи объекта класса CCritiaclSection. Кроме того, в его составе есть объект класса CEvent для сигнализации о поступлении нового символа.

```
// Файл Buffer.h
class CBuffer: public CObject
{
public:
  CBuffer(int nLength = 100, int m_Extra = 10);
private:
  CHAR *m_pBuffer;
  int m_Length, m_Extra;
  int m_Head, m_Tail;
  CCriticalSection m_Lock;
public:
  CEvent m_NotEmpty;
public:
  void WriteChar(CHAR cChar);
  BOOL ReadChar(CHAR& cChar);
public:
  virtual ~CBuffer();
};
// Файл Buffer.cpp
CBuffer::CBuffer(int nLength, int nExtra)
{
  m_Length = nLength;
  m_Extra = nExtra;
  m_Head = m_Tail = 0;
  m_pBuffer = new CHAR[m_Length];
}
CBuffer::~CBuffer()
{
  delete m_pBuffer;
}
BOOL CBuffer::ReadChar(CHAR& cChar)
{
  CSingleLock Lock(&m_Lock, TRUE);
  if (m_Tail == m_Head)
    return FALSE;
  cChar = m_pBuffer[m_Tail++];
  m_Tail %= m_Length;
  return TRUE;
}
```

```
void CBuffer::WriteChar(CHAR cChar)
{
  CSingleLock Lock(&m_Lock, TRUE);
  m_pBuffer[m_Head++] = cChar;
  m_Head %= m_Length;
  // При необходимости размер буфера может быть увеличен
  if (m_Head == m_Tail)
  {
    CHAR *pBuffer = new CHAR[m_Length+m_Extra];
    memcpy(pBuffer, m_pBuffer, sizeof(CHAR) * m_Head);
    memcpy(pBuffer+m_Tail+m_Extra, m_pBuffer+m_Tail,
      sizeof(CHAR) * (m_Length-m_Tail));
    m_Length += m_Extra;
    m_Tail += m_Extra;
    delete m_pBuffer;
    m_pBuffer = pBuffer;
  }
  // Разрешаем доступ к буферу до того, как просигнализируем
  // о поступлении нового символа, чтобы не потерять
  // ни одного такта процессора в случае, если после этого
  // сигнала процессор будет переключен на поток,
  // осуществляющий чтение, а доступ еще не будет разрешен
  Lock.Unlock();
  m_NotEmpty.SetEvent();
}
```

Рис. 31.3. Окно приложения Terminal: к порту COM2 присоединен модем, в окне — результат работы модема

INTERNET И БАЗЫ ДАННЫХ

Глава 32

Работа с информационным сервером Internet

Существует много различных способов работы с информационным сервером Internet (Internet Information Server, IIS), большинство из которых связано с использованием традиционных статических HTML-страниц, требующих применения различных типов скриптов. Например, можно использовать скрипт CGI (Common Gateway Interface, Общий интерфейс шлюзов) для запроса к базе данных из программы на C/C++. В этом случае скрипт форматирует страницу HTML, содержащую результаты запроса, и передает данные клиенту. В этой главе мы не будем рассматривать традиционные способы, т. к. имеется достаточно книг, посвященных этой теме. Тем не менее необходимо понимать, что традиционные методы существуют, работают, и может оказаться, что вам нужны именно они.

С другой стороны, использование новых технологий, например, активных серверных страниц (Active Server Pages, ASP) может обеспечить большую гибкость и существенно увеличить возможности по предоставлению информации в Internet. Об этом мы и будем говорить в данной главе — о новой технологии, которую можно использовать для обеспечения большей содержательности создаваемого Web-сайта. Мы рассмотрим интерфейс программирования приложений для сервера Internet (Internet Server Application Programming Interface, ISAPI) и, что более важно, — одну из доступных технологий, которые работают с ISAPI, такую как ISA (ISAPI Server Application, Приложение сервера ISAPI). Естественно, все эти вопросы будут рассмотрены с точки зрения библиотеки классов MFC.

Так что же такое ISAPI? Несколько упрощенно будем считать, что ISAPI — это набор расширений MFC, которые позволяют работать непосредственно с IIS. В библиотеке реализованы пять классов, часть из которых мы рассмотрим достаточно подробно: CHttpServer, CHttpServerContext, CHttpFilter, CHttpFilterContext и CHtmlStream. Эти классы значительно упрощают создание приложений ISA.

Примечание

Кроме термина ISA, часто используются другие, например, *динамические библиотеки расширений сервера ISAPI* (ISAPI server extension DLLs), встречающиеся в документации Microsoft. Мы будем использовать термин ISA.

С помощью классов ISAPI можно создавать расширения и фильтры ISA для IIS. Однако ваши возможности не ограничены классами ISAPI; имеются также классы WinInet для создания клиентских приложений Internet и все стандартные классы MFC, предоставляющие интерфейсы. Но чем же отличаются фильтры и расширения?

- *Фильтры* позволяют сохранять что-либо внутри или вне сервера, контролируя события на нем. Например, можно создать фильтр ISA, который не допустит на Web-сервер посетителей, пока они не введут правильный пароль. Другой тип фильтра может предотвращать загрузку файлов на FTP-сервер вашего сайта, которые имеют размер больший, чем установлено вами.
- *Расширения* — это скорее приложения или фоновые процессы. Например, можно создать расширение, которое позволяет пользователю обращаться к базе данных без использования скриптов. То же самое расширение могло бы создавать Web-страницы, динамически меняющиеся в зависимости от прав доступа пользователя и содержимого вашей базы данных.

В следующих разделах рассмотрены пять важных областей программирования ISAPI. Краткий теоретический обзор вы можете пропустить, если уже имеете представление о том, как работает ISAPI, и хотите непосредственно перейти к созданию ваших собственных программ. Во втором разделе главы рассказывается, как создать расширения ISAPI, в третьем — как реализовать их на сервере.

Обзор ISAPI (API сервера Internet)

Прежде чем разбираться с ISAPI, необходимо понять, какое место он занимает в схеме Microsoft. Имеются пять уровней поддержки Internet, предоставляемых Visual C++, три из которых могут находиться на сервере. Другие два уровня поддержки специфичны для клиента — вы никогда не встретите их на сервере. В следующем списке определен каждый уровень поддержки:

- *ISAPI (сервер)* — нужен для обеспечения расширения или фильтра непосредственно сервера. Другими словами, клиент не будет взаимодействовать непосредственно с этим уровнем поддержки, а будет видеть только результаты.
- *WinInet (сервер или клиент)* — мы не рассматриваем этот уровень поддержки подробно, а делаем краткий обзор. Этот набор классов позволяет ис-

пользовать специальные методы передачи данных между клиентом и сервером. Имеются три уровня поддержки для разных протоколов: HTTP, FTP и Gopher. По существу эти классы используются для создания сеанса (CInternetSession), который сначала определяет соединение с сервером, а затем тип соединения (CFtpConnection, CHttpConnection или CGopherConnection). После установления соединения пользователь может выполнять задачи, такие как поиск файла (CFtpFileFind или CGopherFileFind). Обычно вам не нужно непосредственно взаимодействовать с этими классами, т. к. Visual C++ уже позаботился об этом.

- *Asynchronous URL-Monikers — асинхронные URL-моникеры* (*сервер или клиент*). Здесь важен следующий момент: нужно помнить, что асинхронный URL-моникер позволяет выполнять задачи в Internet без ожидания. Вы просто сообщаете целевому приложению о том, что вы хотите, и делаете что-нибудь другое. Вся идея состоит в том, что Internet в большинстве ситуаций не предоставляет мгновенного ответа, но даже если он это делает, то без использования асинхронных URL-моникеров долгий процесс передачи файлов мог бы лишить пользователя машины, бесполезной в течение довольно продолжительного времени.
- *ActiveX Documents — документы ActiveX (клиент).* Показывают документ в браузере клиента и позволяют пользователю редактировать его.
- *ActiveX Controls — элементы управления ActiveX (клиент).* Создание основных элементов Web-страницы на основе большого числа скриптов все равно приводит к получению статического изображения. Использование элементов управления ActiveX на Web-странице означает, что старая технология и статический вывод изображений больше не являются помехой — Web-страница теперь может изменяться в соответствии с конкретной ситуацией.

ISA является модулем динамической библиотеки DLL, который загружается на сервере так же, как и любая другая DLL. Эта DLL будет разделять те же самые адреса, которые использует сервер HTTP, и ее можно выгрузить из памяти, если она потребуется еще для чего-нибудь. В следующем списке приводятся преимущества использования ISA по сравнению с другими методами, подобными CGI:

- *Более низкие затраты памяти.* Поскольку ISA загружается на сервере как DLL и использует ту же самую память, что и сервер HTTP, не приходится тратить впустую память, обычно связанную с накладными расходами CGI. Требуется только загрузить логическое требование к ISA выполнить задачу.
- *Скорость.* Загрузка DLL или приложения на языке C в первый раз длится примерно одинаково. Однако как только DLL загружена первый раз, она останется в памяти до момента ее выгрузки, а это означает, что не будет накладных расходов по новой загрузке, если сами не хотите этого. Скрипт CGI загружает C-приложение каждый раз, когда вы вызываете

его. Однако это не лучшая новость. Поскольку DLL ISA разделяет память, как и сервер HTTP, не нужно будет терять время на запросы внутри процесса или любые накладные расходы, связанные с использованием приложения на C в отдельном адресном пространстве.

- *Возможность фильтрации.* Достаточно трудно реализовать эквивалент ISA путем использования CGI и приложения на языке C. Причина проста: приложение на C вызывается, выполняется и затем выгружается — нет способа контроля событий сервера за длительный период.
- *Несколько задач в одной DLL.* Каждая задача, которую для вас должен выполнить CGI, требует дополнительных затрат по вызову каждой процедуры. С другой стороны, ISA может содержать множество команд, каждая из которых является компонентом класса CHttpServer.

Тем не менее, при всех описанных различиях ISA столь же удобен в использовании, как и его эквивалент CGI. Приведем здесь две строки кода из приложения:

```
<!-- Это вызов процедуры CGI с одним параметром. -->
http://host/scripts/sample.exe?Param

<!-- Это вызов процедуры ISA с одним параметром. -->
http://host/scripts/sample.dll?Param
```

Как видите, работать с ISAPI совсем нетрудно. Мы вызываем наш модуль ISA, используя примерно тот же самый код, что и для функции CGI. Единственное различие с точки зрения программирования заключается в том, что ISA использует расширение DLL, в то время как CGI использует расширение EXE (или CGI, что в общем-то эквивалентно для программ на языке C). Более того, если уже в процессе работы переключить ваш сервер на ISA, выполнить несколько операций типа "поиск-замена" на Web-странице, то никто не заметит различий с точки зрения интерфейса. При этом каждый отметил бы более высокую эффективность Web-страницы, что связано с преимуществами использования ISAPI.

Последнее преимущество использования ISA, которое мы упомянули, заключалось в том, что можно выполнять более одной задачи. Ниже приведен обычный вызов ISA, определяющий, что требуется использовать функцию DisplayStr:

```
<!-Вызов функции DisplayStr в процедуре ISA с одним параметром.->
http://host/scripts/sample.dll?DisplayStr?Param
```

Как вы видите, вместо задачи по умолчанию в этом случае вызываем кое-что другое. Первый параметр теперь сообщает, какая функция вызывается в SAMPLE.DLL, а второй содержит список параметров. Этот метод вызова функций известен *как карта разбора* (parse map), создание которой мы рассмотрим далее в этой главе.

Примечание

ISA запускается на сервере, что позволяет обновлять только сервер, а не приложения многих клиентов.

В ISA также имеются некоторые качества, которые вы найдете и в CGI. С одной стороны, ваше приложение выполняется на сервере, а не у клиента, за счет чего проще обновлять ваше приложение, — вы должны заменить на сервере только один файл DLL. (Вы должны остановить сервисы для замены DLL, но это небольшая плата за удобство одной модернизации машины.) Очевидно, что это намного легче, чем пытаться заменить приложения на каждой машине клиента, имеющей доступ к вашей Web-странице.

Классы ISAPI

Для работы с ISA в библиотеке MFC реализованы пять классов:

- *CHttpServer*. Это главный класс, который нужен для создания фильтра или расширения. Класс CHttpServer позволяет определить объект, который работает как DLL-расширение сервера или ISA. Основное назначение объекта CHttpServer состоит в обработке сообщений клиента и сервера. По этой причине в любой динамической библиотеке ISA имеется только один объект CHttpServer.
- *CHttpServerContext*. Класс CHttpServer сам создает этот объект, который является единственным экземпляром отдельного запроса от клиента. Другими словами, для каждого активного запроса клиента "виден" только один объект класса CHttpServerContext, который автоматически уничтожается, когда запрос клиента выполнен.
- *CHttpFilter*. Фильтры ISA требуют использования специального объекта CHttpFilter для контроля событий сервера. Этот объект будет наблюдать за всеми сообщениями сервера, когда клиент с ним взаимодействует. Например, клиент генерирует сообщение SF_NOTIFY_LOG каждый раз, когда пользователь пробует получить доступ к Web-странице.
- *CHttpFilterContext*. Объект CHttpFilter создает один объект CHttpFilterContext для каждого активного события сервера. Класс CHttpFilterContext представляет единственное событие, сформированное отдельным клиентом во время определенного сеанса.
- *CHtmlStream*. Этот класс используется для управления обменом данными между клиентом и сервером. Объект этого класса создается каждый раз, когда требуется передать информацию между клиентом и сервером. В большинстве случаев в DLL имеется только один такой объект. Однако объект CHttpServer может создавать столько объектов CHtmlStream, сколько требуется для успешной передачи данных. Типичный объект

CHtmIStream содержит данные и все тэги, требуемые для создания содержимого страницы HTML. Например, если вы выполнили поиск в базе данных, результирующий объект CHtmIStream будет содержать не только данные, но также и тэги HTML, требуемые для действительного создания Web-страницы, используемой для отображения данных.

Теперь, после краткого знакомства с каждым из имеющихся пяти классов ISAPI рассмотрим более подробно два из них, а именно те, с которыми будем работать в приведенном ниже примере, а также принципы создания, или обновления, карты разбора.

Класс CHttpServer

Этот класс позволяет обрабатывать различные типы клиентских запросов. Все компоненты и методы этого класса можно условно разделить на следующие категории:

- *Создание.* В эту категорию входит конструктор
- *Атрибуты данных.* Шесть функций, используемых для установки начальных и заключительных атрибутов HTML-страницы
- *Переопределяемые методы.* Семь методов, позволяющих программисту более конкретно настроить функционирование объекта CHttpServer

Теперь рассмотрим составляющие этих категорий более подробно.

Создание

Конструктор:

CHttpServer::**CHttpServer**(TCHAR cDelimiter) —

создает объект класса. В каждом модуле может существовать только *один* объект этого класса. Параметр cDelimiter позволяет задать символ, который будет использоваться для разбиения строки запроса к серверу на части. По умолчанию используется символ '&'.

После создания объекта класса его надо инициализировать. Сделать это можно с помощью переопределяемой функции InitInstance, к которой мы еще вернемся.

Атрибуты данных

Функции, входящие в эту категорию, служат для установки начальных и заключительных тэгов страницы. Поэтому приведем описание только одной:

virtual BOOL CHttpServer::**InitInstance**(CHttpServerContext* pCtxt) —

вызывается для каждого запроса к ISA.

Переопределяемые методы

В эту категорию входят семь функций, которые в большинстве случаев достаточно хорошо справляются со своей задачей, и поэтому мы их только перечислим:

CallFunction	Ищет и выполняет соответствующую функцию, ассоциированную с командой в URL
OnParseError	Создает описание ошибки, возвращаемой клиенту
OnWriteBody	Задает данные, возвращаемые клиенту
HttpExtensionProc	Использует функции обратного вызова для чтения данных, пришедших от клиента, и в зависимости от них выполняет соответствующие действия
GetExtensionVersion	Возвращает номер версии расширения
ConstructStream	Создает объект класса CHtmlStream
TermineteExtension	Предоставляет безопасный способ очистки потоков и завершения работы

Класс CHttpServerContext

Чрезвычайно полезный класс, который избавляет от огромного количества "рутинной" работы. Это, собственно говоря, тот класс, который "выполняет" запрос пользователя. Рассматривать компоненты этого класса подробно мы не будем, поскольку практически всю работу по его созданию, поддержанию и удалению берет на себя библиотека, а точнее класс CHttpServer. По этой причине ограничимся только кратким перечислением компонентов.

m_bSendHeaders	Булевский флаг, показывающий, будет ли MFC автоматически посылать заголовок в ответе HTTP. Значение по умолчанию — TRUE, соответствует посылке заголовка
m_pECB	Указатель на структуру EXTENSION_CONTROL_BLOCK, которая содержит информацию, описывающую соединение между клиентом и сервером
m_pStream	Указатель на объект класса CHtmlStream, который сервер может использовать для взаимодействия с клиентом. Запись данных в этот поток приводит к пересылке их клиенту
CHttpServerContext	Конструирует объект класса
GetServerVariable	Копирует информацию, связанную с самим сервером или HTTP-соединением в соответствующий буфер
WriteClient	Немедленно посылает информацию HTTP-клиенту

ReadClient	Читает информацию тела HTTP-запроса клиента в соответствующий буфер
SeverSupportFunction	Предоставляет ISA с тем же основным набором функций, что и те, которое определены в реализации HTTP-сервера
TransmitFile	Посылает данные из файла на сервере тому, кто их запросил
GetChunkSize	Возвращает размер текущей порции данных
SetChunkSize	Устанавливает размер порции данных для текущего контекста

И в классе остался нерассмотренным один оператор (имеющий несколько реализаций):

void operator<<(LPCTSTR psz);
void operator<<(long int dw);
void operator<<(short int w);
void operator<<(const CHtmlStream& stream);
void operator<<(double d);
void operator<<(float f);
CHttpServerContext& operator<<(const CLongBinary& blob);
CHttpServerContext& operator<<(const CByteArray& array) —

записывает данные в поток HTML, которым владеет объект класса CHttpServerContext.

Именно этим оператором чаще всего и придется пользоваться при написании приложений ISA.

Карта разбора

Как вы наверняка уже догадались, карта разбора — не более чем набор макросов, которые, как и ранее рассмотренные макросы, позволяют сопоставить объект класса производного от CHttpServer с запросом клиента. Как обычно, все начинается с операторных скобок:

BEGIN_PARSE_MAP(theClass, baseClass) —

макрос начинающий

и

END_PARSE_MAP(theClass) —

макрос заканчивающий определение карты разбора.

Параметры theClass и theBaseClass определяют, соответственно, класс, для которого создается карта и его базовый класс.

ON_PARSE_COMMAND(FnName, mapClass, Args) —

определяет команду (FnName), которая будет обрабатывать соответствующий запрос клиента; mapClass — имя класса, для которого создается карта разбора; Args — аргументы, которые передаются в функцию FnName; они могут принимать следующие значения:

ITS_EMPTY	Аргументы отсутствуют
ITS_PSTR	Указатель на строку текста
ITS_RAW	Расширению ISA посылаются необработанные данные; этим значением следует пользоваться с осторожностью
ITS_I2	Короткое целое
ITS_I4	Длинное целое
ITS_R4	Вещественное
ITS_R8	Вещественное двойной точности

Примечание

При создании функции FnName следует иметь в виду, что они в качестве параметра должны принимать указатель на объект класса CHttpServerContext.

ON_PARSE_COMMAND_PARAMS(Params) —

определяет параметры по умолчанию, ассоциированные с функцией, которая обрабатывает команду клиента на сервере. Параметр Params задает имя параметра. Этот макрос должен располагаться непосредственно за макросом ON_PARSE_COMMAND_PARAMS, с которым он ассоциирован.

DEFAULT_PARSE_COMMAND(FnName, mapClass) —

позволяет определить функцию, которая будет вызываться, если запрос клиента к серверу не содержит команды. Параметры FnName и mapClass определяют имена, соответственно, функции и класса, с которым ассоциирована команда.

Создание расширения ISAPI

Теперь пришло время посмотреть на конкретный ISA. Проще всего создать каркас расширения ISAPI с помощью мастера ISAPI Extension Wizard. Главная цель этого простейшего примера состоит только в том, чтобы изложить основные идеи работы с ISAPI. Программа примера выводит страницу, на которой предлагается выбрать, какую информацию пользователь хочет получить — о сотрудниках или об имеющихся продуктах. После чего осуществляется переход к новой странице в зависимости от выбора клиента. Еще раз повторим — каркас этого приложения создает для нас ISAPI Extension Wizard и он полностью функционален.

Теперь, чтобы расширение выполняло необходимые нам функции, мы должны решить три задачи, доработав каркас программы, который для нас создал ISAPI Extension Wizard. Первая задача состоит в том, чтобы модифицировать карту разбора (parse map) для новой функции, которую мы хотим добавить. Вот как выглядит эта карта после внесенных изменений.

```
// Карта разбора команды
BEGIN_PARSE_MAP(CWwwdb, CHttpServer)
```

```
  // Команда, выполняемая по умолчанию
  DEFAULT_PARSE_COMMAND(Index, CWwwdb)
  ON_PARSE_COMMAND(Index, CWwwdb, ITS_EMPTY)

  // Добавленная нами команда с параметром
  ON_PARSE_COMMAND(Search, CWwwdb, ITS_PSTR)
  ON_PARSE_COMMAND_PARAMS("Method")
END_PARSE_MAP(CWwwdb)
```

Небольшие комментарии. Макрос ON_PARSE_COMMAND() позволяет определять новую функцию. Обратите внимание, что мы задаем имя функции, класс, которому функция принадлежит, и тип параметров, которые функция будет использовать. Макрос ON_PARSE_COMMAND() требует входных параметров, даже если они не нужны для работы функции. Обратите внимание, что функция Index использует значение ITS_EMPTY, так как ей не требуются параметры, а наша новая функция Search имеет параметр ITS_PSTR, потому что для нее необходим указатель на строку, которую нам дает макрос ON_PARSE_COMMAND_ PARAMS() на следующей строке. Вы должны сообщить Visual C++, как обращаться с параметрами вашей функции. Так, здесь мы предполагаем, что в качестве параметра будет передаваться строка "Method". И, наконец, мы сообщаем Visual C++, какая функция используется по умолчанию, применив для этого макрос DEFAULT_PARSE_COMMAND(). В данном случае — это функция Index, которая выводит пользователю основную страницу.

Второе, что вы должны сделать в коде, — это добавить компонент функции в файл WWWDB.H. Для этого нужно всего две строки, показанные в листинге ниже полужирным шрифтом.

```
class CWwwdb : public CHttpServer
{
...
public:
// Команды разбора
  void Index(CHttpServerContext* pCtxt);
  void Search(CHttpServerContext* pCtxt, LPCTSTR pszMethod);
...
};
```

Как уже говорилось в предыдущем разделе, объект класса CHttpServerContext автоматически создается единственным объектом класса CHttpServer для каждого запроса клиента. Здесь и передается указатель на объект CHttpServerContext, связанный с вызовом пользователем вашей функции. Это также является способом раздельного хранения множества

вызовов функции, т. е. каждый запрос имеет свой объект, ассоциированный с ним.

И последнее, что мы должны сделать для создания этих функций, — добавить их код функции в файл WWDB.CPP. Ниже представлены очень короткие функции, которые пересылают пользователю информационные страницы.

```
void CWwwdb::Index(CHttpServerContext* pCtxt)
{
  // Начинаем страницу
  StartContent(pCtxt);
  WritePageTitle(pCtxt, IDS_INDEX_TITLE);
  CString strOutput;
  // Выводим страницу, описанную в ресурсе IDS_HTML_INDEX
  LoadLongResource(strOutput, IDS_HTML_INDEX);
  *pCtxt << strOutput;
  // Заканчиваем страницу
  EndContent(pCtxt);
}

void CWwwdb::Search(CHttpServerContext* pCtxt, LPCTSTR pszMethod)
{
  // Начинаем страницу
  StartContent(pCtxt);
  WritePageTitle(pCtxt, IDS_SEARCH_TITLE);
  // В зависимости от значения параметра, переданного клиентом,
  // вызываем одну из функций
  if (_tcscmp(pszMethod, _T("ByEmployees")) == 0)  {
    *pCtxt << _T("First page sample wwwDB\r\n");
  }
  else if (_tcscmp(pszMethod, _T("ByProducts")) == 0)  {
    *pCtxt << _T("Second page sample wwwDB\r\n");
  }
  // Заканчиваем страницу
  EndContent(pCtxt);
}
```

Как видите, сами функции довольно просты. Первое, что мы делаем, это предписываем Visual C++ начать Web-страницу, аналогично добавлению в документ тэгов <HTML> и <HEAD>. Затем выводим заголовок — так же, как вводили бы в документ тэг <TITLE>. Теперь, когда мы получили заголовок, можно создавать тело содержимого. Для пересылки информации мы

используем оператор потока, куда направляем прочитанную из ресурсов строку. Более того, все, что вы можете делать со стандартным документом HTML, вы можете делать и с вашим расширением ISAPI. Мы рассмотрим только несколько примеров того, как все эти тэги работают вместе, чтобы сформировать Web-страницу. Заключительная функция, которую мы вызываем — EndContent(), сообщает Visual C++, что мы отправили информацию. Это подобно добавлению </HTML> в конце документа.

Примечание

Текст, который представлен в этом разделе, отличается от того, который вы найдете на сопроводительной дискете. Связано это с тем, что мы хотим показать последовательно все шаги создания приложения, постепенно заполняя его деталями. Расширение этого примера вы найдете в главе 34.

Теперь можно скомпилировать созданное расширение ISAPI, после чего нужно поместить его на ваш Web-сервер в каталог Scripts, и оно готово к тестированию Запустите ваш браузер и наберите адрес аналогичный следующему:

```
http://localhost/scripts/wwwdb.dll?
```

Если вы нигде не допустили опечаток, то получили работающее расширение ISA.

Но, как все прекрасно понимают, создавать что-либо для отображения двух строк текста никому не нужно. Поэтому мы расширим нашу заготовку, добавив в нее обращение к базе данных и вывод полученных результатов пользователю.

Но прежде чем переходить к этому вопросу, давайте рассмотрим классы библиотеки MFC, которые мы можем использовать для решения поставленной задачи.

Глава 33

Классы библиотеки MFC для работы с базами данных

Существует множество способов работы с базами данных, но в этой главе мы максимально подробно рассмотрим только один из них, а именно — использование механизма ODBC (Open Database Connectivity — открытое взаимодействие баз данных). Как и у любого метода, у этого существуют как свои преимущества, так и недостатки.

Классы для работы с ODBC

Эти классы, работая с другими классами приложения, обеспечивают простой доступ к различным базам данных, использующим драйверы ODBC. Приложениям, которые используют базы данных поддерживающих ODBC, следует иметь в своем составе, по крайней мере, два класса — CDatabase и CRecordset. С них мы и начнем.

Класс CDatabase

Объекты этого класса используются для предоставления соединения с базами данных, посредством которых можно манипулировать источником данных.

Компоненты данных

Класс CDatabase содержит только один компонент данных:

HDBC CDatabase::**m_hdbc** —

служит для хранения дескриптора соединения (connection handle) с источником данных ODBC. Обычно нет необходимости непосредственно обращаться к этому компоненту данных, поскольку библиотека автоматически предоставит вам дескриптор, когда вы вы-

зовете функцию Open или OpenEx, и, также автоматически, уберет его, при удалении объекта CDatabase. Следует иметь в виду, что вызов функции Close не приводит к закрытию дескриптора. Вместе с тем, вполне возможны ситуации, когда вам понадобится непосредственное обращение к функциям ODBC API. В этом случае вам понадобится этот дескриптор для передачи его в качестве параметра некоторой функции, как, например, это делается в приведенном ниже фрагменте:

```
nRetcode = ::SQLGetInfo(m_db.m_hdbc, SQL_ODBC_SQL_CONFORMANCE,
                        &nValue, sizeof( nValue ), &cbValue);
```

Создание соединения

CDatabase::**CDatabase** —

служит для создания объекта CDatabase:

После того как объект создан, необходимо создать соединение с определенным источником данных, для чего следует вызвать одну из приведенных ниже функций.

virtual BOOL CDatabase::**Open**(LPCTSTR lpszDSN,
BOOL bExclusive = FALSE, BOOL bReadOnly = FALSE,
LPCTSTR lpszConnect = "ODBC;", BOOL bUseCursorLib = TRUE) —

lpszDSN — определяет имя источника данных, которое должно быть зарегистрировано с помощью программы ODBC Administrator. Это значение должно быть равно NULL, если DSN определено в строке lpszConnect (в форме "DSN=<источник данных>") или может быть равно NULL, если необходимо предоставить пользователю блок диалога для выбора источника данных; bExclusive — все еще не поддерживается — источник данных всегда открывается для совместного использования, и должно быть равно FALSE, в противном случае будет выдано сообщение об ошибке; bReadOnly — позволяет установить соединение с источником данных в режиме "только для чтения" (TRUE), что приводит к запрещению его обновления; после установления такого соединения все зависимые результирующие множества наследуют этот атрибут; lpszConnect — определяет строку, описывающую соединение, которая содержит информацию об источнике данных, идентификаторе пользователя, имеющего к нему доступ, пароль, если он требуется источнику данных, и другую информацию; bUseCursorLib — указывает на необходимость (TRUE), или необязательность загрузки динамической библиотеки ODBC Cursor Library, позволяющей работать с курсорами.

Упрощенная версия рассмотренной функции имеет вид:

virtual BOOL CDatabase::**OpenEx**(LPCTSTR lpszConnectString,
DWORD dwOptions = 0) —

параметры: lpszConnectString — определяет строку соединения с источником данных ODBC, которая включает его имя, а также дополнительную необязательную информацию, такую как идентификатор и пароль пользователя, например, "DSN=Publisher;UID=sa; PWD=irishka". Если в качестве параметра передается NULL, то выводится блок диалога **Data Source**, в котором пользователь может выбрать источник данных; dwOptions — битовая маска, которая определяет комбинацию следующих значений:

CDatabase::**openExclusive**	Не поддерживается — источник данных всегда открывается для совместного использования. При задании этой опции будет выдано сообщение об ошибке
CDatabase::**openReadOnly**	Источник данных открывается в режиме "только для чтения"

CDatabase::**openUseCursorLib**	Указывает на необходимость загрузки динамической библиотеки ODBC Cursor Library, позволяющей работать с курсорами
CDatabase::**noOdbcDialog**	Не выводить блок диалога соединения
CDatabase::**forceOdbcDialog**	Всегда выводить блок диалога соединения

Значение по умолчанию (0) означает, что база данных открывается для совместного использования, с доступом для записи, динамическая библиотека поддержки курсора не загружается и блок диалога для выбора источника данных отображается только в том случае, если не указана дополнительная информация о соединении.

virtual void CDatabase::**Close**( ) —

не имеет параметров и не возвращает никакого значения:

Но, как вы понимаете, никому не нужно просто открыть и закрыть соединение с источником данных. Поэтому следует рассмотреть методы остальных перечисленных категорий.

Атрибуты данных

Функции, входящие в эту категорию, используются для предоставления информации о соединении, драйвере и источнике данных, а также для установки некоторых опций источника данных. Наиболее часто они используются в интерфейсных приложениях. Рассмотрим их подробно.

const CString& CDatabase::**GetConnect**( ) const —

позволяет получить строку, описывающую соединение, которая использовалась во время вызова функций открытия соединения (Open или OpenEx). Если до момента вызова этой функции соединение не было установлено, то возвращается ссылка на пустую строку.

BOOL CDatabase::**IsOpen**( ) const —

позволяет определить, имеется ли (возвращается ненулевое значение) или нет (0) текущее соединение объекта CDatabase с источником данных.

CString CDatabase::**GetDatabaseName**( ) const —

возвращает объект CString, содержащий либо имя текущей базы данных, либо пустую строку. Следует иметь в виду, что это имя отличается от имени источника данных DSN, задаваемого в функциях Open и OpenEx. Что именно вернет эта функция, — зависит от используемого ODBC.

BOOL CDatabase::**CanUpdate**( ) const —

может ли пользователь обновлять базу данных, определяется двумя факторами — возможностями драйвера ODBC (не все драйверы предоставляют возможность обновления) и режимом, в котором была открыта база данных. Если такая возможность есть, то функция возвращает ненулевое значение, в противном случае — 0. Фактор, влияющий на возможность обновления, можно определить, вызвав функцию ::SQLGetInfo с параметром SQL_DATASOURCE_READ_ONLY.

BOOL CDatabase::**CanTransact**( ) const —

позволяет определить, поддерживает ли база данных (возвращается ненулевое значение), или нет (0) транзакции, что определяется драйвером ODBC.

Примечание

Не следует устанавливать нулевое значение этого параметра, поскольку в этом случае соединение может зависнуть.

Операции

Функции этой категории используются для работы с базой данных. К ним относятся функции обработки транзакций (используемые для обновления базы данных) и непосредственного выполнения команд SQL. Прежде чем переходить к описанию функций, необходимо несколько слов сказать о том, что такое собственно транзакции.

Транзакция (transaction) — это логический блок операций, которые необходимо выполнить на сервере как единое целое. Например, если нужно перевести деньги с одного счета на другой, то необходимо обеспечить единство действий — снять деньги с одного счета и *обязательно* положить его на другой. Если какое-либо из этих действий не будет выполнено, то необходимо обеспечить возврат состояния счетов в то, которое они имели до начала операции перевода. Этот пример иллюстрирует основной принцип транзакции — "все или ничего". Можно сказать, что транзакции предусматривают способ объединения набора операций в элементарный исполняемый блок.

С точки зрения пользователя, транзакция — просто событие, которое либо происходит, либо нет. Для разработчика же транзакция — это стиль программирования, который позволяет создавать модули для распределенных вычислений.

Теперь, после краткого введения, вернемся к рассматриваемому предмету — компонентам класса CDatabase.

BOOL CDatabase::**BeginTrans**( ) —

начинает транзакцию с уже присоединенным источником данных. Транзакция состоит из одного или нескольких вызовов методов AddNew, Edit, Delete и Update класса CRecordset. Эту функцию не рекомендуется вызывать после того, как открыт результирующий набор, участвующий в транзакции. Это замечание связано с тем, что в зависимости от используемого драйвера ODBC могут возникнуть проблемы при вызове функции Rollback, характеризующей откат транзакции. Каждый вызов функции BeginTrans должен быть "закрыт" либо CommitTrans, либо Rollback.

Примечание

Перед тем как работать с транзакциями, необходимо убедиться, что источник данных их поддерживает. Сделать это можно с помощью функции CanTransact.

BOOL CDatabase::**CommitTrans**( ) —

завершает транзакцию, начатую функцией BeginTrans. Другими словами, "говорит" источнику данных, что можно принять заданные изменения. Она возвращает ненулевое значение, если изменения были успешно зафиксированы, и 0 в противном случае. При

этом состояние источника данных не определено, и необходима дополнительная проверка данных. Если же вас что-то не устраивает в ходе выполнения транзакции, то ее можно отменить, вызвав функцию Rollback, что приведет к возвращению источника данных в исходное состояние.

BOOL CDatabase::**Rollback**( ) —

отменяет изменения, выполненные во время транзакции. Все вызовы функций AddNew, Edit, Delete и Update класса CRecordset, выполненные после последнего вызова BeginTrans, отменяются, и источник данных возвращается в состояние, в котором он находился до начала транзакции. Если функция возвращает нулевое значение, то необходимо проверить источник данных, чтобы определить его состояние.

void CDatabase::**Cancel**( ) —

прекращает выполнение либо асинхронной операции, либо процесса вторичного потока.

Примечание

Классы ODBC библиотеки MFC не поддерживают асинхронную обработку. Для выполнения такого рода операций необходимо непосредственно вызвать функцию ODBC API ::SQLSetConnectOption.

void CDatabase::**ExecuteSQL**(LPCSTR lpszSQL) —

позволяет непосредственно выполнить команду SQL, задаваемую в завершающуюся нулем строке, на которую указывает параметр lpszSQL. В качестве параметра можно использовать объект класса CString. Следует иметь в виду, что выполнение этой функции не возвращает записей из базы данных и, следовательно, не рекомендуется для выполнения операций выборки:

Переопределяемые методы

В эту категорию входит единственная функция:

virtual void CDatabase::**OnSetOptions**(HSTMT hstmt) —

выполняется, когда вызывается функция ExecuteSQL, и служит для установки параметров работы приложения. В качестве параметра в нее передается дескриптор оператора ODBC, для которого необходимо установить опции.

Теперь рассмотрим класс, который позволяет работать непосредственно с наборами записей.

Класс CRecordset

Все компоненты и методы этого класса можно разбить на семь категорий.

Компоненты данных

В классе CRecordset определены следующие компоненты данных:

HSTMT CRecordset::**m_hstmt** —

содержит дескриптор оператора ODBC, ассоциированного с результирующим набором. Обычно нет необходимости использовать эту переменную непосредственно, однако она требуется при использовании функции ExecuteSQL класса CDatabase.

Примечание

Не используйте m_hstnt до вызова функции Open.

UINT CRecordset::**m_nFields** —

содержит число полей данных в результирующем наборе — число столбцов, получаемых из источника данных. Это поле должно быть корректно инициализировано в конструкторе класса CRecordset.

Примечание

Если использовать ClassWizard, то это будет сделано автоматически.

Это число должно соответствовать числу "выходных столбцов", зарегистрированных в функции DoFieldExchange вызовом метода SetFieldType с параметром CFieldExchange::outputColumn. Если есть необходимость в использовании переменного числа столбцов, то сделать это можно, изменив значение этой переменной.

UINT CRecordset::**m_nParams** —

содержит число параметров, посылаемых в параметризованном запросе. Если эта переменная используется, то она должна быть корректно инициализирована в конструкторе. По умолчанию переменная инициализируется нулем.

Это число должно соответствовать числу "выходных столбцов", зарегистрированных в функции DoFieldExchange вызовом метода SetFieldType с параметром CfieldExchange ::inputColumn, CFieldExchange::outputColumn, CFieldExchange::Column или CFieldExchange::inoutColumn.

CDatabase CRecordset::**m_pDatabase** —

содержит указатель на объект класса CDatabase, посредством которого результирующий набор соединяется с источником данных. Эта переменная устанавливается двумя способами: если вы уже установили соединение с источником данных, то при создании объекта класса CRecordset передайте туда указатель на объект класса CDatabase, или можно передать туда NULL — при этом CRecordset сам создаст объект CDatabase и соединится с ним. В обоих случаях указатель на базу данных хранится в этой переменной. И хотя обычно не требуется отслеживать состояние этой переменной, все же есть случаи, когда делать это необходимо, например, при запуске транзакции или непосредственного выполнения оператора SQL.

CString CRecordset::**m_strFilter** —

служит для определения предложения WHERE оператора SQL. Эта строка используется в качестве фильтра, что позволяет выбирать только записи, удовлетворяющие заданно-

му критерию. Чтобы осуществить "фильтрацию", эту переменную следует определять после создания объекта класса CRecordset, но до вызова функции Open, или "перечитать" запрос с помощью функции Requery.

Примечание

Не нужно включать в строку фильтра ключевое слово WHERE, поскольку библиотека подставляет его сама.

Эту переменную можно определять и для параметризованных запросов, но об этом немного ниже.

CString CRecordse t::**m_strSort** —

служит для определения предложения ORDER BY оператора SQL. Эта строка используется в качестве фильтра, что позволяет сортировать записи, удовлетворяющие заданному критерию. Чтобы осуществить "сортировку", эту переменную следует определять после создания объекта класса CRecordset, но до вызова функции Open, или "перечитать" запрос с помощью функции Requery.

Примечание

Не нужно включать в строку фильтра ключевое слово ORDER BY, поскольку библиотека подставляет его сама.

Помимо перечисления столбцов, по которым производится сортировка, можно указать также одно из ключевых слов (отдельно для каждого столбца), определяющего "направление" сортировки: ASC — по возрастанию и DESC — по убыванию.

Конструирование

В эту категорию входят всего три функции: конструктор, Open и Close, отвечающие за создание объекта, открытие и закрытие соединения с источником данных.

CRecordset::**CRecordset**(CDatabase* pDatabase = NULL) —

служит для создания и инициализации объекта класса CRecordset. В качестве параметра в конструктор можно передать либо указатель на открытую базу данных (параметр pDatabase), либо NULL, что говорит о необходимости создания базы данных по умолчанию.

Примечание

При создании класса, производного от CRecordset, у него должен быть только один конструктор, по параметрам совпадающий с базовым.

virtual BOOL CRecordset::**Open**(
UINT nOpenType = AFX_DB_USE_DEFAULT_TYPE,
LPCTSTR lpszSQL = NULL, DWORD dwOptions = none) —

nOpenType определяет тип доступа к источнику данных и может принимать следующие значения:

CRecordset::**dynaset** Результирующий набор с возможностью двунаправленного просмотра. При этом режиме изменения, вносимые в базу данных другими пользователями, отображаются сразу же. К сожалению, декларированное поведение результирующего набора при этом режиме не поддерживается

CRecordset::**snapshot** Статический результирующий набор с возможностью двунаправленного просмотра. При этом режиме изменения, вносимые в базу данных другими пользователями, не отображаются сразу, а только после закрытия и повторного открытия результирующего набора

CRecordset::**dynamic** Результирующий набор с возможностью двунаправленного просмотра. При этом режиме изменения, вносимые в базу данных другими пользователями, отображаются при выполнении следующей операции. Многие драйверы ODBC не поддерживают этот режим доступа

CRecordset::**forwardOnly** Результирующий набор "только для чтения" с возможностью просмотра "только вперед"

Для класса CRecordset значение этого параметра по умолчанию — CRecordset::snapshot.

Примечание

Если выбранный тип не поддерживается, то происходит обработка исключения.

lpszSQL — указатель на строку, содержащую одно из следующих значений:

- NULL
- Имя таблицы
- Оператор SQL, не обязательно с предложениями WHERE или ORDER BY
- Оператор CALL, определяющий имя предопределенного запроса или сохраненной процедуры

Примечание

Порядок столбцов в результирующем наборе должен совпадать с порядком вызова в переопределенной функции DoFieldExchange.

dwOptions — битовая маска, которая определяет комбинацию следующих значений:

CRecordset::**none** Опции не установлены. Это значение параметра не может использоваться с другими значениями. По умолчанию результирующий набор можно редактировать (Edit) и удалять (Delete), а также добавлять в него новые записи (AddNew). Возможность обновления зависит от источника данных, а также от режима открытия результирующего набора

CRecordset::**appendOnly** Нельзя редактировать и удалять записи результирующего набора. Можно только добавлять новые записи. Не может использоваться совместно со значением CRecordset::readOnly

CRecordset::**readOnly** Результирующий набор открывается в режиме "только для чтения". Не может использоваться совместно со значением CRecordset::appendOnly

CRecordset::**optimizeBulkAdd**	Для оптимизации добавления нескольких записей за один раз используется подготовленный оператор SQL. Это значение можно использовать только если для обновления результирующего набора не используется функция ODBC API SQLSetPos. Не может использоваться совместно со значением CRecordset::useMultiRowFetch
CRecordset::**useMultiRowFetch**	Позволяет извлекать несколько строк в одной операции выборки — блочная выборка строк. Не может использоваться совместно с CRecordset::optimizeBulkAdd. Если в качестве параметра установлена эта опция, то автоматически устанавливается опция CRecordset::noDirtyFieldCheck, а для результирующих наборов с возможностью просмотра "только вперед" (CRecordset::forwardOnly) устанавливается также опция CRecordset::useExtendedFetch
CRecordset::**skipDeletedRecords**	Пропускает все удаленные записи при перемещении по результирующему набору. Эта опция недоступна для результирующих наборов с возможностью просмотра "только вперед". Записи, удаленные другими пользователями в то время, пока результирующий набор открыт, не пропускаются
CRecordset::**useBookmarks**	Если источник данных поддерживает закладки (Bookmarks), то для работы с ними необходимо указать эту опцию. Закладки замедляют выборку данных, но вместе с тем улучшают производительность при перемещении по записям. Недоступна для результирующих наборов с возможностью просмотра "только вперед"
CRecordset::**noDirtyFieldCheck**	Выключает автоматическую проверку "грязных" полей (двойная буферизация), что позволяет улучшить производительность. При этом отмечать поля как "грязные" необходимо выполнять вручную с помощью функций SetFieldDirty и SetFieldNull. Для результирующих наборов недоступна двойная буферизация отдельных полей — действует принцип "или все, или ничего". Эта опция устанавливается автоматически, если задается опция CRecordset::useMultiRowFetch. Однако в этом случае нельзя использовать функции SetFieldDirty и SetFieldNull для результирующих наборов, поддерживающих блочную выборку строк
CRecordset::**executeDirect**	Не следует использовать эту опцию для подготовленных операторов SQL. Если не предполагается вызывать функцию Requery, то определение этой опции позволит повысить производительность

CRecordset::**useExtendedFetch**	Установка этой опции приводит к тому, что вместо SQLFetch выполняется SQLExtendedFetch. Она спроектирована для реализации блочной выборки строк из результирующих наборов с возможностью просмотра "только вперед" и устанавливается автоматически при задании опции CRecordset::useMultiRowFetch
CRecordset::**userAllocMultiRowBuffers**	При установке этой опции пользователю необходимо будет самому выделять память в буфере для хранения данных. Ее следует использовать вместе с опцией CRecordset::useMultiRowFetch

Эту функцию следует вызывать для выполнения запроса, определяющего результирующий набор. Очевидно, что объект класса CRecordset должен быть создан перед вызовом этой функции. Более того, соединение с источником данных зависит от того, как именно он создан.

- Если в качестве параметра был использован указатель на объект CDatabase, который еще не соединен с источником данных, то функция Open использует метод GetDefaultConnect для открытия объекта базы данных
- Если в качестве параметра использовался NULL, то автоматически создается объект класса CDatabase и осуществляется попытка соединения с базой данных

Примечание

Доступ к источнику данных посредством объекта CRecordset никогда не может быть эксклюзивным.

При вызове этой функции выполняется запрос, обычно оператор SELECT, в результате чего выбираются записи на основе критериев перечисленных ниже.

Значение параметра lpszSQL	Выбираемые записи определяются	Пример
NULL	строкой, возвращаемой функцией GetDefaultSQL	
Имя таблицы	столбцами, указанными в функции DoFieldExchange	"Authors"
Имя предопределенного запроса (сохраненной процедуры)	столбцами, определенными в запросе	"(call BooksList)"
SELECT список-столбцов FROM список-таблиц	заданными столбцами из определенных таблиц	SELECT AuthorID, FullName FROM Authors

Примечание

При написании строки SQL необходимо тщательно следить за тем, чтобы в нее не попали дополнительные пробелы. Например, если между круглой скобкой и ключевым словом CALL (а также SELECT) будет хоть один пробел, то библиотека MFC неправильно поймет имя таблицы, что приведет к обработке исключения (исключительной ситуации). Аналогичная ситуация возникает, если пробел появится между круглой скобкой и символом '?' в параметризованном запросе.

При успешном выполнении функции возвращается ненулевое значение, и 0 в противном случае. Если полученный результирующий набор не пустой, то текущей является первая запись.

Перед вызовом функции Open можно определить дополнительные условия, соответствующие параметрам предложений WHERE и ORDER BY оператора SQL. Если же их определить после того, как результирующий набор открыт, то они не окажут никакого влияния на результат выполнения запроса. В этом случае необходимо воспользоваться функцией Requery для обновления записей.

virtual void CRecordset::**Close**( ) —

служит для закрытия результирующего набора. Если он не был открыт, то функция просто завершается. После выполнения этой функции вся память, выделенная для результирующего набора, и дескриптор HSTMT ODBC, ассоциированный с ним, возвращаются системе. В том случае, когда объект класса CRecordset создавался с помощью оператора new, вызов функции Close осуществляется автоматически при удалении объекта.

Атрибуты результирующего набора

В эту категорию входят пятнадцать функций, работающих непосредственно с атрибутами CRecordset и позволяющими получить информацию о результирующем наборе.

BOOL CRecordset::**CanAppend**( ) const —

позволяет определить, можно ли добавлять записи в результирующий набор. Да — если она возвращает ненулевое значение, и нет — если 0. Для добавления записей необходимо вызвать функцию AddNew.

BOOL CRecordset::**CanBookmark**( ) const —

позволяет определить, можно ли отмечать записи в результирующем наборе с помощью закладок. Да — если она возвращает ненулевое значение, и нет — если 0. Результат работы функции не зависит от установки опции CRecordset::**useBookmarks** параметра dwOptions функции Open, а определяется только тем, поддерживает ли закладки используемый драйвер ODBC.

Примечание

Закладки не поддерживаются в результирующих наборах с возможностью просмотра "только вперед".

BOOL CRecordset::**CanRestart**( ) const —

позволяет определить, можно ли обновлять данные в результирующем наборе, используя для этого функцию Requery. Да — если она возвращает ненулевое значение, и нет — если 0.

BOOL CRecordset::**CanScroll**( ) const —

позволяет определить, можно ли перемещаться по записям в результирующем наборе. Да — если она возвращает ненулевое значение, и нет — если 0.

BOOL CRecordset::**CanTransact**( ) const —

позволяет определить, поддерживает ли результирующий набор транзакции. Да — если она возвращает ненулевое значение, и нет — если 0.

BOOL CRecordset::**CanUpdate**( ) const —

позволяет определить, можно ли обновлять записи в результирующем наборе. Да — если она возвращает ненулевое значение, и нет — если 0. Обновление не поддерживается, если источник данных "только для чтения" или если результирующий набор открыт в режиме "только для чтения" путем установки значения CRecordset::**readOnly** параметра dwOptions.

short CRecordset::**GetODBCFieldCount**( ) const —

возвращает общее число полей в текущем объекте класса CRecordset.

long CRecordset::**GetRecordCount**( ) const —

позволяет определить размер результирующего набора. Возвращает число записей в результирующем наборе, 0 — если он не содержит записей, или –1, если число записей не может быть определено.

Примечание

Общее число записей в результирующем наборе может быть определено только после того, как приложение переместилось за последнюю запись. С точки зрения производительности это число не обновляется при вызове функции MoveLast. Добавление записей посредством вызова функций AddNew и Update увеличивает это число, а удаление посредством функции Delete — уменьшает его.

const CString& CRecordset::**GetTableName**( ) const —

позволяет получить имя таблицы, на которой основывается запрос результирующего набора, или пустую строку. Эта функция работает корректно только в случае, если результирующий набор базируется на таблице, и не работает, если в основе лежит объединение нескольких таблиц или предопределенный запрос (сохраненная процедура).

const CString& CRecordset::**GetSQL**( ) const —

позволяет получить оператор SQL, который используется для выборки записей результирующего набора. Чаще всего это оператор SELECT. Возвращаемая функцией строка, естественно, отличается от той, которая передавалась в качестве параметра lpszSQL функции Open. Связано это с тем, что CRecordset создает полное предложение SQL, базируясь на информации, переданной функции Open, о том, что было определено с помощью ClassWizard, на значениях, записанных в переменные m_strFilter и m_strSort, а также на специфицированных параметрах запроса.

BOOL CRecordset::**IsOpen**( ) const —

позволяет определить, открыт ли уже результирующий набор. Хорошим стилем программирования является обращение к этой функции перед вызовом функции Open.

BOOL CRecordset::**IsBOF**( ) const —

позволяет определить, является ли текущая запись первой в наборе данных. Возвращает ненулевое значение, если результирующий набор не содержит записей или указатель помещен до первой записи, и 0 в противном случае. Если эта функция возвратила ненулевое значение, то текущая запись не определена и при вызове функции MovePrev возникает ошибка.

BOOL CRecordset::**IsEOF**( ) const —

позволяет определить, является ли текущая запись последней в наборе данных. Возвращает ненулевое значение, если результирующий набор не содержит записей или указатель помещен за последней записью, и 0 в противном случае. Если эта функция возвратила ненулевое значение, то текущая запись не определена, и при вызове функции MoveNext возникает ошибка.

BOOL CRecordset::**IsDeleted**( ) const —

позволяет определить, была ли текущая запись удалена. Если при перемещении на запись и вызове этой функции она возвращает ненулевое значение, необходимо перейти к другой записи до того, как производить какие-либо операции над результирующим набором.

Примечание

Эту функцию не следует использовать при работе с блочной выборкой строк. Вместо нее следует вызывать функцию GetRowStatus.

Примечание

Результат, возвращаемый функцией IsDeleted, зависит от многих факторов, таких как: тип результирующего набора, можно ли его обновлять, определена ли при открытии опция CRecordset::**skipDeletedRecords**, позволяет ли используемый драйвер удалять записи, число работающих пользователей.

Операции обновления результирующего набора

В эту категорию входят пять функций, которые позволяют добавлять, удалять и редактировать записи в результирующем наборе.

virtual void CRecordset::**AddNew**( ) —

служит для подготовки новой записи к добавлению в результирующий набор. Все поля записи исходно устанавливаются в NULL (в терминологии баз данных, что означает "не имеет значения" и отличается от смысла NULL в C++). Чтобы сохранить внесенные в эту пустую запись изменения в источнике данных, необходимо вызвать функцию Update.

Примечание

Если перейти к другой записи до вызова Update, новая запись теряется без какого-либо предупреждения.

Для динамического набора (dynaset) новая запись автоматически добавляется в конец результирующего набора. Для того чтобы увидеть эту новую запись в "мгновенном снимке", необходимо вызвать функцию Requery.

При попытке вызвать эту функцию для результирующего набора, который еще не открыт, а также если он не поддерживает режим обновления (проверяется вызовом функции CanUpdate), происходит обработка исключения.

virtual void CRecordset::**Delete**( ) —

удаляет текущую запись. После успешного удаления необходимо вызвать одну из функций перемещения по записям, чтобы корректно завершить операцию. Например, если этого не сделать и снова вызвать функцию Delete, то произойдет обработка исключения.

Примечание

В отличие от функций AddNew и Edit вызов функции Delete не требует вызова Update. Если по каким-либо причинам вызов этой функции завершился неудачно, поля данных просто остаются без изменений.

virtual void CRecordset::**Edit**( ) —

позволяет изменять поля текущей записи. Для корректного завершения операции обновления записи и сохранения изменений в источнике данных необходимо вызвать функцию Update. Если переместиться к другой записи до вызова Update, восстанавливаются предыдущие значения текущей записи. В случаях, когда требуется записать в некоторый столбец значение NULL (не содержит данных), необходимо вызвать функцию SetFieldNull с параметром TRUE. Если нужно оставить без изменений какое-либо поле, вызовите функцию SetFieldDirty с параметром TRUE, даже в том случае, если в нем было записано значение NULL.

В зависимости от текущего режима блокировки запись может быть блокирована либо пока не вызвана функция Update или до перемещения к другой записи, либо только во время вызова функции Edit. Режим блокировки можно изменить с помощью функции SetLockingMode.

virtual BOOL CRecordset::**Update**( ) —

вызов этой функции является обязательным для корректного завершения операций добавления и обновления записей в результирующий набор и возвращает ненулевое значение, если запись была успешно обновлена, и 0 в противном случае. Обе функции AddNew и Edit подготавливают специальный буфер, в котором, собственно, и происходит добавление и редактирование данных. Вызов функции Update сохраняет эти данные в источнике данных.

Примечание

Если вызвать функцию Update до вызова AddNew или Edit, возникает исключение.

Все рассмотренные выше функции (AddNew, Edit, Delete и Update) можно включать в транзакции, если работу с ними поддерживает результирующий набор.

void CRecordset::**CancelUpdate**( ) —

позволяет сбросить любое "повисшее" обновление, обусловленное операциями AddNew или Edit, до вызова функции Update. Если установлен режим автоматического отмечания поля как "грязного" (режим, устанавливаемый по умолчанию при открытии результирующего набора), то CancelUpdate восстанавливает переменные в те значения, которые они имели до вызова функции Edit или AddNew. В противном случае будут оставлены измененные значения (для этого необходимо установить параметр dwOptions в значение CRecordset::**noDirtyFieldCheck** при вызове функции Open).

Примечание

Использование функций AddNew, Edit, Delete, Update и CancelUpdate при работе с записями в режиме блочной выборки строк приведет к ошибке доступа. В классе CRecordset не реализован механизм для обновления блоков строк данных. В этом случае необходимо воспользоваться функциями ODBC API.

Операции перемещения по результирующему набору

К основным функциям этой категории относятся пять: Move, MoveFirst, MoveLast, MoveNext и MovePrev. Их мы и рассмотрим в первую очередь.

virtual void CRecordset::**Move**(long nRows,
WORD wFetchType = SQL_FETCH_RELATIVE) —

nRows — количество строк, на которое необходимо переместиться вперед (положительное значение) или назад (отрицательное); wFetchType — определяет набор строк, которые функция должна выбрать:

wFetchType	Выбираемый набор строк
SQL_FETCH_RELATIVE (значение по умолчанию)	Набор строк включает nRows строк, начиная с первой в текущем наборе
SQL_FETCH_NEXT	Следующий набор строк. Параметр nRows игнорируется
SQL_FETCH_PRIOR	Предыдущий набор строк. Параметр nRows игнорируется
SQL_FETCH_FIRST	Первый набор строк. Параметр nRows игнорируется
SQL_FETCH_LAST	Последний набор строк. Параметр nRows игнорируется
SQL_FETCH_ABSOLUTE	Если nRows > 0, то набор строк включает nRows строк, начиная с начала результирующего набора. Если nRows < 0, то набор строк включает nRows строк начиная с конца результирующего набора. Если nRows = 0, то возвращается BOF — начало файла
SQL_FETCH_BOOKMARK	Набор строк начинается со строки, значение закладки которой соответствует nRows

Если результирующий набор не содержит записей, или предпринимается попытка переместиться за начало или конец результирующего набора, — возникает исключение.

Примечание

Для результирующего набора с возможностью просмотра "только вперед" разрешено только одно значение параметра wFetchType — SQL_FETCH_NEXT.

void CRecordset::**MoveFirst**( ) —

делает текущей первую запись результирующего набора. Перед ее использованием рекомендуется вызвать функцию IsBOF.

void CRecordset::**MoveLast**( ) —

делает текущей последнюю запись результирующего набора. Перед ее использованием рекомендуется вызвать функцию IsEOF.

void CRecordset::**MoveNext**( ) —

делает текущей первую запись следующего набора строк. Перед ее использованием рекомендуется вызвать функцию IsBOF. При работе с отдельными строками (не с блоками строк) текущей становится следующая запись.

void CRecordset::**MovePrev**( ) —

делает текущей первую запись предыдущего набора строк. Перед ее использованием рекомендуется вызвать функцию IsEOF. При работе с отдельными строками (не с блоками строк) текущей становится предыдущая запись.

Примечание

Для результирующего набора с возможностью просмотра "только вперед" эти функции не работают.

Последняя из рассматриваемых функций этой категории позволяет установить текущую запись по ее абсолютному положению в результирующем наборе:

void CRecordset::**SetAbsolutePosition**(long nRows) —

перемещает указатель на текущую запись, базируясь на ее порядковом номере (nRows). При этом отсчет начинается с единицы.

Примечание

Для результирующего набора с возможностью просмотра "только вперед" эта функция не работает.

Если задать отрицательное значение параметра nRows, то отсчет начинается с последней записи.

Примечание

Пользоваться этой функцией следует с большой осторожностью, поскольку при добавлении и удалении записей нельзя гарантировать сохранение абсолютной позиции записи. Для перехода к конкретной записи лучше использовать закладки.

Другие операции над результирующим набором

В эту категорию входят функции, которые трудно отнести к какому-то одному типу.

void CRecordset::**Cancel**( ) —

"просьба" к источнику данных отменить либо обрабатываемую асинхронную операцию, либо второй поток. Библиотека MFC не поддерживает асинхронные операции, и для них необходимо непосредственно вызвать функцию SQLSetConnectOption из ODBC API.

DWORD CRecordset::**GetRowsetSize**( ) const —

возвращает число строк, которые можно получить во время выборки. Для блочной выборки значение по умолчанию для результирующего набора установлено в значение 25. Изменить эти значения можно с помощью функции SetRowsetSize. Для обычной выборки значение по умолчанию равно 1.

DWORD CRecordset::**GetRowsFetched**( ) const —

позволяет определить, сколько строк действительно получено после выборки. Это значение зависит от размера набора строк. Например, если в результирующем наборе 10 записей, а размер набора строк равен 4, то функция GetRowsetFetched возвращает значение 4 для первых двух наборов и 2 — для последнего.

WORD CRecordset::**GetRowStatus**(WORD wRow) const —

позволяет получить значение состояния строки, заданной ее позицией в наборе строк (wRow), которая может принимать значение от 1 до размера набора строк. Возвращаемые функцией значения:

SQL_ROW_SUCCESS	Строка не изменилась
SQL_ROW_UPDATED	Строка была обновлена
SQL_ROW_DELETED	Строка была удалена
SQL_ROW_ADDED	Строка была добавлена
SQL_ROW_ERROR	Строка не получена из-за ошибки
SQL_ROW_NOROW	Нет строки, соответствующей заданному номеру wRow

BOOL CRecordset::**IsFieldDirty**(void* pv) —

возвращает ненулевое значение, если поле данных было изменено после вызова AddNew или Edit; в противном случае 0. Проверяемое поле данных задается указателем pv; если это значение равно NULL, то проверяются все поля.

BOOL CRecordset::**IsFieldNull**(void* pv) —

возвращает ненулевое значение, если поле данных отмечено как содержащее значение NULL (в терминологии баз данных) и 0 в противном случае. Проверяемое поле данных задается указателем pv; если это значение равно NULL, то проверяются все поля.

BOOL CRecordset::**IsFieldNullable**(void* pv) —

возвращает ненулевое значение, если поле данных допускает значение NULL (в терминологии баз данных) и 0 в противном случае. Проверяемое поле данных задается указателем pv; если это значение равно NULL, то проверяются все поля.

void CRecordset::**RefreshRowset**(WORD wRow, WORD wLockType = SQL_LOCK_NO_CHANGE) —

обновляет данные и состояние строки, заданной параметром wRow, в текущем наборе строк. Если значение wRow равно 0, то обновляется каждая строка. Параметр wLockType показывает состояние блокировки строки после обновления и может принимать одно из следующих значений:

SQL_LOCK_NO_CHANGE (значение по умолчанию)	Драйвер или источник данных гарантирует, что строка останется в том же состоянии, в котором она находилась перед вызовом функции
SQL_LOCK_EXCLUSIVE	Драйвер или источник данных блокирует строку эксклюзивно. Не все источники данных поддерживают этот тип блокировки
SQL_LOCK_UNLOCK	Драйвер или источник данных не блокирует строку. Не все источники данных поддерживают этот тип блокировки

virtual BOOL CRecordset::**Requery**( ) —

позволяет обновить результирующий набор. Ее следует вызывать для результирующего набора типа "мгновенный снимок" (snapshot) после того, как вы или другой пользователь внесли в него изменения. Для динамического (dynaset) результирующего набора обновление производится автоматически, кроме случая добавления новой записи. Кроме этого, ее следует вызывать при изменении значений параметров m_strFilter и/или m_strSort. При параметризованном запросе также следует использовать эту функцию. Если функция возвращает нулевое значение — ошибка, то результирующий набор закрывается.

Примечание

Функцию Requery можно вызывать только для открытого результирующего набора. Проверить это можно с помощью функции CanRestart.

void CRecordset::**SetFieldNull**(void* pv, BOOL bNull = TRUE) —

позволяет пометить поле данных результирующего набора (параметр pv) как содержащее (bNull = TRUE) или не содержащее (bNull = FALSE) значение NULL. Если в качестве pv передается NULL, помечаются все поля.

При добавлении новой записи в результирующий набор все поля исходно устанавливаются в значение NULL и помечаются как измененные.

Примечание

Функции SetFieldDirty и SetFieldNull следует вызывать только после вызова функций AddNew или Edit.

void CRecordset::**SetLockingMode**(UINT nMode) —

позволяет установить определенный тип блокировки, задаваемый параметром nMode:

Оптимистическая (optimistic)	Блокирование записи осуществляется только во время вызова функции Update
Пессимистическая (pessimistic)	Блокирование записи начинается при вызове функции Edit и сохраняется до того момента, пока изменения не будут корректно завершены вызовом функции Update

По умолчанию для результирующего набора установлена оптимистическая блокировка. Изменить тип блокировки можно в любой момент после создания объекта и открытия результирующего набора, но до вызова функции Edit.

void CRecordset::**SetRowsetCursorPosition**(WORD wRow, WORD wLockType = SQL_LOCK_NO_CHANGE) —

перемещает курсор на строку в наборе строк, заданную параметром wRow. Параметр wLockType позволяет задать влияние выполнения этой функции на состояние блокировки строк. (Допустимые значения см. в описании функции RefreshRowset.)

Переопределяемые методы

В эту категорию входят восемь функций, которые можно (и нужно) переопределить в каждом конкретном приложении, чтобы настроить его на выполнение текущих задач.

virtual BOOL CRecordset::**Check**(RETCODE nRetCode) const —

позволяет проверить значение, возвращаемое функциями ODBC API. Она возвращает ненулевое значение, если nRetCode равно SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NO_DATA_FOUND или SQL_NEED_DATA; в противном случае 0.

virtual void CRecordset::**CheckRowsetError**(RETCODE nRetCode) —

аналогична функции Check, но отслеживает выполнение операций с блочными наборами строк. Полный список допустимых значений параметра nRetCode обоих функций приведен ниже:

SQL_SUCCESS	Функция завершилась успешно; дополнительной информации нет
SQL_SUCCESS_WITH_INFO	Функция завершилась успешно, возможно с фатальной ошибкой; дополнительную информацию можно получить с помощью функции SQLError
SQL_NO_DATA_FOUND	Из результирующего набора выбраны все строки
SQL_ERROR	Функция завершилась с ошибкой. Дополнительную информацию можно получить с помощью функции SQLError
SQL_INVALID_HANDLE	Функция завершилась с ошибкой, вследствие того, что указан неправильный дескриптор окружения, соединения или оператора. Дополнительной информации нет
SQL_STILL_EXECUTING	Функция, которая запущена асинхронно, все еще выполняется. При использовании библиотеки MFC это значение никогда не возвращается, поскольку она использует только синхронную обработку
SQL_NEED_DATA	Во время обработки оператора драйвер определил, что приложению необходимо послать значения данных параметров

Следующие две функции очень похожи и различаются, в основном, тем, что одна работает с "обычным" результирующим набором, а вторая с блочным:

virtual void CRecordset::**DoFieldExchange**(CFieldExchange* pFX)

и

virtual void CRecordset::**DoBulkFieldExchange**(CFieldExchange* pFX) —

библиотека классов автоматически вызывает одну из этих функций для организации обмена данными между полями результирующего набора и соответствующими столбцами текущей записи в источнике данных.

Примечание

Эти функции доступны только в том случае, если для результирующего набора используется класс производный от CRecordset. Если же используется непосредственно класс CRecordset, то для получения данных необходимо использовать функцию GetFieldValue.

В качестве параметра функции принимают указатель на объект CFieldExchange, который автоматически создается и передается библиотекой классов MFC.

Собственно обмен данными осуществляется с помощью механизма RFX (record field exchange — обмен полями записи), который работает в обоих направлениях: от полей данных результирующего набора к записям источника данных и наоборот

Примечание

ClassWizard не поддерживает обмена полями блочных записей и необходимо переопределить соответствующую функцию вручную, определив в ней вызовы функций Bulk RFX (блочный обмен полями записи).

virtual CString CRecordset::**GetDefaultConnect**( ) —

библиотека классов MFC вызывает эту функцию, чтобы получить строку, содержащую источник данных, на котором базируется результирующий набор.

virtual CString CRecordset::**GetDefaultSQL**( ) —

библиотека классов MFC вызывает эту функцию, чтобы получить строку, содержащую оператор SQL, на котором базируется результирующий набор. Это должно быть или имя таблицы, или непосредственно оператор SELECT. Можно также определить здесь вызов предопределенного запроса, используя оператор CALL.

Примечание

Между круглой скобкой и ключевым словом, например CALL или SELECT, не должно быть пробелов.

Если библиотека MFC не сможет найти имя таблицы, или корректно интерпретировать оператор CALL, функция возвращает пустую строку.

virtual void CRecordset::**OnSetOptions**(HSTMT hstmt) —

функция вызывается библиотекой MFC и позволяет установить опции результирующего набора, заданного дескриптором HSTMT. Реализация этой функции по умолчанию проверяет, поддерживает ли источник данных курсоры, и устанавливает соответствующие опции. Если необходимо установить еще какие-нибудь опции, следует переопределить эту функцию. Например, если источник данных поддерживает эксклюзивный доступ, то для реализации этой возможности необходимо в ней установить соответствующую опцию.

virtual void CRecordset::**SetRowsetSize**(DWORD dwNewRowsetSize) —

позволяет задать, сколько строк (параметр dwNewRowsetSize) можно получить за одну блочную выборку.

Примечание

Вызов функции SetRowsetSize без реализации поддержки выборки блоков строк приведет к ошибке.

Получить текущее значение размера набора строк можно с помощью функции GetRowsetSize.

Рассмотренные классы безусловно существенно облегчают работу с результирующими наборами, однако без предоставления этой информации пользователю, да еще в удобном виде, она мало что значит. К счастью, в библиотеке MFC реализовано множество классов представления (view), которые делают эту работу достаточно простой.

Глава 34

Использование расширения ISAPI

Теперь, когда вы знакомы с классами для работы с базами данных, можно приступать к усовершенствованию нашего приложения. В качестве базы данных для этого примера мы воспользовались базой данных Northwind, входящей в поставку SQL Server 7.0. Связано это с тем, что разработка и создание базы данных — отдельная большая тема, которая выходит за рамки данной книги.

Рис. 34.1. Исходная страница созданного расширения

Прежде чем привести исходные тексты, обрисуем в двух словах, что именно мы собираемся делать. В качестве основы используем созданное в главе 32 приложение, которое по умолчанию "показывает" пользователю страницу (рис. 34.1), где он должен выбрать, какую именно информацию он хочет получить, — о сотрудниках организации, или о продаваемых ею продуктах. Выбрав соответствующий переключатель и нажав кнопку **Go**, пользователь должен увидеть страницу с результатами выборки запрошенной информации из базы данных.

Если вспомнить приведенный в главе 32 код, то становится ясно, что надо внести изменения в следующий фрагмент, заменив вывод текстов "заглушек" на нечто более содержательное:

```
if (_tcscmp(pszMethod, _T("ByEmployees")) == 0)
  {
    *pCtxt << _T("First page sample wwwDB\r\n");
  }
  else if (_tcscmp(pszMethod, _T("ByProducts")) == 0)
  {
    *pCtxt << _T("Second page sample wwwDB\r\n");
  }
```

После этого небольшого вступления, приведем собственно фрагменты кода.

Примечание

Приводить полные тексты мы не будем — желающие найдут их на сопроводительной дискете, а остановимся только на наиболее интересных фрагментах.

```
class CEmployees : public CRecordset
{
public:
  CEmployees(CDatabase* pDatabase = NULL);
  DECLARE_DYNAMIC(CEmployees)
// При создании класса ClassWizard создает переменные, соответствующие
// всем столбцам таблицы. Нам остается только "убрать лишние"
// Field/Param Data
  //{{AFX_FIELD(CEmployees, CRecordset)
  long     m_EmployeeID;
  CString  m_LastName;
  CString  m_FirstName;
  CString  m_Title;
  CString  m_Address;
  CString  m_City;
```

```
  CString  m_Country;
  CString  m_HomePhone;
  //}}AFX_FIELD
...
};

class CProduct : public CRecordset
{
public:
  CProduct(CDatabase* pDatabase = NULL);
  DECLARE_DYNAMIC(CProduct)
// Если при создании класса было указано несколько таблиц,
// то все поля всех таблиц помещаются здесь: Field/Param Data
  //{{AFX_FIELD(CProduct, CRecordset)
  long     m_ProductID;   // Это поля таблицы Employees
  CString  m_ProductName;
  CString  m_UnitPrice;
  CString  m_CompanyName; // А это — таблицы Suppliers
  //}}AFX_FIELD
...
};
CString CEmployees::GetDefaultSQL()
{
  CString strQuery;
  // Создаем свой запрос, где перечисляем только то,
  // что нам конкретно нужно
  strQuery.Format("SELECT DISTINCT EmployeeID, LastName, FirstName,
                                   Title, Address, City, Country,
                                   HomePhone"
                  "FROM Employees");
  return strQuery;
}
CString CProduct::GetDefaultSQL()
{
  CString strQuery;
  // Создаем свой запрос, где объединяем в один результирующий набор
  // данные из двух таблиц
  strQuery.Format("SELECT DISTINCT ProductID, ProductName,
                                   UnitPrice, CompanyName"
```

```
                    "FROM Products, Suppliers"
                    "WHERE Suppliers.SupplierID = Products.SupplierID"
                    "ORDER BY ProductID");
  return strQuery;
}
void CWwwdb::Search(CHttpServerContext* pCtxt, LPCTSTR pszMethod)
{
  // Начинаем страницу
  StartContent(pCtxt);
  // Создаем "верхнюю" часть страницы,
  // ее описание находится в ресурсе IDS_SEARCH_TITLE
  WritePageTitle(pCtxt, IDS_SEARCH_TITLE);
  // В зависимости от выбора пользователя посылаем ему страницу
  // либо со списком сотрудников, либо со списком продуктов
  if (_tcscmp(pszMethod, _T("ByEmployees")) == 0)
    SearchEmployes(pCtxt);
  else if (_tcscmp(pszMethod, _T("ByProducts")) == 0)
    SearchProducts(pCtxt);
  // Заканчиваем страницу
  EndContent(pCtxt);
}
void CWwwdb::SearchEmployes(CHttpServerContext *pCtxt)
{
  CString strOutput;
  // При создании соединения с базой данных минимум, что требуется, —
  // это источник данных, идентификатор пользователя и его пароль
  CString strConnect(_T("DSN=Sample InternetDB;UID=sa;PWD=;"));
  // Ошибки будем обрабатывать с помощью механизмов
  // обработки исключений
  try  {
    // Создаем объект "база данных"
    CDatabase db;
    // Создаем соединение созданного объекта с "реальной" базой данных
    if (!db.OpenEx(strConnect, CDatabase::openReadOnly |
                               CDatabase::noOdbcDialog))    {
      // Если возникли проблемы, — сообщаем об этом пользователю
      *pCtxt <<
_T("ODBC Datasource 'Sample InternetDB' could not be opened\r\n");
      *pCtxt <<
```

```
_T("<br>Did you set it up as a <i>system</i> data source?<br>\r\n");
    }
    else    {
      // Если соединение с базой данных установлено —
      // создаем объект "результирующий набор"
      CEmployees employees(&db);
      // и, конечно же, пытаемся его открыть
      if (employees.Open())      {
        if (employees.IsBOF())
          *pCtxt <<
            _T("No employees are available at this time.<br>\r\n");
        else        {
          BOOL bFirstRow = TRUE;

          while (!employees.IsEOF())          {
            if (bFirstRow == TRUE)            {
              // Если прочитанная запись первая —
              // формируем заголовок таблицы,
              // в которую будут выводиться результаты запроса
              WriteSearchHeader(pCtxt);
              bFirstRow = FALSE;
            }
            // Формируем строку в формате HTML
            strOutput.Format(IDS_EMPLROW,
                             employees.m_EmployeeID,
                             (LPCTSTR) employees.m_LastName,
                             (LPCTSTR) employees.m_FirstName,
                             (LPCTSTR) employees.m_Title,
                             (LPCTSTR) employees.m_Address,
                             (LPCTSTR) employees.m_City,
                             (LPCTSTR) employees.m_Country,
                             (LPCTSTR) employees.m_HomePhone);

            // И выводим ее на страницу
           *pCtxt << strOutput;
            // Переходим к следующей записи
            employees.MoveNext();
          }
        }
```

```
        // После того как все записи прочитаны, —
        // закрываем результирующий набор
        employees.Close();
      }
      else
        // Об ошибке открытия объекта сообщаем пользователю
        *pCtxt << _T("Couldn't open record set<br>\r\n");

      // Закрываем соединение с базой данных
      db.Close();
    }
  }
  // В случае непредвиденных ситуаций обрабатываем возникшее исключение
  catch (CDBException* pEx)
  {
    *pCtxt << _T("WWWDB Failed: Couldn't open data source\r\n");
    *pCtxt << strConnect;
    *pCtxt << _T("\r\n");
    TCHAR szError[1024];
    if (pEx->GetErrorMessage(szError, sizeof(szError)))    {
      *pCtxt << szError;
      *pCtxt << _T("\r\n");
    }
  }
  // После вывода всех записей корректно закрываем таблицу
  *pCtxt << _T("</TABLE>");
}
void CWwwdb::SearchProducts(CHttpServerContext *pCtxt)
{
  // При выводе информации о продуктах принцип обработки
  // такой же, как и в предыдущем случае
  CString strOutput;
  CString strConnect(_T("DSN=Sample InternetDB;UID=sa;PWD=;"));
  try  {
    CDatabase db;
    if (!db.OpenEx(strConnect, CDatabase::openReadOnly |
                               CDatabase::noOdbcDialog))    {
      *pCtxt <<
_T("ODBC Datasource 'Sample InternetDB' could not be opened\r\n");
```

```
      *pCtxt <<
_T("<br>Did you set it up as a <i>system</i> data source?<br>\r\n");
    }
    else    {
      CProduct product(&db);

      if (product.Open())      {
        if (product.IsBOF())
          *pCtxt <<
     _T("Sorry, no products are available at this time.<br>\r\n");
        else        {
          BOOL bFirstRow = TRUE;
          while (!product.IsEOF())          {
            if (bFirstRow == TRUE)            {
              WriteProductHeader(pCtxt);
              bFirstRow = FALSE;
            }
            // Формируем строку в формате HTML
            strOutput.Format(IDS_PRODROW,
                             product.m_ProductID,
                             (LPCTSTR) product.m_ProductName,
                             (LPCTSTR) product.m_UnitPrice,
                             (LPCTSTR) product.m_CompanyName);
            // и выводим ее на страницу
            *pCtxt << strOutput;
            product.MoveNext();
          }
        }
        product.Close();
      }
      else
       *pCtxt << _T("Couldn't open record set<br>\r\n");

      db.Close();
    }
  }
  catch (CDBException* pEx)
  {
   // Обрабатываем исключение
```

```
  }
  // После вывода всех записей корректно закрываем таблицу
  *pCtxt << _T("</TABLE>");
}
```

Теперь обещанные комментарии. Прежде всего, необходимо создать источник данных ODBC. Сделать это достаточно просто — воспользуйтесь приложением ODBC Control Panel системы Windows. В данном случае мы назвали источник данных "Sample InternetDB".

После создания источника данных можно приступать к созданию новых классов. Запустите AppWizard и нажмите кнопку **New class**. В появившемся диалоговом окне введите имя класса — в нашем случае это CEmployees и CProduct, и выберите в качестве базового — CRecordset. Вот собственно и все.

Примечание

При создании класса CProduct в соответствующем диалоговом окне нужно выбрать две таблицы: Products и Suppliers, чтобы не заниматься этим вручную и посылать пользователю названия компаний, поставляющих продукты, а не их идентификаторы.

Следующим нашим шагом будет изменение созданной AppWizard функции GetDefaulSQL в соответствии с представленным выше кодом. Можно было бы оставить ее без изменений — но мы демонстрируем заодно и способ формирования собственного запроса.

Осталось только подключить созданные классы к нашему приложению. Для этого мы реализовали функции SearchEmployes и SearchProducts. Рассмотрим вторую подробно.

Прежде всего необходимо создать объект CDatabase и соединить его с источником данных Sample InternetDB. За это отвечает следующий фрагмент кода:

```
CString strConnect(_T("DSN=Sample InternetDB;UID=sa;PWD=;"));
CDatabase db;
if (!db.OpenEx(strConnect, CDatabase::openReadOnly |
                           CDatabase::noOdbcDialog))
```

Если все прошло успешно, можно создавать и открывать результирующий набор

```
CProduct product(&db);
if (product.Open())
  ...
```

После этого мы имеем практически неограниченный доступ к полученному из базы данных результирующему набору. Теперь требуется "организовать

место", куда мы будем помещать полученные записи. Делаем мы это с помощью функции:

```
void CWwwdb::WriteProductHeader(CHttpServerContext* pCtxt)
{
  CString strOutput;
  // Загружаем текст страницы из ресурса IDS_PRODUCT_HEADER
  LoadLongResource(strOutput, IDS_PRODUCT_HEADER);

  // и посылаем ее пользователю
  *pCtxt << strOutput;
}
```

Для большей ясности приведем сам текст этого ресурса:

```
<H1>Products
<HR>
<P>
<TABLE BORDER WIDTH="100%">
  <TR>
    <TD><B>ProductID</B></TD>
    <TD><B>ProductName</B></TD>
    <TD><B>UnitPrice</B></TD>
    <TD><B>CompanyName</B></TD>
  </TR>
```

Примечание

К сожалению, объем книги по техническим причинам ограничен и не позволяет нам подробнее остановиться на структуре и тэгах HTML-документов. Тем, кто не знаком с этими вопросами, мы рекомендуем обратиться к соответствующей литературе.

Здесь мы создаем таблицу, в первой строке которой размещаем названия столбцов. Затем помещаем полученные результаты последовательно в строки таблицы:

```
  strOutput.Format(IDS_PRODROW,
                   product.m_ProductID,
                   (LPCTSTR) product.m_ProductName,
                   (LPCTSTR) product.m_UnitPrice,
                   (LPCTSTR) product.m_CompanyName);
  *pCtxt << strOutput;
```

Полученная процедура повторяется для каждой записи результирующего набора. Формат вывода задан опять же в ресурсе:

```
<TR>
  <TD>%d</TD>
  <TD>%s</TD>
  <TD>%s</TD>
  <TD>%s</TD>
</TR>
```

Перемещение по записям осуществляется вызовом функции:

```
product.MoveNext();
```

После того как все записи выведены, о чем нам "сообщит" вызов функции:

```
product.IsEOF()
```

остается только корректно "закрыть" таблицу:

```
*pCtxt << _T("</TABLE>");
```

Вот, собственно, и все. На рис. 34.2 и 34.3 приведены результаты работы созданного нами расширения.

EmployeeID	LastName	FirstName	Title	Address	City	Country	HomePhone
1	Davolio	Nancy	Sales Representative	507 - 20th Ave. E. Apt. 2A	Seattle	USA	(206) 555-9857
2	Fuller	Andrew	Vice President, Sales	908 W. Capital Way	Tacoma	USA	(206) 555-9482
3	Leverling	Janet	Sales Representative	722 Moss Bay Blvd.	Kirkland	USA	(206) 555-3412
4	Peacock	Margaret	Sales Representative	4110 Old Redmond Rd.	Redmond	USA	(206) 555-8122
5	Buchanan	Steven	Sales Manager	14 Garrett Hill	London	UK	(71) 555-4848
6	Suyama	Michael	Sales Representative	Coventry House Miner Rd.	London	UK	(71) 555-7773
7	King	Robert	Sales Representative	Edgeham Hollow Winchester Way	London	UK	(71) 555-5598
8	Callahan	Laura	Inside Sales Coordinator	4726 - 11th Ave. N.E.	Seattle	USA	(206) 555-1189
9	Dodsworth	Anne	Sales Representative	7 Houndstooth Rd.	London	UK	(71) 555-4444

Рис. 34.2. Страница со списком сотрудников

MFC Internet Database Sample: MFC Internet Database Sample - List Employees - Microsoft Internet Explorer

File Edit View Go Favorites Help

Back Forward Stop Refresh Home Search Favorites History Channels Fullscreen Mail Print

Address http://localhost/scripts/wwwdb.dll? Links

Products

ProductID	ProductName	UnitPrice	CompanyName
1	Chai	18.0000	Exotic Liquids
2	Chang	19.0000	Exotic Liquids
3	Aniseed Syrup	10.0000	Exotic Liquids
4	Chef Anton's Cajun Seasoning	22.0000	New Orleans Cajun Delights
5	Chef Anton's Gumbo Mix	21.3500	New Orleans Cajun Delights
6	Grandma's Boysenberry Spread	25.0000	Grandma Kelly's Homestead
7	Uncle Bob's Organic Dried Pears	30.0000	Grandma Kelly's Homestead
8	Northwoods Cranberry Sauce	40.0000	Grandma Kelly's Homestead
9	Mishi Kobe Niku	97.0000	Tokyo Traders
10	Ikura	31.0000	Tokyo Traders
11	Queso Cabrales	21.0000	Cooperativa de Quesos 'Las Cabras'
12	Queso Manchego La Pastora	38.0000	Cooperativa de Quesos 'Las Cabras'
13	Konbu	6.0000	Mayumi's
14	Tofu	23.2500	Mayumi's

Local intranet zone

Рис. 34.3. Страница со списком продуктов и названия компаний, которые их поставляют

Описание сопроводительной дискеты

Сопроводительная дискета содержит самораспаковывающийся архив файлов с примерами приложений, приведенными в книге.

Установка примеров

Установку примеров на ваш компьютер можно провести двумя способами:

1. Вставьте дискету в дисковод и запустите файл Install.exe, находящийся на ней.
2. В появившемся диалоговом окне задайте путь к папке, в которую необходимо установить примеры, и запустите его.

Либо

1. Скопируйте файл Install.exe с дискеты в папку, в которую будут установлены примеры, и запустите его.
2. В появившемся диалоговом окне нажмите кнопку Extract.

Использование примеров

Полные тексты программ примеров приведены в одноименных каталогах.

- Для использования большинства примеров, а также для их модификации или перекомпиляции требуется Visual C++ версии не ниже 4.0, а в некоторых случаях — Visual C++ 6.0

Внимание

Несовместимость русскоязычной и американской (пан-европейской) версий Windows в поддержке национальных языков требует для использования данных примеров в версиях Windows, отличных от русскоязычной, исключить директивы "Russian resources" из исходных файлов ресурсов всех приложений до компиляции.

Эти изменения следует произвести вне системы разработки Visual C++ (Microsoft Developer Studio). Следует исключить следующие строки:

```
/////////////////////////////////////////////////////////////////////////////
// Russian resources
#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_RUS)
#ifdef _WIN32
LANGUAGE LANG_RUSSIAN, SUBLANG_DEFAULT
```

```
#pragma code_page(1251)
#endif //_WIN32
...
Здесь указаны описания ресурсов
...
#endif    // Russian resources
/////////////////////////////////////////////////////////////////////////////
```

В противном случае названия пунктов меню и текст элементов блоков диалога будет невозможно прочитать, а русский текст в файлах ресурсов после компиляции будет заменен символами вопроса (?); исполняемые файлы созданы в рамках русскоязычной версии Windows, поэтому названия пунктов меню и текст элементов управления блоков диалога будут заменены символами подчеркивания (_).

- Для просмотра содержимого файлов необходимо пользоваться *только* средствами Windows, т. к. комментарии, добавленные авторами, написаны на русском языке, естественно, в кодировке, отличной от используемой в MS-DOS

Примеры приложений

Animate (Animate)	Использование элемента просмотра видеоклипов
Bitmap (16-bit) (CBmpOne)	Создание контекстнозависимого битового массива в памяти
Bitmap (DIB) (CBmpTwo)	Создание контекстнонезависимого битового массива в памяти
Cap Join (CapJoin)	Вывод линий, созданных с различными типами соединений
Check Box (CheckBox)	Список переключателей, содержимое элементов которого отображается приложением
Clip Path (ClipPath)	Заполнение области отсечения, границы которой определены текстовой строкой
ControlBars	Создание и использование панелей элементов управления
Create Font (CrFont)	Создание шрифтов для отображения строки текста под различными углами
DlgSmpl	Приложение, главным окном которого является блок диалога
DragList Box (DragList)	Список, позволяющий копировать его элементы
Dynamic Menu (DynMenu)	Динамическое создание меню
Fill Regions (Region)	Заполнение области отсечения, полученной в результате комбинирования различных регионов

First	Минимальное приложение на базе классов библиотеки MFC
Graphics	Создание 10 потоков, каждый из которых имеет отличающийся относительный приоритет
Header (Header)	Создание заголовка пользовательской таблицы
Hot Key (HotKey)	Назначение и использование клавиш вызова различных типов
Image List (ImageDrag)	Реализация операции drag-and-drop для элементов списка изображений
Inheritance	Демонстрация основных принципов создания наследуемых потоков и дочерних процессов
Line Styles (LnStyle)	Вывод линий различного типа
List Box (LBSmpl)	Самоотображаемый список, каждый элемент которого содержит значок и многострочные пояснения. Каждый элемент списка может иметь свою высоту
List View (ImageDrag)	Пример из главы 25, использование элемента управления "просмотр списка" с реализацией операции drag-and-drop для его элементов
MDI	Пример из главы 12 (MDI) многодокументное приложение
Note'n'Draw (Notedraw)	MDI-приложение на базе классов библиотеки MFC, реализующих архитектуру "документ/представление" (document/view)
OEM Bitmaps (OEMBmp)	Отображение системных битовых массивов
PQ Clock (PQClock)	Отображение действующих часов. В этом примере использовано 90% всех возможностей графического вывода
Progress Bar (InProg)	Размещение индикатора в нижней части родительского окна, не являющегося блоком диалога
RichEdit	Пример приложения на основе расширенного редактора, который даже "в чистом виде" поддерживает возможности форматирования символов текста и абзацев
Static (Static)	Переопределение цветов для отображения статического элемента управления
Styles	Использование различных стилей при создании окон
Template	Создание блока диалога непосредственно в памяти, т. е. без использования ресурсов
Terminal	Пример организации взаимодействия с последовательным портом
UIThread	Демонстрация принципов создания интерфейсных потоков и обмена сообщениями между ними
wwwdb	Демонстрация принципов создания приложения для работы с базами данных через Internet

Предметный указатель

A

ACN_START, 686
ACN_STOP, 686
ACS_AUTOPLAY, 685
ACS_CENTER, 685
ACS_TRANSPARENT, 685
AfxBeginThread, 868; 876; 895
AfxEndThread, 897
AfxGetApp, 124
AfxGetInstanceHandle, 125
AfxMesageBox, 131
AfxRegisterClass, 134
AfxRegisterWndClass, 134
ALTERNATE, 438
AnsiToOem, 627; 651
ANSI_CHARSET, 462; 477
ANSI_FIXED_FONT, 436
ANSI_VAR_FONT, 436
ARABIC_CHARSET, 462; 477

B

BALTIC_CHARSET, 462; 477
BEGIN_PARSE_MAP, 956
BLACK_BRUSH, 436
BLACK_PEN, 436
BLACKNESS, 453
BLACKONWHITE, 439
BS_3STATE, 620
BS_AUTO3STATE, 620
BS_AUTOCHECKBOX, 620
BS_AUTORADIOBUTTON, 620
BS_BITMAP, 621
BS_BOTTOM, 621
BS_CENTER, 621
BS_CHECKBOX, 620
BS_DEFPUSHBUTTON, 620
BS_FLAT, 621
BS_GROUPBOX, 620
BS_ICON, 621
BS_LEFT, 621
BS_LEFTTEXT, 620
BS_MULTILINE, 621
BS_NOTIFY, 621
BS_OWNERDRAW, 621; 664
BS_PUSHBUTTON, 619
BS_PUSHLIKE, 621
BS_RADIOBUTTON, 620
BS_RIGHT, 621
BS_RIGHTBUTTON, 621
BS_TEXT, 621
BS_TOP, 621
BS_VCENTER, 621

C

CAnimateCtrl
  Close, 685
  Create, 685
  Open, 685
  Play, 686
  Seek, 686
  Stop, 686
CArchive
  Abort, 542
  bNoFlushOnDelete, 542
  CArchive, 541
  Close, 542
  IsLoading, 543
  IsStoring, 543
  load, 542
  m_pDocument, 541
  Read, 542
  ReadString, 543
  store, 542
  Write, 543
  WriteString, 543
CBitmap

CreateBitmap, 482; 486
CreateBitmapInderect, 482
CreateCompatibleBitmap, 436; 482; 486
CreateDiscardableBitmap, 482
GetBitmap, 484
GetBitmapBits, 482
GetBitmapDimension, 467; 482; 483
HBITMAP, 467
LoadBitmap, 482
LoadMappedBitmap, 482
LoadOEMBitmap, 482
SetBitmapBits, 482; 486
SetBitmapDimension, 467; 482; 483
CBitmapButton
AutoLoad, 665; 666
LoadBitmaps, 666
SizeToContent, 666
CBN_CLOSEUP, 650
CBN_EDITUPDATE, 649
CBN_SELENDCANCEL, 650
CBN_SELENDOK, 650
CBrush
CBrush, 474
CreateBrushIndirect, 474
CreateDIBPatternBrush, 474
CreateHatchBrush, 474; 508
CreatePatternBrush, 474
CreateSolidBrush, 474; 508
CreateSysColorBrush, 474
HBRUSH, 467
CBS_AUTOHSCROLL, 651
CBS_DISABLENOSCROLL, 651
CBS_DROPDOWN, 649; 651; 652; 675
CBS_HASSTRINGS, 651
CBS_NOINTEGRALHEIGHT, 651
CBS_OEMCONVERT, 651
CBS_OWNERDRAWFIXED, 651
CBS_OWNERDRAWVARIABLE, 651
CBS_SIMPLE, 649; 651; 674
CBS_SORT, 651
CButton
Create, 622; 666
DrawItem, 625
GetButtonStyle, 624
GetCheck, 623
GetState, 623
SetBitmap, 624
SetButtonStyle, 624
SetCheck, 623
SetCursor, 624
SetIcon, 624
SetState, 623
CCdmUI
SetText, 265
CCheckListBox
Create, 667
DrawItem, 669
Enable, 668
GetCheck, 667
GetCheckStyle, 668
IsEnabled, 668
MeasureItem, 669
OnGetCheckPosition, 668
SetCheck, 668
SetCheckStyle, 668
CClientDC, 234
CCmdTarget
BeginWaitCursor, 131; 151
EndWaitCursor, 131; 151
OnCmdMsg, 151
RestoreWaitCursor, 131; 151
CCmdUI, 154; 244
ContinueRouting, 155
Enable, 155; 244; 258; 265
m_nID, 154
m_nIndex, 154
m_pMenu, 155
m_pOther, 155
m_pSubMenu, 155
SetCheck, 155; 231; 244
SetRadio, 155; 244
SetText, 155; 258
CColorDialog, 377
CColorDialog, 391
DoModal, 391
GetColor, 391
GetSavedCustomColors, 392
m_cc, 390
OnColorOK, 392
SetCurrentColor, 392
CComboBox
AddString, 645; 654
Create, 650
DeleteString, 654
DrawItem, 654

FindString, 654
FindStringExact, 654
GetDroppedControlRect, 652
GetDroppedState, 652
GetDroppedWidth, 652
GetExtendedUI, 653
GetLBText, 650; 653
GetLBTextLen, 653
GetText, 651
InsertString, 645; 654
MeasureItem, 654
ResetContent, 654
SelectString, 654
SetDroppedWidth, 652
SetEditSel, 653
SetExtendedUI, 653
ShowDropDown, 652
CCommandLineInfo, 128
CCommonDialog, 377
CControlBar, 127; 239
CalcFixedLayout, 240
EnableDocking, 242
GetBarStyle, 240
GetCount, 240
GetDockingFrame, 241
IsFloating, 242
m_bAutoDelete, 240
SetBarStyle, 240
CCreateContex, 187
CCriticalSection, 926
Lock, 926
Unlock, 926
CCS_BOTTOM, 729
CCS_NOPARENTALIGN, 730
CCS_NORESIZE, 730
CCS_TOP, 730
CCtrlView, 562
m_dwDefaultStyle, 562
m_strClass, 562
CDatabase
m_hdbc, 961
Open, 962
OpenEx, 962
Close, 963
IsOpen, 963
GetDatabaseName, 963
CanUpdate, 963
CanTransact, 963
BeginTrans, 964
CommitTrans, 964
Rollback, 965
Cancel, 965
ExecuteSQL, 965
OnSetOptions, 965
CDataExchange
Fail, 371
m_bSaveAndValidate, 370
m_pDlgWnd, 371
PrepareCtrl, 371
PrepareEditCtrl, 371
CDC
AddMetaFileComment, 462
AngleArc, 447; 450
Arc, 447; 450; 502
ArcTo, 447; 450
BeginPath, 450; 504
BitBlt, 453
Chord, 449; 450; 469
CloseFigure, 450
CreateCompatibleDC, 432
CreateDC, 432
DeleteTempMap, 434
DPtoHIMETRIC, 446
DPtoLP, 446
Draw3dRect, 447
DrawEdge, 447
DrawEscape, 462
DrawFocusRect, 449
DrawFrameControl, 447
DrawIcon, 447
DrawState, 447; 448; 487
DrawText, 233; 457
Ellipse, 449; 450; 469; 501
EndDoc, 462
EndPage, 462
EndPath, 450; 504
EnumObjects, 435
Escape, 462
ExcludeClipRect, 446
ExcludeUpdateRgn, 446
ExtFloodFill, 455; 503
ExtTextOut, 440; 450; 456
FillPath, 450; 504
FillRect, 447; 507
FillRgn, 446
FillSolidRect, 447

FloodFill, 455
FrameRect, 447
FrameRgn, 446
FromHandle, 433
GetBrushOrg, 435
GetCharABCWidth, 462
GetCharWidth, 462
GetClipBox, 447
GetCurrentBitmap, 434
GetCurrentBrush, 434
GetCurrentFont, 434
GetCurrentPalette, 434
GetCurrentPen, 434
GetDeviceCaps, 433; 434
GetFontData, 462
GetGlyphOutline, 462
GetHalftoneBrush, 437
GetKerningPairs, 462
GetOutlineTextMetrics, 462
GetOutputCharWidth, 462
GetOutputTextExtent, 456
GetOutputTextMetrics, 459
GetPixel, 452
GetSafeHdc, 434
GetTextAllign, 456
GetTextExtent, 456
GetTextFace, 456
GetTextMetrics, 459; 505
HIMETRICtoDP, 446
HIMETRICtoLP, 446
IntersectClipRect, 446
InvertRect, 447
InvertRgn, 446
LineTo, 447; 450; 469; 502
LPtoDP, 446; 502
LPToHIMETRIC, 446
m_hAttribDC, 431
m_hDC, 431
MoveTo, 450; 502
PaintRgn, 446; 503
PatBlt, 453
Pie, 449; 450
PlayMetaFile, 462
PlgBlt, 452
PolyBezier, 447; 450; 504
PolyBezierTo, 447; 450
PolyDraw, 447; 450
Polygon, 449; 450; 506
Polyline, 447; 450
PolylineTo, 447; 450
PolyPolygon, 449; 450
PolyPolyline, 447; 450
PtVisible, 447
Rectangle, 449; 450; 469
RestoreDC, 434
RoundRect, 449; 450; 469; 501
SaveDC, 434
ScrollDC, 455
SelectClipPath, 452
SelectClipRgn, 436; 446; 452; 502
SelectObject, 435; 501
SelectPalette, 437
SelectStockObject, 436; 466
SetAttribDC, 431
SetBkColor, 439; 505
SetBkMode, 437; 505
SetBrushOrg, 435
SetMapMode, 441; 500
SetMiterLimit, 471
SetOutputDC, 431
SetPixel, 452
SetPolyFillMode, 437
SetROP2, 438
SetStretchBltMode, 439
SetTextAlign, 456; 457; 459; 505
SetTextColor, 439
SetTextJustification, 456
SetViewportExt, 444; 500
SetViewportOrg, 442; 500
SetWindowExt, 445; 500
SetWindowOrg, 442
StartDoc, 462
StartPage, 462
StokePath, 450
StretchBlt, 454
StretchBltMode, 455
StrokeAndFillPath, 450
TabbedTextOut, 456
TextOut, 234; 440; 450; 456; 505
UpdateColors, 437
CDialog, 127; 166; 340
Create, 344
CreateIndirect, 345
DoModal, 341
EndDialog, 346
GetDefID, 348

GotoDlgCtrl, 347
InitModalIndirect, 344
MapDialogRect, 347
NextDlgCtrl, 347
OnCancel, 346
OnInitDialog, 345
OnOK, 346
OnSetFont, 346
PrevDlgCtrl, 347
SetDefID, 348
CDialogBar, 127; 239; 265
Create, 266
CDocManager
m_templateList, 526
CDocTemplate
AddDocument, 517
CDocTemplate, 517
CloseAllDocuments, 519
CreateNewDocument, 520
CreateNewFrame, 520
docName, 519
fileNewName, 519
filterExt, 519
filterNewName, 519
GeNextDoc, 522
GetDocString, 518
GetFirstDocPosition, 522
InitialUpdateFrame, 521
LoadTemplate, 517
MatchDocType, 520
OpenDocumentFile, 522
regFileTypeId, 519
regFileTypeName, 519
RemoveDocument, 517
SaveAllModified, 519
SetDefaultTitle, 522
windowTitle, 518
CDocument
AddView, 548
CanCloseFrame, 550
CDocument, 548
DeleteContents, 550
GetDocTemplate, 548
GetFile, 552
GetFirstViewPosition, 549
GetNextView, 549
GetPathName, 548
GetTitle, 548
IsModified, 548
OnChangedViewList, 551
OnCloseDocument, 551
OnNewDocument, 551
OnOpenDocument, 552
OnSaveDocument, 552
PreCloseFrame, 552
ReleaseFile, 552
RemoveView, 549
SetModifiedFlag, 548
SetPathName, 548
SetTitle, 548
UpdateAllViews, 549
CDragListBox
BeginDrag, 679
CancelDrag, 679
Dragging, 680
DrawInsert, 679
DrawSingle, 679
Dropped, 680
ItemFromPt, 679
m_nLast, 679
CEdit
CanUndo, 631
CharFromPos, 634
Clear, 631
Copy, 631
Create, 626
Cut, 631
EmptyUndoBuffer, 631
GetFirstVisibleLine, 629
GetLimitText, 633
GetLine, 630
GetLineCount, 630
GetMargins, 633
GetModify, 632
GetPasswordChar, 633
GetRect, 632
GetSel, 628
LimitText, 633
LineFromChar, 630
LineIndex, 630; 631
LineLength, 630
LineScroll, 629
Paste, 631
PosFromChar, 634
ReplaceSel, 629
SetHandle, 632

SetLimitText, 633
SetMargins, 633
SetModify, 632
SetPasswordChar, 633
SetReadOnly, 632
SetRect, 632
SetRectNP, 632
SetSel, 629
Undo, 631
CEditView
CEditView, 563
dwStyleDefault, 563
FindText, 564
GetBufferLength, 563
GetSelectedText, 563
LockBuffer, 563
OnFindText, 564
OnReplaceAll, 564
OnReplaceSel, 564
OnTextNotFound, 565
SerializeRaw, 564
SetTabStops, 563
UnlockBuffer, 564
CEvent, 928
PulseEvent, 928
ResetEvent, 929
SetEvent, 928
Unlock, 927
CF_TEXT, 631
CFE_AUTOCOLOR, 796
CFE_BOLD, 796
CFE_ITALIC, 796
CFE_PROTECTED, 796
CFE_STRIKEOUT, 796
CFE_UNDERLINE, 796
CFileDialog, 377; 382
DoModal, 383
GetFileExt, 383
GetFileName, 383
GetFileTitle, 383
GetNextPathName, 383
GetPathName, 383
GetReadOnlyPref, 383
GetStartPosition, 383
m_ofn, 378
OnFileNameChange, 384
OnFileNameOK, 384
OnFolderChange, 384
OnInitDone, 384
OnLBSelChangedNotify, 384
OnShareViolation, 384
OnTypeChange, 384
CFindReplace
FindNext, 404
CFindReplaceDialog, 377; 404
Create, 404
GetFindString, 404
GetNotifier, 404
GetReplaceString, 404
IsTerminating, 405
m_fr, 402
MatchCase, 405
MatchWholeWord, 405
ReplaceAll, 405
ReplaceCurrent, 405
SearchDown, 405
CFM_BOLD, 796
CFM_COLOR, 796
CFM_FACE, 796
CFM_ITALIC, 796
CFM_OFFSET, 796
CFM_PROTECTED, 796
CFM_SIZE, 796
CFM_STRIKEOUT, 796
CFM_UNDERLINE, 796
CFont
CreateFont, 475; 479; 508
CreateFontIndirect, 475
CreatePointFont, 475; 480
CreatePointFontIndirect, 475; 480
HFONT, 467
CFontDialog, 377; 398
DoModal, 398
GetColor, 399
GetCurrentFont, 399
GetFaceName, 399
GetSize, 399
GetStyleName, 399
GetWeight, 399
IsItalic, 399
IsStrikeOut, 399
IsUnderLine, 399
m_cf, 395
CFormView, 127
CFrameWnd, 166
Create, 170; 187

DockControlBar, 242
EnableDocking, 242
FloatControlBar, 243
GetDockState, 170
LoadAccelTable, 170; 331
LoadBarState, 170; 242
LoadFrame, 170; 186
m_bAutoMenuEnable, 170
OnEndPrintPreview, 603
OnSetPreviewMode, 603
RecalcLayout, 243
SaveBarState, 170; 242
SetDockState, 170
ShowControlBar, 170; 242
CGdiObject
Attach, 466; 482
CreateStockObject, 466; 473
DeleteObject, 464; 504
FromHandle, 464
HGDIOBJ, 467
GetObject, 467; 484
GetSafeHandle, 463
m_hObject, 463
CHARFORMAT, 796
CHeaderCtrl
Create, 729
DeleteItem, 731
DrawItem, 736
GetItem, 732
GetItemCount, 732
InsertItem, 731
SetItem, 732
CHINESEBIG5_CHARSET, 462; 477
CHOOSECOLOR, 390
CHOOSEFONT, 395
CHotKeyCtrl
Create, 689
GetHotKey, 690; 692
SetHotKey, 690
SetRules, 690
CHttpServer
CallFunction, 955
CHttpServer, 954
InitInstance, 954
CHttpServerContext, 955
CImageList
Add, 722
Attach, 721; 723
BeginDrag, 724
Create, 721; 722
Detach, 721; 723
DragEnter, 724
DragLeave, 724
DragMove, 724
DragShowNolock, 724
Draw, 723
EndDrag, 724
ExtractIcon, 722
GetImageInfo, 727
GetSafeHandle, 721
m_hImageList, 721
Read, 722
Remove, 722
Replace, 722
SetBkColor, 723
SetDragCursorImage, 724
Write, 722
CLIP_CHARACTER_PRECIS, 478
CLIP_DEFAULT_PRECIS, 478
CLIP_EMBEDDED, 478
CLIP_LH_ANGLES, 478
CLIP_MASK, 478
CLIP_STROKE_PRECIS, 478
CLIP_TT_ALWAYS, 478
CListBox
AddString, 639; 640; 645
CompareItem, 639; 645
Create, 635
DeleteString, 640
Dir, 640
DrawItem, 643; 646
FindString, 641
GetCount, 641
GetCurSel, 642
GetHorizontalExtent, 637
GetItemData, 643
GetItemDataPtr, 643
GetItemHeight, 637
GetItemRect, 637
GetLocale, 638
GetSel, 641
GetSelItems, 642
GetText, 642
GetTextLen, 642
GetTopIndex, 643
InitStorage, 640

InsertString, 639; 640; 645
ItemFromPoint, 643
MeasureItem, 645; 646
ResetContent, 641
SelectString, 641
SetColumnWidth, 635; 637
SetCurSel, 634; 642
SetHorizontalExtent, 637
SetItemData, 643
SetItemDataPtr, 643
SetItemHeight, 637
SetItemRange, 642
SetLocale, 638
SetSel, 642
SetTabStops, 638
SetTopIndex, 643
CListCtrl
Arrange, 745
Create, 738
CreateDragImage, 755
DeleteColumn, 743
DeleteItem, 741
DrawItem, 755
EditLabel, 753
EnsureVisible, 745
FindItem, 752
GetBkColor, 744
GetCallbackMask, 749
GetColumn, 743
GetColumnWidth, 743
GetCountPerPage, 744
GetEditControl, 753
GetImageList, 747
GetItem, 750
GetItemCount, 749
GetItemData, 750
GetItemPosition, 750
GetItemState, 750
GetItemText, 750
GetNextItem, 751
GetOrigin, 745
GetSelectedCount, 750
GetStringWidth, 741
GetTextBkColor, 744
GetTextColor, 744
GetTopIndex, 744
GetViewRect, 744
InsertColumn, 741
InsertItem, 740
RedrawItems, 745
Scroll, 745
SetBkColor, 744
SetCallbackMask, 749
SetColumn, 743
SetColumnWidth, 743
SetImageList, 747
SetItem, 750
SetItemCount, 749
SetItemData, 750
SetItemPosition, 751
SetItemState, 750
SetItemText, 750
SetTextBkColor, 744
SetTextColor, 744
SortItems, 752
Update, 745
CListView, 568
GetListCtrl, 568
CloseHandle, 74; 924
CLR_NONE, 723; 744
CMDIChildWnd, 166
Create, 206
GetMDIFrame, 207
m_hMenuShared, 206
MDIActivate, 207
MDIDestroy, 207
MDIMaximize, 207
MDIRestore, 207
CMDIFrameWnd, 166
CreateClient, 205
GetWindowMenuPopup, 206
MDIActivate, 204
MDICascade, 205
MDIGetActive, 204
MDIIconArrange, 204
MDIMaximize, 204
MDINext, 204
MDIRestore, 204
MDISetMenu, 205
MDITile, 205
CMenu
AppendMenu, 298; 645
CheckMenuItem, 301
CheckMenuRadioItem, 301
CMenu, 296
CreateMenu, 296

CreatePopupMenu, 296
DeleteMenu, 300
DestroyMenu, 298
DrawItem, 305
EnableMenuItem, 300
GetMenuContextHelpId, 306
GetMenuItemCount, 302
GetMenuItemID, 302
GetMenuState, 302
GetMenuString, 302
GetSubMenu, 302
InsertMenu, 299; 645
LoadMenu, 296; 297
LoadMenuIndirect, 297
m_hMenu, 296
MeasureItem, 305
ModifyMenu, 300; 645
RemoveMenu, 300
SetMenuContextHelpId, 306
SetMenuItemBitmaps, 301
TrackPopupMenu, 304
CMultiDocTemplate
CMultiDocTemplate, 523
CMultiLock
CMultiLock, 932
IsLocked, 933
Lock, 932
Unlock, 933
CMutex
CMutex, 926
Unlock, 926
CObject, 78
COleResizerBar, 239
COLORONCOLOR, 439
ConstructStream, 955
CPageSetupDialog, 377
CPageSetupDialog, 417
CreatePrinterDC, 418
DoModal, 417
GetDeviceName, 418
GetDevMode, 418
GetDriverName, 418
GetMargins, 418
GetPaperSize, 418
GetPortName, 418
OnDrawPage, 418
PreDrawPage, 419
CPaintDC, 232
CPalette
HPALETTE, 467
CPen, 468
CreatePen, 473; 508; 509
CreatePenIndirect, 473
HPEN, 467
CPoint, 233
CPrintDialog, 377; 412
CreatePrinterDC, 413
DoModal, 412
GetCopies, 413
GetDefaults, 413
GetDeviceName, 413
GetDevMode, 413
GetDriverName, 413
GetFromPage, 413
GetPortName, 413
GetPrinterDC, 413
GetToPage, 413
m_pd, 405; 414
PrintAll, 413
PrintCollate, 413
PrintRange, 414
PrintSelection, 414
CProgressCtrl
Create, 698
OffsetPos, 699
SetPos, 699
SetRange, 699
SetStep, 699
StepIt, 699
CPropertyPage, 127
CancelToClose, 834
Construct, 834
CPropertyPage, 834
OnApply, 836
OnCancel, 836
OnKillActive, 835
OnOK, 835
OnQueryCancel, 836
OnReset, 836
OnSetActive, 835; 846
OnWizardBack, 846
OnWizardFinish, 847
OnWizardNext, 847
QuerySiblings, 835
SetModified, 820; 834
CPropertySheet, 127; 815

AddPage, 825; 831
Construct, 816
Create, 819; 828; 831
DoModal, 818; 828; 831; 846
EndDialog, 828
GetActiveIndex, 826
GetActivePage, 826
GetPage, 826
GetPageCount, 826
GetPageIndex, 826
GetTabControl, 827
m_psh, 821
OnInitDialog, 829; 830
PressButton, 827
RemovePage, 826
SetActivePage, 826; 827; 847
SetFinishText, 846
SetTitle, 827
SetWizardButtons, 846
WizardMode, 846
CREATE_SUSPENDED, 870
CreateDIBitmap, 487
CreateFile, 74
CreateProcess, 900
CreatePropertySheetPage, 825
CREATESTRUCT, 188
CreateThread, 900
CRecordset
m_hstmt, 966
m_nFields, 966
m_nParams, 966
m_pDatabase, 966
m_strFilter, 966
m_strSort, 967
Open, 967
Close, 971
CanAppend, 971
CanBookmark, 971
CanRestart, 972
CanScroll, 972
CanTransact, 972
CanUpdate, 972
GetODBCFieldCount, 972
GetRecordCount, 972
GetTableName, 972
GetSQL, 972
IsOpen, 973
IsBOF, 973
IsEOF, 973
IsDeleted, 973
AddNew, 973
Delete, 974
Edit, 974
Update, 974
CancelUpdate, 975
Move, 975
MoveFirst, 976
MoveLast, 976
MoveNext, 976
MovePrev, 976
SetAbsolutePosition, 976
Cancel, 977
GetRowsetSize, 977
GetRowsFetched, 977
GetRowStatus, 977
IsFieldNull, 977
IsFieldNullable, 977
RefreshRowset, 978
Requery, 978
SetFieldNull, 978
SetLockingMode, 978
SetRowsetCursorPosition, 979
Check, 979
CheckRowsetError, 979
DoFieldExchange, 979
DoBulkFieldExchange, 980
GetDefaultConnect, 980
GetDefaultSQL, 980
OnSetOptions, 980
SetRowsetSize, 981
CRect, 233
CRgn
CombineRgn, 491; 503
CopyRgn, 492
CreateEllipticRgn, 490
CreateEllipticRgnIndirect, 490; 503
CreateFromPath, 492
CreatePolygonRgn, 490
CreatePolyPolygonRgn, 490
CreateRectRgn, 489
CreateRectRgnIndirect, 490; 502
CreateRoundRectRgn, 491
EqualRgn, 492
operator HRGN, 467
CRichEditCtrl, 782
CanPaste, 792

Clear, 790; 791
Copy, 791
Create, 782
Cut, 791
DisplayBand, 792; 808
EmptyUndoBuffer, 790
FindText, 790
FormatRange, 792
GetCharPos, 789
GetDefaultCharFormat, 795
GetEventMask, 793
GetFirstVisibleLine, 787
GetIRichEditOle, 800
GetLimitText, 787
GetLine, 787
GetLineCount, 787
GetModify, 784
GetParaFormat, 798; 799
GetRect, 785
GetSel, 789
GetSelectionCharFormat, 797
GetSelectionType, 790
GetSelText, 789
GetTextLength, 787
HideSelection, 791
LimitText, 787
LineFromChar, 788
LineIndex, 788
LineLength, 788
LineScroll, 788
Paste, 791
PasteSpecial, 791
ReplaceSel, 791
RequestResize, 786
SetBackgroundColor, 784
SetDefaultCharFormat, 797
SetEventMask, 793; 797
SetModify, 784
SetOLECallback, 800
SetOptions, 783; 791
SetParaFormat, 799
SetReadOnly, 784
SetRect, 785
SetSel, 789; 791
SetSelectionCharFormat, 797
SetTargetDevice, 793
SetWordCharFormat, 797
Undo, 790; 791
CRichEditDoc
CreateClientItem, 810
GetStreamFormat, 810
GetView, 810
CRichEditView, 804
AdjustDialogPosition, 807
CanPaste, 807
DoPaste, 807
FindText, 807
FindTextSimple, 807
GetCharFormatSelection, 805
GetDocument, 804
GetInPlaceActiveItem, 806
GetMargins, 806
GetPageRect, 806; 808
GetPaperSize, 805
GetParaFormatSelection, 805
GetPrintRect, 806; 808
GetPrintWidth, 806
GetRichEditCtrl, 804
GetSelectedItem, 806
GetTextLength, 805
InsertFileAsObject, 808
InsertItem, 808
IsRichEditFormat, 807
IsSelected, 809
m_nBulletIndent, 806
m_nWordWrap, 806
OnCharEffect, 805
OnFindNext, 809
OnInitialUpdate, 808
OnParaAlign, 805
OnReplaceAll, 809
OnReplaceSel, 809
OnTextNotFound, 809
PrintInsideRect, 808
PrintPage, 808
SetCharFormat, 805
SetMargins, 806
SetPaperSize, 805
SetParaFormat, 805
WrapChanged, 806
CRITICAL_SECTION, 859
CRuntimeClass, 79
CScrollBar
Create, 657
EnableScrollBar, 660
GetScrollInfo, 660

GetScrollLimit, 662
GetScrollPos, 659
GetScrollRange, 659
SetScrollBar, 656
SetScrollInfo, 660
SetScrollPos, 659
SetScrollRange, 659
ShowScrollBar, 660
CScrollView, 569
GetDeviceScrollPosition, 571
GetDeviceScrollSizes, 570
GetScrollPosition, 571
GetTotalSize, 570
ResizeParentToFit, 573
ScrollToPosition, 571
SetScaleToFitSize, 571
SetScrollSizes, 569
CSemaphore, 929
Unlock, 929
CSingleDocTemplate, 521
CSingleLock, 931
IsLocked, 931
Lock, 931
Unlock, 931
CSize, 233
CSliderCtrl
ClearSel, 706
ClearTics, 707
Create, 705
GetChannelRect, 706
GetLineSize, 706
GetNumTics, 706
GetPageSize, 706
GetPos, 707
GetRange, 706
GetRangeMax, 706
GetRangeMin, 706
GetSelection, 706
GetThumbRect, 706
GetTic, 707
GetTicArray, 707
SetLineSize, 706
SetPageSize, 706
SetPos, 707
SetRange, 706
SetRangeMax, 706
SetRangeMin, 706
SetSelection, 706
SetTicFreq, 707
VerifyPos, 707
CSpinButtonCtrl
Create, 711
GetBase, 713
GetBuddy, 711
GetPos, 711
GetRange, 712
SetBase, 713
SetBuddy, 711
SetPos, 711
SetRange, 712
CSplitterWnd, 576
ActivateNext, 581
CanActivateNext, 581
Create, 576
CreateScrollBarCtrl, 581
CreateStatic, 577
CreateView, 577
DeleteColumn, 581
DeleteRow, 581
DeleteView, 580
DoScroll, 581
GetActivePane, 581
GetColumnCount, 579
GetColumnInfo, 579
GetRowCount, 579
GetRowInfo, 579
GetScrollStyle, 580
IdFromEowCol, 579
OnDrawSplitter, 580
RecalcLayout, 579
SetActivePane, 581
SetColumnInfo, 579
SetRowInfo, 579
SetScrollStyle, 580
SplitRow, 580
CSrollBar
SetScrollRange, 662
CStatic, 612
Create, 612
GetBitmap, 615
GetCursor, 616
GetEnhMetaFile, 617
GetIcon, 616
SetBitmap, 615
SetCursor, 617
SetEnhMetaFile, 617

SetIcon, 616
CStatusBar, 239; 258
Create, 258
GetPaneInfo, 260
GetPaneStyle, 260
GetPaneText, 260
GetStatusBarCtrl, 260
SetIndicators, 259
SetPaneInfo, 259
SetPaneStyle, 260
SetPaneText, 260
CStatusBarCtrl, 261
DrawItem, 263
GetBorders, 262
GetParts, 262
GetRect, 263
GetText, 261
GetTextLenght, 261
SetMinHeight, 263
SetParts, 262
SetSimple, 263
SetText, 261
CString
LoadString, 333
CSyncObject, 924
HANDLE, 923
Lock, 924
m_hObject, 923
Unlock, 924
CTabCtrl, 839
AdjustRect, 841
Create, 839
DeleteAllItems, 841
DeleteItem, 841
DrawItem, 841
GetCurFocus, 845
GetCurSel, 844
GetImageList, 842
GetItem, 842
GetItemCount, 842
GetItemRect, 843
GetRowCount, 844
GetTooltips, 844
InsertItem, 839; 840
RemoveImage, 841
SetCurSel, 844
SetImageList, 842
SetItem, 842
SetItemSize, 844
SetPadding, 844
SetTooltips, 845
CToolBar, 127; 239; 244
CommandToIndex, 247
Create, 244
GetButtonInfo, 248
GetButtonStyle, 247
GetButtonText, 248
GetItemID, 247
GetItemRect, 247
GetToolBarCtrl, 249
LoadBitmap, 246
LoadToolBar, 245; 246
SetBitmap, 247
SetButtonInfo, 248
SetButtons, 246
SetButtonStyle, 247
SetButtonText, 248
SetHeight, 247
SetSizes, 246
CToolBarCtrl, 249
AddBitmap, 252
AddButtons, 253
AddString, 252
AddStrings, 252
AutoSize, 253
Customize, 252
DeleteButton, 253
GetBitmapFlags, 251
GetButton, 251
GetButtonCount, 251
GetRows, 251
InsertButton, 253
RestoreState, 252
SaveState, 252
SetOwner, 251
SetRows, 251
CToolTip
Create, 717
CToolTipCtrl
Activate, 718
AddTool, 717
DelTool, 718
GetText, 719
GetToolInfo, 719
HitTest, 719
RelayEvent, 718

SetDelayTime, 719
SetToolInfo, 719
UpdateTipText, 719
CTreeCtrl
Create, 764
CreateDragImage, 773
DeleteAllItems, 765
DeleteItem, 769
EditLabel, 773
EnsureVisible, 771
Expand, 769
GetChildItem, 769
GetCount, 765
GetEditControl, 773
GetFirstVisibleItem, 771
GetImageList, 765
GetIndent, 765
GetItemImage, 769
GetItemRect, 769
GetItemState, 769
GetNextItem, 770
GetNextSiblingItem, 771
GetNextVisibleItem, 771
GetParentItem, 771
GetPrevSiblingItem, 771
GetPrevVisibleItem, 771
GetSelectedItem, 771
GetVisibleCount, 765
InsertItem, 766
ItemHasChildren, 769
Select, 770
SelectDropTarget, 770
SelectItem, 770
SetImageList, 765
SetIndent, 765
SetItem, 769
SetItemImage, 769
SetItemState, 769
SortChildren, 772
SortChildrenCB, 772
CTreeView, 568
GetTreeCtrl, 568
CView, 166; 558
DoPreparePrinting, 590
DoPrintPreview, 602
GetDocument, 558
OnActivateFrame, 560
OnActivateView, 560
OnBeginPrinting, 592
OnDraw, 425; 561
OnEndPrinting, 593
OnFilePrint, 594
OnInitialUpdate, 558
OnPrepareDC, 425; 593
OnPreparePrinting, 589
OnPrint, 594
OnScroll, 560
OnUpdate, 559
CWaitCursor, 131
CWinApp, 125; 126
AddDocTemplate, 517; 524
CloseAllDocuments, 526
CreatePrinterDC, 588
DoMessageBox, 131
DoWaitCursor, 131
Enable3dControls, 127
Enable3dControlsStatic, 127
EnableShellOpen, 127
ExitInstance, 130
GetFirstDocTemplatePosition, 525
GetNextDocTemplate, 525
GetPrinterDeviceDefaults, 587
GetProfileBinary, 129
GetProfileInt, 128
GetProfileString, 129
HideApplication, 131
InitInstance, 130
LoadCursor, 128
LoadIcon, 128
LoadOEMCursor, 128
LoadOEMIcon, 128
LoadStandardCursor, 128
LoadStandardIcon, 128
LoadStdProfileSettings, 127
m_hInstance, 125
m_hPrevInstance, 125
m_lpCmdLine, 125
m_nCmdShow, 125
m_pDocManager, 526
m_pszAppName, 125
m_pszExeName, 126
m_pszHelpFilePath, 126
m_pszProfileName, 126
m_pszRegistryKey, 126
OnFileNew, 526
OnFileOpen, 526

OnFilePrintSetup, 585
OnIdle, 130
OpenDocumentFile, 525
ParseCommandLine, 128
ProcessShellCommand, 128
RegisterShellFileTypes, 127
Run, 130
SaveAllModified, 526
SelectPrinter, 418
SetDialogBkColor, 127
SetRegistryKey, 127
WriteProfileBinary, 129
WriteProfileInt, 129
WriteProfileString, 129
CWinThread, 125; 877; 895
CreateThread, 868; 877; 895; 905
ExitInstance, 878
GetThreadPriority, 876
InitInstance, 878
m_bAutoDelete, 876; 897
m_hThread, 876
m_nThreadID, 876
m_pMainWnd, 126; 170
OnIdle, 879
PreTranslateMessage, 880
ProcessWndProcException, 880
PumpMessage, 142
ResumeThread, 877
Run, 879
SetThreadPriority, 876
SuspendThread, 877; 899; 910
CWnd, 143; 166
ArrangeIconicWindows, 167
Attach, 167
BeginPaint, 168; 232
BringWindowToTop, 168; 224
CenterWindow, 168
Create, 167
CreateEx, 167; 187
Default, 156; 656
DestroyWindow, 819
Detach, 167
DoDataExchange, 369
DrawMenuBar, 301
EnableToolTips, 714; 716
EnableWindow, 167; 201
EndPaint, 168; 232
FindWindow, 168; 223
FromHandle, 167
FromHandlePermanent, 167
GetActiveWnd, 167
GetCapture, 167
GetClientRect, 168; 233
GetCurrentMessage, 156; 655
GetDC, 168
GetDlgItem, 347
GetExStyle, 167; 197
GetFocus, 167
GetFont, 168
GetForegroundWindow, 167; 224
GetIcon, 167
GetParent, 207
GetSafeWnd, 167; 196
GetStyle, 167; 197; 632
GetWindowDC, 168
GetWindowPlacement, 167
GetWindowRect, 168
GetWindowText, 168; 650
GetWindowTextLenght, 168
Invalidate, 168; 231
IsIconic, 167; 224
IsWindowEnable, 167; 201
IsZoomed, 167
m_hWnd, 167
ModifyStyle, 167; 197
ModifyStyleEx, 197
MoveWindow, 167; 194; 786
OnDestroy, 898
OnHScroll, 655
OnInitMenu, 309
OnInitMenuPopup, 310
OnMenuSelect, 310
OnVScroll, 656
PostMessage, 156; 898
PostNcDestroy, 898
PreCreateWindow, 167
PreSubclassWindow, 167
PreTranslateMessage, 156
Print, 168
PrintClient, 168
ReleaseDC, 168
SendMessage, 156
SendNotifyMessage, 156
SetActiveWnd, 167
SetCapture, 167
SetFocus, 167

SetFont, 168
SetForegroundWindow, 167; 224
SetIcon, 167
SetScrollPos, 656
SetWindowPlacement, 167
SetWindowPos, 167; 189; 786
SetWindowText, 168; 629; 632; 633; 791
ShowWindow, 168
SubclassWindow, 167
UpdateData, 369
UpdateWindow, 168

D

deadlock, 861
DEFAULT_CHARSET, 462; 477
DEFAULT_PALETTE, 437
DEFAULT_PARSE_COMMAND, 957
DEFAULT_PITCH, 479
DEFAULT_QUALITY, 479
DEVICE_DEFAULT_FONT, 436
DEVMODE, 410
DEVNAMES, 412
DispatchMessage, 62; 130; 156; 879
DKGRAY_BRUSH, 436; 437
DL_COPYCURSOR, 680
DL_MOVECURSOR, 680
DL_STOPCURSOR, 680
DLGITEMTEMPLATE, 342
DLGTEMPLATE, 341
DRAFT_QUALITY, 479
DRAWITEMSTRUCT, 644
DSS_DEFAULT, 449
DSS_DISABLED, 449
DSS_MONO, 449
DSS_NORMAL, 449
DSS_UNION, 449
DST_BITMAP, 448
DST_COMPLEX, 448
DST_ICON, 448
DST_PREFIXTEXT, 449
DST_TEXT, 449
DSTINVERT, 453
DT_BOTTOM, 458
DT_CALCRECT, 458
DT_CENTER, 458
DT_EXPANDTABS, 458
DT_EXTERNALLEADING, 458
DT_LEFT, 458
DT_NOCLIP, 458
DT_NOPREFIX, 458
DT_RIGHT, 458
DT_SINGLELINE, 458
DT_TABSTOP, 458
DT_TOP, 458
DT_VCENTER, 458
DT_WORDBREAK, 458; 459
DuplicateHandle, 901; 923

E

EASTEUROPE_CHARSET, 462; 477
ECO_AUTOHSCROLL, 783
ECO_AUTOVSCROLL, 783
ECO_AUTOWORDSELECTION, 783
ECO_NOHIDESEL, 783
ECO_READONLY, 783
ECO_SAVESEL, 784
ECO_VERTICAL, 784
ECO_WANTRETURN, 783
ECOOP_AND, 783
ECOOP_OR, 783
ECOOP_SET, 783
ECOOP_XOR, 783
EN_CHANGE, 793
EN_CORRECTTEXT, 794
EN_DROPFILES, 794
EN_ERRSPACE, 793
EN_HSCROLL, 793
EN_IMECHANGE, 794
EN_KILLFOCUS, 793
EN_MAXTEXT, 793
EN_MSGFILTER, 794
EN_OLEOPFAILED, 794
EN_PROTECTED, 794; 797
EN_REQUESTRESIZE, 794
EN_SAVECLIPBOARD, 794
EN_SELCHANGE, 794
EN_SETFOCUS, 793
EN_STOPNOUNDO, 794
EN_UPDATE, 794
EN_VSCROLL, 794
END_PARSE_MAP, 956
ENM_CHANGE, 795
ENM_CORRECTTEXT, 795

ENM_DROPFILES, 795
ENM_IMECHANGE, 795
ENM_KEYEVENTS, 795
ENM_MOUSEEVENTS, 795
ENM_NONE, 795
ENM_PROTECTED, 795
ENM_REQUESTRESIZE, 795
ENM_SCROLL, 795
ENM_SELCHANGE, 795
ENM_UPDATE, 795
ES_AUTOHSCROLL, 626; 627; 793
ES_AUTOVSCROLL, 626; 627; 793
ES_CENTER, 627
ES_DISABLENOSCROLL, 782
ES_LEFT, 627
ES_LOWERCASE, 627
ES_MULTILINE, 627
ES_NOHIDESEL, 627
ES_OEMCONVERT, 627
ES_PASSWORD, 628
ES_READONLY, 628
ES_RIGHT, 628
ES_SAVESEL, 782
ES_SELECTIONBAR, 782
ES_SUNKEN, 782
ES_UPPERCASE, 628
ES_WANTRETURN, 628
ESB_DISABLE_BOTH, 660
ESB_DISABLE_LTUP, 660
ESB_DISABLE_RTDN, 660
ESB_ENABLE_BOTH, 660
ETO_CLIPPED, 457
ETO_OPAQUE, 457
event, 863

F

FF_DECORATIVE, 461; 479
FF_DONTCARE, 461; 479
FF_MODERN, 461; 479
FF_ROMAN, 461; 479
FF_SCRIPT, 461; 479
FF_SWISS, 461; 479
FIFO, 853
FINDREPLACE, 402
FIXED_PITCH, 479
FLOODFILLBORDER, 455
FLOODFILLSURFACE, 455
FT_MATCHCASE, 790
FT_WHOLEWORD, 790
FW_BLACK, 477
FW_BOLD, 477
FW_DEMIBOLD, 477
FW_DONTCARE, 477
FW_EXTRABOLD, 477
FW_EXTRALIGHT, 477
FW_HEAVY, 477
FW_LIGHT, 477
FW_MEDIUM, 477
FW_NORMAL, 477
FW_REGULAR, 477
FW_SEMIBOLD, 477
FW_THIN, 477
FW_ULTRABOLD, 477
FW_ULTRALIGHT, 477

G

GB2312_CHARSET, 462; 477
GetChunkSize, 956
GetClassLong, 195
GetDialogBaseUnits, 636
GetExitCodeThread, 878; 900
GetExtensionVersion, 955
GetFileSize, 74
GetMessage, 61; 142; 156; 852
GetOpenFileName, 74
GetSaveFileName, 74
GetServerVariable, 955
GetSysColor, 796
GetSystemMetrics, 199
GetWindowLong, 196; 743
GM_ADVANCED, 476
GRAY_BRUSH, 436; 437
GREEK_CHARSET, 462; 477

H

HALFTONE, 439
HANGEUL_CHARSET, 462; 477
HD_ITEM, 730
HD_NOTIFY, 733
HDF_BITMAP, 731
HDF_CENTER, 731
HDF_LEFT, 731
HDF_OWNERDRAW, 731

HDF_RIGHT, 731
HDF_STRING, 731
HDI_BITMAP, 730
HDI_FORMAT, 730
HDI_HEIGHT, 730
HDI_LPARAM, 731
HDI_TEXT, 731
HDI_WIDTH, 731
HDN_BEGINTRACK, 733
HDN_DIVIDERDBLCLICK, 733
HDN_ENDTRACK, 733
HDN_ITEMCHANGED, 733
HDN_ITEMCHANGING, 733
HDN_ITEMCLICK, 733
HDN_ITEMDBLCLICK, 733
HDN_TRACK, 733
HDS_BUTTONS, 729
HDS_HIDDEN, 729
HDS_HORZ, 729
HDS_VERT, 729
HEBREW_CHARSET, 462; 477
High, 855
HKCOMB_A, 691
HKCOMB_C, 691
HKCOMB_CA, 691
HKCOMB_NONE, 691
HKCOMB_S, 691
HKCOMB_SA, 691
HKCOMB_SC, 691
HKCOMB_SCA, 691
HOLLOW_BRUSH, 436
HOTKEYF_ALT, 690
HOTKEYF_CONTROL, 690
HOTKEYF_EXT, 690
HOTKEYF_SHIFT, 690
HS_BDIAGONAL, 474
HS_CROSS, 474
HS_DIAGCROSS, 474
HS_FDIAGONAL, 474
HS_HORIZONTAL, 474
HS_VERTICAL, 474

I

ID_APPLY_NOW, 835
Idle, 854
ILD_BLEND25, 723
ILD_BLEND50, 723
ILD_NORMAL, 723
ILD_OVERLAYMASK, 723
ILD_TRANSPARENT, 723
INDEXTOOVERLAYMASK, 723
ITS_EMPTY, 957
ITS_I2, 957
ITS_I4, 957
ITS_PSTR, 957
ITS_R4, 957
ITS_R8, 957
ITS_RAW, 957

J, L

JOHAB_CHARSET, 462; 477
LBS_DISABLENOSCROLL, 636
LBS_EXTENDEDSEL, 635
LBS_HASSTRINGS, 635; 639; 645; 678; 680
LBS_MULTICOLUMN, 635
LBS_MULTIPLESEL, 635; 641; 678
LBS_NOINTEGRALHEIGHT, 635; 646
LBS_NOREDRAW, 635
LBS_NOTIFY, 634; 635
LBS_OWNERDRAWFIXED, 636; 643; 678; 680
LBS_OWNERDRAWVARIABLE, 636; 637; 643; 646; 678; 680
LBS_SORT, 636; 639; 678
LBS_STANDARD, 636
LBS_USETABSTOPS, 636
LBS_WANTKEYBOARDINPUT, 635; 636
LOGFONT, 476; 797
LTGRAY_BRUSH, 436; 437
LV_COLUMN, 742
LV_ITEM, 740
LVA_ALIGNLEFT, 745
LVA_ALIGNTOP, 745
LVA_DEFAULT, 745
LVA_SNAPTOGRID, 745
LVA_SORTASCENDING, 745
LVA_SORTDESCENDING, 745
LVCF_FMT, 742
LVCF_SUBITEM, 742
LVCF_TEXT, 742
LVCF_WIDTH, 742
LVIF_IMAGE, 740

LVIF_PARAM, 740
LVIF_STATE, 740
LVIF_TEXT, 740
LVIS_CUT, 741; 749
LVIS_DROPHILITED, 741; 749
LVIS_FOCUSED, 741; 749
LVIS_OVERLAYMASK, 749
LVIS_SELECTED, 741; 749
LVIS_STATEIMAGEMASK, 749
LVN_BEGINDRAG, 753; 754
LVN_BEGINLABELEDIT, 753
LVN_BEGINRDRAG, 753; 754
LVN_COLUMNCLICK, 753
LVN_DELETEALLITEMS, 753
LVN_DELETEITEM, 753
LVN_ENDLABELEDIT, 738; 753
LVN_GETDISPINFO, 753
LVN_INSERTITEM, 753
LVN_ITEMCHANGED, 754
LVN_ITEMCHANGING, 754
LVN_KEYDOWN, 754
LVN_SETDISPINFO, 754
LVNI_ABOVE, 751
LVNI_ALL, 751
LVNI_BELOW, 751
LVNI_DROPHILITED, 751
LVNI_FOCUSED, 751
LVNI_HIDDEN, 751
LVNI_MARKED, 751
LVNI_SELECTED, 751
LVNI_TOLEFT, 751
LVNI_TORIGHT, 751
LVS_ALIGNLEFT, 738
LVS_ALIGNTOP, 738
LVS_AUTOARRANGE, 738
LVS_EDITLABELS, 738
LVS_ICON, 738
LVS_LIST, 738
LVS_NOCOLUMNHEADER, 738
LVS_NOLABELWRAP, 738
LVS_NOSCROLL, 738
LVS_NOSORTHEADER, 738
LVS_OWNERDRAWFIXED, 738
LVS_REPORT, 738
LVS_SHAREIMAGELISTS, 738
LVS_SINGLESEL, 738
LVS_SMALLICON, 738
LVS_SORTASCENDING, 738
LVS_SORTDESCENDING, 738

M

m_bSendHeaders, 955
m_pECB, 955
m_pStream, 955
MAC_CHARSET, 462; 477
MAXIMUM_WAIT_OBJECTS, 932
MEASUREITEMSTRUCT, 645
MENUITEMTEMPLATE, 297
MENUITEMTEMPLATEHEADER, 297
MERGECOPY, 454
MERGEPAINT, 454
MF_BITMAP, 310
MF_BYCOMMAND, 300
MF_BYPOSITION, 300
MF_CHECKED, 298; 310
MF_DISABLED, 298; 310
MF_ENABLED, 298
MF_GRAYED, 298; 310
MF_MENUBARBREAK, 298
MF_MENUBREAK, 298
MF_MOUSESELECT, 310
MF_OWNERDRAW, 298; 299; 310
MF_POPUP, 298; 310
MF_SEPARATOR, 299; 310
MF_STRING, 299
MF_SYSMENU, 310
MF_UNCHECKED, 298
MINMAXINFO, 199
MM_ANISOTROPIC, 441; 442
MM_HIENGLISH, 441; 442
MM_HIMETRIC, 441; 442
MM_ISOTROPIC, 441; 442
MM_LOENGLISH, 441; 442
MM_LOMETRIC, 441; 442
MM_TEXT, 440; 441; 476
MM_TWIPS, 441; 442
MOD_ALT, 692
MOD_CONTROL, 692
MOD_SHIFT, 692
MSG, 61
MsgForMultipleObjetcs, 925
MsgWaitForMultipleObjects, 861

N

NM_LISTVIEW, 754
NM_TREEVIEW, 773
NM_UPDOWNW, 709
NMHDR, 255
Normal, 854
NOTSRCCOPY, 454
NOTSRCERASE, 454
NULL_BRUSH, 436
NULL_PEN, 436

O

OBM_BTNCORNERS, 483
OBM_BTSIZE, 483
OBM_CHECK, 483
OBM_CHECKBOXES, 483
OBM_CLOSE, 483
OBM_COMBO, 483
OBM_DNARROW, 482
OBM_DNARROWD, 482
OBM_DNARROWI, 482
OBM_LFARROW, 482
OBM_LFARROWD, 482
OBM_LFARROWI, 482
OBM_MNARROW, 483
OBM_OLD_CLOSE, 483
OBM_OLD_DNARROW, 482
OBM_OLD_LFARROW, 482
OBM_OLD_REDUCE, 483
OBM_OLD_RESTORE, 483
OBM_OLD_RGARROW, 482
OBM_OLD_UPARROW, 483
OBM_OLD_ZOOM, 483
OBM_REDUCE, 483
OBM_REDUCED, 483
OBM_RESTORE, 483
OBM_RESTORED, 483
OBM_RGARROW, 482
OBM_RGARROWD, 482
OBM_RGARROWI, 482
OBM_SIZE, 483
OBM_UPARROW, 483
OBM_UPARROWD, 483
OBM_UPARROWI, 483
OBM_ZOOM, 483
OBM_ZOOMD, 483
ODA_DRAWENTIRE, 644
ODA_FOCUS, 644
ODA_SELECT, 644
ODS_DISABLED, 644
ODS_FOCUS, 644
ODS_SELECTED, 644
ODT_BUTTON, 644
ODT_COMBOBOX, 644
ODT_LISTBOX, 644
ODT_MENU, 644
OEM_CHARSET, 462; 477
OEM_FIXED_FONT, 437
ON_BN_CLICKED, 622
ON_BN_DOUBLECLICKED, 622
ON_CBN_CLOSEUP, 649
ON_CBN_DBLCLK, 649
ON_CBN_DROPDOWN, 649
ON_CBN_EDITCHANGE, 649
ON_CBN_EDITUPDATE, 650
ON_CBN_ERRSPACE, 650
ON_CBN_SELCHANGE, 650
ON_CBN_SELENDCANCEL, 650
ON_CBN_SELENDOK, 650
ON_CBN_SETFOCUS, 650
ON_EN_CHANGE, 626
ON_EN_ERRSPACE, 626
ON_EN_HSCROLL, 626
ON_EN_KILLFOCUS, 626
ON_EN_MAXTEXT, 626
ON_EN_SETFOCUS, 626
ON_EN_UPDATE, 626
ON_EN_VSCROLL, 626
ON_LBN_DBLCLK, 634
ON_LBN_ERRSPACE, 635
ON_LBN_KILLFOCUS, 635
ON_LBN_SELCANCEL, 634
ON_LBN_SELCHANGE, 634
ON_LBN_SETFOCUS, 634
ON_PARSE_COMMAND, 956
ON_PARSE_COMMAND_PARAMS, 957
ON_WM_CHARTOITEM, 634
ON_WM_VKEYTOITEM, 635
OnParseError, 955
OnWriteBody, 955
OPAQUE, 437
OPENFILENAME, 379
OUT_CHARACTER_PRECIS, 478

OUT_DEFAULT_PRECIS, 478
OUT_DEVICE_PRECIS, 478
OUT_OUTLINE_PRECIS, 478
OUT_RASTER_PRECIS, 478
OUT_STRING_PRECIS, 478
OUT_STROKE_PRECIS, 478
OUT_TT_ONLY_PRECIS, 478
OUT_TT_PRECIS, 478
OWNERREDRAW, 666

P

PAGESETUPDLG, 414
PARAFORMAT, 798
PATCOPY, 453
PATINVERT, 453
PATPAINT, 453
PeekMessage, 142; 156
PFA_CENTER, 799; 805
PFA_LEFT, 799; 805
PFA_RIGHT, 799; 805
PFM_ALIGNMENT, 799
PFM_NUMBERING, 799
PFM_OFFSET, 799
PFM_OFFSETINDENT, 799
PFM_RIGHTINDENT, 799
PFM_STARTINDENT, 799
PFM_TABSTOPS, 799
PostMessage, 42; 898; 907
PostQuitMessage, 878; 897
PostThreadMessage, 898
PropertySheet, 823
PropSheetProc, 825
PreTranslateMessage, 130
PRINTDLG, 406
process, 851
PROOF_QUALITY, 479
PROPSHEETHEADER, 822
PROPSHEETPAGE, 832
PS_ALTERNATE, 471
PS_COSMETIC, 470; 473
PS_DASH, 469
PS_DASHDOT, 469
PS_DASHDOTDOT, 469
PS_DOT, 469
PS_ENDCAP_FLAT, 471
PS_ENDCAP_ROUND, 471
PS_ENDCAP_SQUARE, 471
PS_GEOMETRIC, 470; 473
PS_INSIDEFRAME, 469
PS_JOIN_BEVEL, 471
PS_JOIN_MITER, 471
PS_JOIN_ROUND, 471
PS_NULL, 469
PS_SOLID, 469
PS_USERSTYLE, 471; 473
PSBTN_APPLYNOW, 827
PSBTN_BACK, 827
PSBTN_CANCEL, 827
PSBTN_FINISH, 827
PSBTN_HELP, 827
PSBTN_NEXT, 827
PSBTN_OK, 827
PSH_DEFAULT, 822
PSH_HASHELP, 822
PSH_MODELESS, 823
PSH_NOAPPLYNOW, 823
PSH_PROPSHEETPAGE, 823
PSH_PROPTITLE, 823
PSH_RTLREADING, 824
PSH_USECALLBACK, 823
PSH_USEHICON, 823
PSH_USEICONID, 824
PSH_USEPSTARTPAGE, 824
PSH_WIZARD, 824
PSP_DEFAULT, 832
PSP_DLGINDIRECT, 832
PSP_HASHELP, 832
PSP_RTLREADING, 833
PSP_USECALLBACK, 832
PSP_USEHICON, 833
PSP_USEICONID, 833
PSP_USEREFPARENT, 833
PSP_USETITLE, 833
PSWIZB_BACK, 846
PSWIZB_DISABLEDFINISH, 846
PSWIZB_FINISH, 846
PSWIZB_NEXT, 846

Q

QS_ALLINPUT, 932
QS_HOTKEY, 932
QS_INPUT, 932
QS_KEY, 932
QS_MOUSE, 932

QS_MOUSEBUTTON, 932
QS_MOUSEMOVE, 932
QS_PAINT, 932
QS_POSTMESSAGE, 933
QS_SENDMESSAGE, 933
QS_TIMER, 933

R

R2_BLACK, 438
R2_COPYPEN, 438
R2_MASKNOTPEN, 438
R2_MASKPEN, 439
R2_MASKPENNOT, 438
R2_MERGENOTPEN, 438
R2_MERGEPEN, 438
R2_MERGEPENNOT, 438
R2_NOP, 438
R2_NOT, 438
R2_NOTCOPYPEN, 438
R2_NOTMASKPEN, 439
R2_NOTMERGEPEN, 439
R2_NOTXORPEN, 439
R2_WHITE, 438
R2_XORPEN, 439
ReadClient, 956
ReadFile, 74
Realtime, 855
RECT, 233
RegisterClipboardFormat, 792
RegisterHotKey, 692
RegisterWindowMessage, 150; 156
RGB, 127
RGN_AND, 452; 491
RGN_COPY, 452; 491
RGN_DIFF, 452; 492
RGN_OR, 452; 492
RGN_XOR, 452; 492
RUSSIAN_CHARSET, 462; 477

S

SB_BOTTOM, 560; 657
SB_ENDSCROLL, 656; 657
SB_LEFT, 656
SB_LINEDOWN, 560; 657
SB_LINELEFT, 656
SB_LINERIGHT, 656
SB_LINEUP, 561; 657
SB_PAGEDOWN, 561; 657
SB_PAGELEFT, 656
SB_PAGERIGHT, 656
SB_PAGEUP, 561; 657
SB_RIGHT, 656
SB_THUMBPOSITION, 656; 657
SB_THUMBTRACK, 561; 656; 657
SB_TOP, 560; 657
SBS_BOTTOMALIGN, 657
SBS_HORZ, 657
SBS_LEFTALIGN, 658
SBS_SIZEBOX, 658
SBS_SIZEBOXBOTTOMRIGHTALIGN, 658
SBS_SIZEBOXTOPLEFTALIGN, 658
SBS_TOPALIGN, 658
SBS_VERT, 658
SC_CLOSE, 308
SC_HOTKEY, 308
SC_HSCROLL, 308
SC_ICON, 309
SC_KEYMENU, 309
SC_MAXIMIZE, 309
SC_MINIMIZE, 309
SC_MOUSEMOVE, 309
SC_MOVE, 309
SC_NEXTWINDOW, 309
SC_PREVWINDOW, 309
SC_RESTORE, 309
SC_SCREENSAVE, 309
SC_SIZE, 309
SC_TASKLIST, 309
SC_VSCROLL, 309
SC_ZOOM, 309
SCROLLINFO, 661; 662
SECURITY_DESCRIPTOR, 870
SEL_EMPTY, 790
SEL_MULTICHAR, 790
SEL_MULTIOBJECT, 790
SEL_OBJECT, 790
SEL_TEXT, 790
semaphore, 863
SendMessage, 42
SetChunkSize, 956
SetClassLong, 194
SetEndOfFile, 74
SetGraphicsMode, 476
SetPriorityClass, 854

SetThreadPriority, 855
SetWindowLong, 195; 743
SeverSupportFunction, 956
SF_RTF, 810
SF_TEXT, 810
SHIFTJIS_CHARSET, 462; 477
SIF_ALL, 661
SIF_DISABLENOSCROLL, 661
SIF_PAGE, 661
SIF_POS, 661
SIF_RANGE, 661
splitBar, 580
splitBorder, 580
splitBox, 580
splitIntersection, 580
SRCAND, 454
SRCCOPY, 454
SRCERASE, 454
SRCINVERT, 454
SRCPAINT, 454
SS_BITMAP, 614
SS_BLACKFRAME, 613
SS_BLACKRECT, 613
SS_CENTER, 613
SS_CENTERIMAGE, 614
SS_ENHMETAFILE, 614
SS_GRAYFRAME, 613
SS_GRAYRECT, 613
SS_ICON, 613; 614
SS_LEFT, 613
SS_LEFTNOWORDWRAP, 613
SS_NOPREFIX, 613
SS_RIGHT, 614
SS_SIMPLE, 614
SS_USERITEM, 614
SS_WHITEFRAME, 614
SS_WHITERECT, 614
STRETCH_ANDSCANS, 439
STRETCH_HALFTONE, 439
STRETCH_ORSCANS, 439
SuspendThread, 899
SYMBOL_CHARSET, 462; 477
SYNCHRONIZE, 901
SYSTEM_FIXED_FONT, 437
SYSTEM_FONT, 437

T

TA_UPDATECP, 457; 459
TB_BOTTOM, 704
TB_ENDTRACK, 704
TB_LINEDOWN, 704
TB_LINEUP, 704
TB_PAGEDOWN, 704
TB_PAGEUP, 704
TB_THUMBPOSITION, 704
TB_THUMBTRACK, 704
TB_TOP, 704
TBNOTIFY, 256
TBS_AUTOTICKS, 705
TBS_BOTH, 705
TBS_BOTTOM, 705
TBS_ENABLESELRANGE, 705
TBS_HORZ, 705
TBS_LEFT, 705
TBS_NOTICKS, 705
TBS_RIGHT, 705
TBS_TOP, 705
TBS_VERT, 705
TC_ITEM, 842
TCIF_IMAGE, 842
TCIF_PARAM, 843
TCIF_TEXT, 842
TCN_KEYDOWN, 839
TCN_SELCHANGE, 839
TCN_SELCHANGING, 839
TCS_BUTTONS, 840
TCS_FIXEDWIDTH, 840
TCS_FOCUSNEVER, 840
TCS_FOCUSONBUTTONDOWN, 840
TCS_FORCEICONLEFT, 840
TCS_FORCELABELLEFT, 840
TCS_MULTILINE, 840
TCS_OWNERDRAWFIXED, 840
TCS_RAGGEDRIGHT, 840
TCS_RIGHTJUSTIFY, 840
TCS_SHAREIMAGELISTS, 840
TCS_SINGLELINE, 840
TCS_TABS, 840
TCS_TOOLTIPS, 840
TerminateExtension, 955
TerminateThread, 865; 900
TEXTMETRIC, 460
THAI_CHARSET, 462; 477

thread, 851
THREAD_ALL_ACCESS, 901
THREAD_DIRECT_IMPERSONATION, 901
THREAD_GET_CONTEXT, 901
THREAD_IMPERSONATE, 901
THREAD_PRIORITY_ABOVE_NORMAL, 869
THREAD_PRIORITY_BELOW_NORMAL, 869
THREAD_PRIORITY_HIGHEST, 868
THREAD_PRIORITY_IDLE, 869
THREAD_PRIORITY_LOWEST, 869
THREAD_PRIORITY_NORMAL, 869
THREAD_PRIORITY_TIME_CRITICAL, 868
THREAD_QUERY_INFORMATION, 902
THREAD_SET_CONTEXT, 902
THREAD_SET_INFORMATION, 902
THREAD_SET_THREAD_TOKEN, 902
THREAD_SUSPEND_RESUME, 902
THREAD_TERMINATE, 902
TMP_CENTERALIGN, 305
TMP_LEFTALIGN, 305
TMP_LEFTBUTTON, 305
TMP_RIGHTALIGN, 305
TMP_RIGHTBUTTON, 305
TMPF_DEVICE, 461
TMPF_FIXED_PITCH, 461
TMPF_TRUETYPE, 461
TMPF_VECTOR, 461
transaction, 964
TranslateMessage, 62; 130; 156
TransmitFile, 956
TRANSPARENT, 437
TTN_NEEDTEXT, 714
TTS_ALWAYSTIP, 717
TTS_NOPREFIX, 717
TURKISH_CHARSET, 462; 477
TV_INSERTSTRUCT, 766
TV_ITEM, 767
TV_SORTCB, 772
TVE_COLLAPSE, 770
TVE_COLLAPSERESET, 770
TVE_EXPAND, 770
TVE_TOGGLE, 770
TVGN_CARET, 770
TVGN_CHILD, 770
TVGN_DROPHILITE, 770
TVGN_FIRSTVISIBLE, 770
TVGN_NEXT, 771
TVGN_NEXTVISIBLE, 771
TVGN_PARENT, 771
TVGN_PREVIOUS, 771
TVGN_PREVIOUSVISIBLE, 771
TVGN_ROOT, 771
TVI_FIRST, 766; 767
TVI_LAST, 767
TVI_SORT, 766; 767
TVIF_CHILDREN, 767
TVIF_HANDLE, 767
TVIF_IMAGE, 767
TVIF_PARAM, 767
TVIF_SELECTEDIMAGE, 767
TVIF_STATE, 767
TVIF_TEXT, 767
TVIS_CUT, 767
TVIS_DISABLED, 768
TVIS_DROPHILITED, 768
TVIS_EXPANDED, 768
TVIS_EXPANDEDONCE, 768
TVIS_FOCUSED, 768
TVIS_OVERLAYMASK, 768
TVIS_SELECTED, 768
TVIS_STATEIMAGEMASK, 768
TVN_BEGINDRAG, 772
TVN_BEGINLABELEDIT, 772
TVN_BEGINRDRAG, 772
TVN_DELETEITEM, 773
TVN_ENDLABELEDIT, 773
TVN_GETDISPINFO, 773
TVN_ITEMEXPANDED, 773
TVN_ITEMEXPANDING, 773
TVN_KEYDOWN, 773
TVN_SELCHANGED, 773
TVN_SELCHANGING, 773
TVN_SETDISPINFO, 773
TVS_DISABLEDRAGDROP, 764
TVS_EDITLABELS, 764
TVS_HASBUTTONS, 764
TVS_HASLINES, 764
TVS_LINESATROOT, 764
TVS_SHOWSELALWAYS, 764
TVSIL_NORMAL, 765
TVSIL_STATE, 765

U

UD_MAXVAL, 712
UD_MINVAL, 712
UDN_DELTAPOS, 708
UnregisterHotKey, 692

V

VARIABLE_PITCH, 479
VK_DOWN, 704
VK_END, 704
VK_HOME, 704
VK_LEFT, 704
VK_NEXT, 704
VK_PRIOR, 704
VK_RIGHT, 704
VK_UP, 704

W

WA_ACTIVE, 560
WA_CLICKACTIVE, 560
WA_INACTIVE, 560
WAIT_ABANDONED_0, 933
WAIT_OBJECT_0, 933
WaitCommEvent, 861
WaitForMultipleObjects, 861; 925
WaitForMultipleObjectsEx, 861
WaitForSingleObject, 861
WaitForSingleObjectEx, 861
WHITE_BRUSH, 436
WHITE_PEN, 436
WHITENESS, 453
WHITEONBLACK, 439
WINDING, 438
WinMain, 123
WM_CHAR, 62
WM_HSCROLL, 709
WM_KEYUP, 704
WM_LBUTTONUP, 704
WM_QUIT, 62; 879
WM_SETHOTKEY, 691
WM_VSCROLL, 709
WNDCLASS, 134
WNDCLASSEX, 134
WrapNone, 806
WrapToTargetDevice, 806
WrapToWindow, 806
WriteClient, 955
WriteFile, 74
WS_GROUP, 622
WS_MAXIMIZE, 782
WS_TABSTOP, 622