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

Ю. Тихомиров

# MFC

САМОУЧИТЕЛЬ

- Обзор классов
- Поддержка работы с графикой
- Архитектура “документ/представление”
- Работа с базами данных



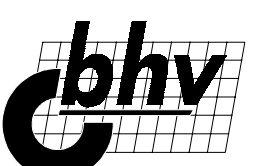


Дискета содержит примеры приложений и тексты программ

**Библиотека для разработки Windows-приложений**

Юрий Тихомиров

# САМОУЧИТЕЛЬ MFC




Дюссельдорф ◆ Киев ◆ Москва ◆ Санкт-Петербург

УДК 681.3.06

Библиотека MFC — мощный и гибкий инструмент разработки Windows-приложений на базе языка Visual C++. Книга содержит подробную информацию об основных классах библиотеки и их компонентах. Рассмотрены принципы создания одно- и многодокументных приложений, все типы окон, элементы управления и работа с базами данных. Практические примеры помогут усвоить теоретический материал и одновременно освоить современный стиль программирования. В приложении излагаются основы языка C++, что позволяет рекомендовать книгу не только подготовленным программистам, желающим познакомиться с библиотекой MFC, но и новичкам.

*Для программистов, разрабатывающих приложения для Windows*

**Группа подготовки издания:**

| | |
|---|---|
| Главный редактор | *Екатерина Кондукова* |
| Зав. редакцией | *Наталья Таркова* |
| Редактор | *Ирина Агафонова* |
| Компьютерная верстка | *Ольги Сергиенко* |
| Корректор | *Зинаида Дмитриева* |
| Дизайн обложки | *Татьяны Емельяновой* |
| Зав. производством | *Николай Тверских* |

**Тихомиров Ю. В.**
Самоучитель MFC. — СПб.: БХВ — Санкт-Петербург, 2000. — 640 с.: ил.

ISBN 5-8206-0096-7

© Ю. В. Тихомиров, 2000
© Оформление, издательство "БХВ — Санкт-Петербург", 2000

Лицензия ЛР № 065953 от 15.06.98. Подписано в печать 17.02.2000.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 51,6.
Тираж 5000 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 Академической типографии "Наука" РАН.
199034, Санкт-Петербург, 9 линия, 12.

# Содержание

**Введение** ..... **11**

**ЧАСТЬ I. БИБЛИОТЕКА MICROSOFT FOUNDATION CLASSES (MFC)** ..... **17**

**Глава 1. Знакомьтесь — библиотека классов MFC** ..... **19**

Макросы, глобальные функции и переменные ..... 20
Типы данных ..... 21
Получение информации о приложении ..... 22
Модель объекта времени выполнения (run-time object) ..... 23
Диагностика объектов ..... 24
Основные макросы ..... 24
Основные глобальные переменные ..... 25
Форматирование строк и окна сообщений ..... 25
Иерархия классов MFC ..... 28
*CObject* — вершина иерархии классов ..... 29

**Глава 2. Создание приложений на базе библиотеки классов MFC** ..... **31**

Соглашения об именах библиотеки MFC ..... 35
Включаемые файлы ..... 35

**ЧАСТЬ II. ПРОГРАММИРОВАНИЕ С ИСПОЛЬЗОВАНИЕМ БИБЛИОТЕКИ MFC** ..... **37**

**Глава 3. Основные составляющие приложения на базе библиотеки классов MFC** ..... **39**

Функция *WinMain* ..... 42
Минимальная программа для Windows ..... 46
Создание окна ..... 48

**Глава 4. Классы окон библиотеки MFC** ..... **50**

Окна, определенные в системе Windows ..... 51
Окна Windows и библиотека MFC ..... 60
Создание главного окна SDI-приложения ..... 60

Создание простейшего меню ....63
Создание главного окна приложения *(продолжение)* ....65
Создание дочерних окон ....69
Ограничение размеров окна ....79
Функция *GetSystemMetrics* ....80
Ограничение доступа к окну ....81

**Глава 5. Поговорим о сообщениях ....83**

Обработка сообщений в библиотеке MFC ....83
Цикл обработки сообщений MFC ....83
Типы сообщений MFC ....85
Карта сообщений ....87
Компоненты карты сообщений ....88
Стандартный маршрут команды ....94
Команды обновления и класс *CCmdUI* ....97
Функции для работы с сообщениями ....99

**Глава 6. Каждый должен заниматься своим делом ....101**

**ЧАСТЬ III. MFC И ГРАФИЧЕСКИЙ ВЫВОД ....109**

**Глава 7. Управление графическим выводом ....111**

Идеология графического вывода ....111
Аппаратно-независимый графический вывод ....112
Контексты устройств ....112
Типы контекстов устройств ....113
Контексты экрана ....114
Контекст принтера ....115
Объект в памяти ....115
Информационный контекст ....116
Графические объекты ....116
Графические режимы ....116
Работа со шрифтами. Шрифты TrueType ....117
Классы графического интерфейса ....118
Классы контекстов устройств ....119
Графические объекты ....121

**Глава 8. Кисти, карандаши и многое другое ....125**

Изменение начального размера окна ....125
Создание графических объектов Windows ....127
Создание шрифтов ....127
Создание кистей ....135
Создание битовых массивов ....137
Создание карандашей ....139
Работа с регионами ....143
Создание регионов ....144
Создание прямоугольных регионов ....144
Создание эллиптических регионов ....144

Создание сложных регионов на базе многоугольников ... 144
Создание регионов с закругленными углами ... 145
Комбинирование регионов ... 145
Изменение размеров окна ... 148
Обработка сообщения WM_PAINT ... 149
Организация процесса рисования ... 150

**Глава 9. Класс поддержки рисования ... 153**

Создание объекта класса и его связь с контекстом устройства ... 154
Компоненты класса ... 154
Конструктор ... 155
Инициализация ... 155
Доступ к объектам рисования ... 157
Функции контекста устройства ... 157
Настройка процесса рисования ... 158
Функции средств рисования ... 158
Установка объектов рисования ... 158
Функции настройки цветов и палитр ... 160
Режимы рисования ... 160
Режим отображения ... 163
Настройка режимов отображения ... 163
Преобразование координат ... 169
Функции рисования ... 169
Отображение регионов ... 169
Отсечения ... 169
Рисование линий ... 169
Базовые функции рисования ... 170
Битовые массивы ... 170
Пиктограммы ... 170
Строки ... 170
"Управляемый вывод" изображений ... 171
Общие параметры функций ... 171
Отображение эллипсов и многоугольников ... 172
Контуры ... 173
Отображение битовых массивов ... 175
Функции, использующие битовые массивы ... 176
Прокрутка ... 179
Управление выводом текста ... 179
Вывод текста ... 179
Информация о шрифтах ... 185
Дополнительные функции ... 185
Интерфейс низкого уровня с устройствами ... 185
Управление процессом печати документов ... 186
Метафайлы ... 186

**Глава 10. Рисуем графические объекты ... 187**

Проверка того, что окно свернуто ... 189
Установка системы координат ... 189

Рисование циферблата часов ... 190
Вывод текста ... 194
Рисование стрелок ... 195

**ЧАСТЬ IV. MFC И ВЗАИМОДЕЙСТВИЕ С ПОЛЬЗОВАТЕЛЕМ ... 197**

**Глава 11. Вводим элементы пользовательского интерфейса ... 199**

Работа с панелями инструментов ... 200
Класс *CToolBar* ... 202
Класс *CToolBarCtrl* ... 212
Работа со строкой состояния ... 220
Класс *CStatusBar* ... 221
Работа с таймером ... 225
Рисование с помощью мыши ... 229

**Глава 12. В глубине меню ... 235**

Основные типы меню ... 235
Создание меню на основе шаблона ... 244
Добавление элемента в системное меню ... 250
Создание контекстного меню ... 255
Самоотображение элементов меню ... 256
Создание собственных маркеров состояния ... 263

**Глава 13. Создание многодокументных приложений ... 267**

Класс *CMDIFrameWnd* ... 268
Класс *CMDIChildWnd* ... 271
Пример MDI-приложения ... 272
Поиск запущенного экземпляра приложения ... 273
Изменение полосы меню ... 277
Стандарт Unicode ... 282
Еще раз о сообщении WM_PAINT ... 284

**ЧАСТЬ V. ИНТЕРАКТИВНОЕ ВЗАИМОДЕЙСТВИЕ С ПРИЛОЖЕНИЕМ ... 289**

**Глава 14. Модальные и немодальные блоки диалога ... 291**

Класс *CDialog* ... 294
Создаем простейший блок диалога ... 302
Блок диалога в качестве главного окна приложения ... 306
Обмен данными с блоком диалога ... 307
Создание блока диалога на основе шаблона в памяти ... 312
Стандартные блоки диалога ... 316
Для выбора цветов не обязательно создавать новый класс ... 316

**Глава 15. Наборы свойств ... 322**

Создание набора свойств ... 323
Создание объекта "набор свойств" ... 324

Настройка окна набора свойств ... 325
Добавление страниц ... 326
Создание модального окна свойств ... 328
Создание немодального окна набора свойств ... 328
Обмен данными ... 329
Операция Apply ... 329
Операции над набором свойств ... 331
Изменение параметров отображения ... 332
Настройка страниц набора свойств ... 338
Создание страницы свойств ... 339
Изменение состояния ... 339
Переопределяемые функции ... 339
Мастера ... 341
Создание мастеров ... 342
Переопределяемые функции ... 342

**Глава 16. Эти разнообразные элементы управления ... 344**

Создание элементов управления ... 345
Создание элементов управления в редакторе ресурсов ... 345
Создание элементов управления в тексте приложения ... 345
Статические элементы управления ... 347
Изменение цвета ... 347
Список ... 349
Изменение параметров списка ... 349
Функции для работы с содержимым списка ... 350
Операции над элементами списка ... 352
Переопределяемые функции ... 354
Список, имеющий флажки ... 355
Функции для работы с расширенным списком ... 356
Виртуальные функции ... 357
Комбинированный список ... 363
Просмотр видеоклипов ... 363
Элемент управления "анимация" ... 364
Уведомления ... 365
Индикатор ... 365
Пример использования просмотра видеоклипов и индикатора ... 367
Счетчик ... 370
Уведомления ... 370
Автоматическое изменение ... 372
Параметры элемента управления ... 373
Просмотр списка ... 375
Режимы вывода ... 375
Создание элемента "просмотр списка" ... 377
Работа со столбцами ... 378
Параметры просмотра списка ... 380
Работа со списком в целом ... 381
Списки изображений ... 383
Виртуальные списки ... 384
Основные и дополнительные поля ... 384

Записи по запросу (Callback Items) ... 385
Изменение содержимого списка ... 386
Поиск и сортировка записей ... 388
Редактирование надписей записей ... 392
Обработка уведомлений ... 392
Реализация просмотра списка с возможностью перемещения записей ... 393
Переопределяемые функции ... 394
Пример реализации просмотра списка ... 394
Просмотр дерева ... 398
Создание элемента управления "просмотр дерева" ... 398
Списки изображений ... 399
Функции для работы с просмотром дерева в целом ... 399
Изменения содержимого дерева ... 400
Поиск и сортировка записей дерева ... 407
Обработка уведомлений ... 410
Пример реализации просмотра дерева ... 411

**ЧАСТЬ VI. АРХИТЕКТУРА "ДОКУМЕНТ/ПРЕДСТАВЛЕНИЕ" ... 413**

**Глава 17. Основы архитектуры "документ/представление" ... 415**

Создание различных типов документов ... 418
Шаблоны однодокументных приложений ... 422
Шаблоны многодокументных приложений ... 424
Место объекта-приложения в архитектуре "документ/представление" ... 425
Роль фреймов в архитектуре "документ/представление" ... 429
Создание каркаса приложения на базе архитектуры "документ/представление" ... 435

**Глава 18. Документ и его представления ... 438**

Документы ... 439
Класс *CDocument* ... 441
Сериализация ... 447
Представления ... 453
Класс *CView* ... 456
Класс *CCtrlView* ... 460
Класс *CEditView* ... 461
Класс *CScrollView* ... 467
Класс *CSplitterWnd* ... 471

**Глава 19. Печать и предварительный просмотр документов ... 478**

Выбор и настройка параметров принтера ... 480
Создание контекста устройства ... 483
Печать документов и библиотека MFC ... 483
Предварительный просмотр документа ... 493

**ЧАСТЬ VII. РАБОТАЕМ С БАЗАМИ ДАННЫХ ... 503**

**Глава 20. Библиотека MFC и базы данных ... 505**

Что такое ODBC ... 505

Проект MFC AppWizard ... 506
Классы для работы с ODBC ... 512
Класс *CDatabase* ... 512
Создание соединения ... 512
Атрибуты данных ... 515
Операции ... 515
Класс *CRecordset* ... 516
Компоненты данных ... 516
Конструирование ... 518
Атрибуты результирующего набора ... 522
Операции обновления результирующего набора ... 524
Операции перемещения по результирующему набору ... 526
Другие операции над результирующим набором ... 528
Переопределяемые методы ... 530
Класс *CRecordView* ... 531
Создание объекта ... 532
Атрибуты данных ... 532
Операции ... 533
Класс *CFieldExchange* ... 534

**Глава 21. Настройка приложения на работу с базами данных ... 538**

Извлечение информации из базы данных ... 538
Подготовка формы для отображения данных ... 538
Отображение и обновление содержимого базы данных ... 541
Добавление и удаление записей в таблице ... 547
Добавление записей в таблицу ... 548
Удаление записей из таблицы ... 556
Сортировка записей ... 558
Поиск информации в базе данных ... 562

**ЧАСТЬ VIII. ПРИЛОЖЕНИЯ ... 569**

**Приложение 1. Основы языка программирования C++ ... 571**

Дополнительные сведения ... 572
Комментарии ... 572
Ключевые слова ... 572
Константы ... 572
Блочные объявления ... 573
Ссылки ... 574
Имена перечислений, структур и объединений ... 575
Распределение памяти ... 575
Встраиваемые функции ... 576
Перегрузка функций ... 576
Задание параметров функции по умолчанию ... 577
Операции ... 577
Библиотеки потоков ... 578
Классы ... 579
Инкапсуляция ... 579

Разграничение доступа (скрытие данных и методов) ... 583
Друзья классов ... 584
Конструкторы и деструкторы ... 586
Конструктор по умолчанию ... 589
Конструктор копирования ... 590
Несколько слов о деструкторах ... 590
Наследование ... 591
Виртуальные функции — полиморфизм ... 599
Перегрузка операций ... 601
Шаблоны ... 602
Обработка исключений ... 604
Исключения C++ ... 607
Типы исключений ... 613
Специальные функции ... 614
Порядок обработки исключений ... 616
Рекомендации по использованию ... 617

**Приложение 2. Основные типы сообщений Windows ... 620**

Аппаратные сообщения ... 621
Сообщения обслуживания окна ... 623
Сообщения об организации интерфейса пользователя ... 625
Сообщения о завершении ... 627
Частные сообщения ... 627
Информационные сообщения системных ресурсов ... 628
Сообщения о совместном использовании данных ... 629
Внутрисистемные сообщения ... 630

**Приложение 3. Описание сопроводительной дискеты ... 631**

Установка примеров ... 631
Использование примеров ... 631
Список приложений ... 632

**Предметный указатель ... 633**

# Введение

Получаю письма: "Помогите стать актером".
Отвечаю: "Бог поможет!"
*Фаина Раневская*

Приходит программист на стрельбы. Делает десять выстрелов — все мимо. Инструктор в недоумении разводит руками: "Как же так?" Программист подходит вплотную к мишени, нажимает на курок — естественно, попадает в "десятку" и говорит: "У меня все в порядке — это у вас не работает". Это, конечно же, анекдот, но любой человек, занимающийся программированием, подтвердит, что сам не раз оказывался в ситуации, когда был полностью уверен, что протестировал свою программу вдоль и поперек и "выловил" все ошибки, и все же программа, оказавшись в руках заказчика, непостижимым образом начинала выдавать всякую чушь. Не зря же считается, что любая прекрасно работающая программа содержит как минимум две ошибки.

К чему я это рассказываю? В названии книги, которую вы держите в руках, есть слово "самоучитель". Другими словами, вы в той или иной степени — новичок. И именно поэтому вам мой первый совет — разрабатывая любую, пусть самую простую программу, всегда старайтесь предусмотреть все, даже самые невероятные ситуации, которые могут возникнуть в процессе ее эксплуатации. Помните: ответственность за сбои лежит на программисте, а не на пользователе.

Данная книга посвящена программированию для системы Windows. 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.

Какой бы язык программирования вы не выбрали, очевидно, что базой будет Windows API (Application Programming Interface, Интерфейс прикладного программирования): только освоив его, можно писать программы, использующие все возможности, предоставляемые операционной системой. Одна-

ко сложность современных приложений поднялась на такой уровень, что разработка коммерческого программного обеспечения с использованием только Windows API и языка C уже не может удовлетворить программиста. В настоящее время предлагается два подхода к построению сложного ПО. Первый — это использование систем визуального программирования, а второй — применение библиотек классов, которые самостоятельно выполняют массу черновой работы и при этом гораздо в большей степени, нежели сама система Windows, поддерживают программиста, структурируя и облегчая его деятельность. Не будем заниматься сравнением достоинств и недостатков — каждый вправе сам решить, что ему больше подходит.

Но если вас заинтересовала эта книга, то рискну предположить, что вы, как и я, предпочитаете создавать все сами, а не полагаться на готовые решения, несмотря на то, что поначалу много времени будет уходить на рутину. А затем, по мере накопления собственного опыта, можно перейти на хорошо известный принцип "ножниц и клея", перенося уже написанные и отлаженные фрагменты из одной программы в другую. Это обычная и, на мой взгляд, правильная практика.

Основной темой книги является изучение библиотеки MFC (Microsoft Foundation Classes, Базовые классы Microsoft). Несмотря на то, что эта библиотека сложна и многогранна, и на сегодняшний день существует более десяти ее версий, для начала работы с ней вполне достаточно знакомства с языком программирования C++. Более того, если вы еще не знакомы с этим языком — не отчаивайтесь. Поставляемые исходные тексты библиотеки MFC могут значительно помочь одновременно изучить и C++. Новичкам даже не потребуется знание принципов программирования в Windows. Дело в том, что в компилятор Visual C++ включено специальное средство, позволяющее автоматизировать процесс разработки приложения, — мастер MFC AppWizard, который создаст для вас работающую программу. Конечно, это будет только шаблон, и наполнять его конкретным содержанием придется самостоятельно. Но первоначальное знакомство со сгенерированным кодом принесет начинающим программистам значительную пользу.

Теперь несколько слов о методике изложения материала. Она несколько отличается от принятой для самоучителей и базируется на двух основных подходах:

- освоение нового материала на примере подробного разбора конкретного кода;
- разбор теоретических основ с описанием некоторых наиболее часто употребляемых функций и переменных различных классов.

На мой взгляд, такая методика позволит значительно лучше освоить спектр возможностей, предоставляемых библиотекой MFC. Сначала вы рассматриваете конкретные примеры, а потом пытаетесь реализовать уже собственные

идеи, используя приведенные описания компонентов различных классов. На основе своего, достаточно богатого опыта, могу вас заверить, что без самостоятельной работы, ограничиваясь только разбором кем-то написанного кода, научиться программировать невозможно.

И еще одно замечание по этому поводу. Первоначально я предполагал в конце каждой главы поместить вопросы и упражнения, чтобы помочь закрепить прочитанный материал. Однако в процессе работы над книгой я отказался от этой идеи. И связано это опять же с попыткой подтолкнуть вас к самостоятельной работе. На прилагаемой дискете вы найдете многочисленные примеры, в которых реализованы подходы и возможности либо совсем не описанные, либо только упоминающиеся в тексте. Попробуйте их найти и самостоятельно разработать на их основе работающие приложения. Примеры написаны для версии компилятора Visual C++ 5.0, в которую включена библиотека MFC версии 4.22. Однако рассматриваются и некоторые особенности библиотеки версии 4.23, поставляемой с пакетом Visual C++ 6.0.

Материала по MFC настолько много, что мне приходилось достаточно долго решать, какие именно вопросы изложить подробно, какие просто упомянуть, а какие вообще не затрагивать, и в какой последовательности все это изложить. В результате структура книги, состоящей из 21 главы, оказалась следующей:

- Первые две главы знакомят со структурой библиотеки MFC и процессом создания вашего первого приложения.
- В главе 3 проводится подробный разбор минимальной программы для Windows, написанной на базе библиотеки MFC и класса приложения *CwinApp*.
- Глава 4 знакомит вас с различными типами окон и одним из основополагающих классов библиотеки — *CWnd*.
- Глава 5 посвящена подробному изложению одного из основных механизмов Windows — работы с сообщениями.
- О том, как "распределить нагрузку" между классами библиотеки при создании любого приложения, рассказывается в главе 6.
- Следующие четыре главы — с 7 по 10 — посвящены вопросам организации процесса вывода информации на экран, а точнее — в окно.
- Вопросы взаимодействия программы и пользователя рассматриваются в главах 11 и 12. В главе 11 подробно разбираются вопросы работы с панелями инструментов и строкой состояния, а в главе 12 вы на примерах познакомитесь с разнообразными меню.
- Глава 13 целиком посвящена описанию процесса создания многодокументных приложений.

- В главе 14 рассматриваются основные принципы создания и работы с блоками диалога. Описывается класс *CDialog*, предназначенный для решения этих задач, и класс *CColorDialog*, поддерживающий работу с системными (стандартными) блоками диалога, использование которых позволяет настраивать практически все параметры (открытие и сохранение файла, выбор шрифта и цвета, поиск и т. д.), необходимые для работы приложений.
- В главе 15 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, выбирая соответствующую вкладку, доступную в произвольный момент времени, а в мастере переход от одного к другому может осуществляться только последовательно. Несмотря на имеющиеся различия, эти элементы управления по способу работы с ними очень схожи и базируются на одних и тех же классах — *CPropertySheet*, *CPropertyPage* и *CTabCtrl*.
- В главе 16 сосредоточена информация об элементах управления (Controls). Естественно, что я не смог рассмотреть даже большую их часть, но то, что вы найдете в этой главе, поможет вам легко разобраться и со всеми остальными элементами управления.
- В части VI (главы с 17 по 19) сосредоточено все, что необходимо знать о поддержке со стороны библиотеки MFC архитектуры "документ/представление" (document/view architecture). Данная архитектура напрямую не поддерживается Windows, поэтому описания только классов и функций явно недостаточно. Здесь представлена взаимосвязь различных классов, последовательность вызовов тех или иных функций. Эти знания позволят вам в дальнейшем "минимизировать трудозатраты" и переопределять только те функции, новое содержание которых наполнит нужным вам смыслом разрабатываемые приложения.
- Две заключительные главы — 20 и 21 — знакомят вас с вопросами создания приложений для работы с базами данных.
- Для тех, кто не знаком с языком программирования C++, я включил в книгу приложение 1, в котором вы найдете введение в объектно-ориентированное программирование и язык C++, а также некоторые нюансы его использования.
- Система Windows — это система, основанная на сообщениях. Поэтому я включил в книгу приложение 2, где представлены различные типы используемых сообщений.
- Неотъемлемой частью книги являются приложения-примеры, записанные на прилагаемой дискете. Их достаточно много и они охватывают как

то, что можно найти на ее страницах, так и те вопросы, которым из-за недостатка места не было уделено внимание.

В заключение я хотел бы поблагодарить редактора книги Ирину Агафонову, а также всех сотрудников издательства "БХВ — Санкт-Петербург", принимавших участие в подготовке рукописи к печати.

Все замечания и пожелания можно присылать по электронной почте по адресу **bhv@mail.nevalink.ru**.



# Часть I

## Библиотека Microsoft Foundation Classes (MFC)

**Глава 1.** Знакомьтесь — библиотека классов MFC

**Глава 2.** Создание приложений на базе библиотеки классов MFC

# ГЛАВА 1



# Знакомьтесь — библиотека классов MFC

Чего не понимают, тем не владеют.
*Иоганн Вольфганг Гете*

Для того чтобы выполнять какую-либо работу на профессиональном уровне, недостаточно поверхностного знакомства с тем инструментарием, которым собираешься пользоваться. Необходимо совершенно четко и ясно представлять себе внутреннюю структуру и логику работы используемого средства. В полной мере это относится и к широко применяемой в настоящее время для построения (создания) программных продуктов библиотеке классов MFC (Microsoft Foundation Class). К MFC следует подходить именно как к инструменту, который, принимая на себя большую часть черновой работы, требует от нас не просто знакомства с ним, но и глубокого изучения. Только знание всех нюансов построения и возможностей этой библиотеки позволит нам быстро и легко создавать программы любой степени сложности. В противном случае никогда не будет уверенности, что созданное приложение при любых условиях работает правильно — слишком много скрыто 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 неизвестно кем и когда обрабатываются. Так что какая уж тут уверенность.

Что же представляет собой библиотека MFC? Это набор классов, охватывающих большую часть функциональных возможностей операционных систем Microsoft Windows, а также предоставляющих разработчику значительное количество не только очень мощных дополнительных классов, но и целые механизмы, которые, не нарушая идеологию операционной системы, существенно ее расширяют и ... упрощают.

Перед создателями библиотеки стояла задача создания объектно-ориентированного интерфейса для Windows, удовлетворяющего следующим основным целям проектирования программных продуктов:

- сокращение усилий по программированию приложений для Windows;
- скорость выполнения программ, написанных с использованием библиотеки, должна быть сопоставима с программами, написанными на языке C с использованием Win32 API;
- минимальный размер вспомогательного кода;
- способность напрямую вызывать любую C-функцию Win32 API;
- легкость использования Win32 API в C++ должна быть такая же, как и при использовании традиционного C.

Надо сказать, что поставленная задача была решена на очень высоком уровне. Созданная библиотека классов охватывает все компоненты Windows — окна, блоки диалога, контексты устройств, общие объекты GDI (битовые образы и кисти), элементы управления и многие другие стандартные элементы. Суть программирования под 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 C. Наряду с этим вы можете совершенно свободно смешивать вызовы библиотеки классов с прямыми вызовами Win32 API. Кроме того, за сравнительно небольшой промежуток времени корпорацией Microsoft было разработано несколько версий библиотеки MFC, которые становились все мощнее и удобнее. Вместе с Visual C++ 6.0 поставляется версия 4.23 библиотеки, и есть все основания считать, что ее развитие будет продолжено. К тому же поддержка библиотеки MFC компиляторами и средствами разработки ПО, созданными другими компаниями, позволяет сконцентрироваться именно на ней. Но давайте рассмотрим все по порядку. Начнем с макросов, глобальных функций и переменных, которые, хотя и не входят непосредственно в библиотеку, очень тесно с ней связаны и значительно облегчают программирование, а затем перейдем к общему обзору библиотеки.

# 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. Поскольку в данной книге мы не будем подробно рассматривать работу с базами данных, в сети и с использованием OLE (этим вопросам будут посвящены следующие книги), то ограничимся изучением только основных макросов, глобальных функций и переменных. Начнем, естественно, с начала. Практически все глобальные функции начинаются с префикса "Afx". Исключение составляют большинство функций для работы с базами данных и функции, обеспечивающие обмен данными. Для всех без исключения глобальных переменных применяется префикс "Afx", а все макросы записываются заглавными буквами.

## Типы данных

Большинство типов данных полностью соответствуют представленным в SDK, однако есть типы, специфичные только для MFC.

| | |
|---|---|
| BOOL | Булевское значение |
| BSTR | 32-битный указатель на символ |
| BYTE | 8-битное целое без знака |
| COLORREF | 32-битное значение, используемое для задания цвета |
| DWORD | 32-битное целое без знака 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 константную строку символов |
| LPSTR | 32-битный указатель на строку символов |
| LPCTSTR | 32-битный указатель на константную 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 целое без знака для Win32 и 16-битное — для Windows 3.x |
| WNDPROC | 32-битный указатель на оконную процедуру |
| WORD | 16-битное целое без знака |
| 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 целое без знака, используемое для обозначения позиции элемента в коллекции |
| LPCRECT | 32-битный указатель на немодифицируемую структуру RECT |
| HINSTANCE | 32-битное целое без знака для идентификации дескриптора экземпляра приложения |

**Примечание**

Функция обратного вызова или CallBack-функция вызывается операционной системой по указателю, переданному ей ранее приложением. Это весьма удобный и часто используемый механизм подключения к стандартным функциям специфических обработчиков.

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

## Получение информации о приложении

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

`CWinApp* **AfxGetApp**()`

Возвращает указатель на единственный объект *CWinApp* приложения.

`LPCNSRT **AfxGetAppName**()`

Возвращает строку, заканчивающуюся нулем, содержащую имя приложения.

`HINSTANCE **AfxGetInstanceHandle**()`

Возвращает дескриптор текущего приложения для исполняемого файла (.EXE); в случае, если функция 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-пиктограммы.

```
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 при выгрузке библиотеки из памяти.

```
HINSTANCE AFXAPI AfxLoadLibrary(LPCTSTR lpszModuleName)
```

Загружает в память модуль (.DLL или .EXE файл), 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 (run-time object)

Как будет показано дальше, все классы, производные от *CObject* и *CruntimeClass*, приобретают определенные возможности по обслуживанию объектов, включающие динамическую информацию о классе (run-time class information), сериализацию и динамическое создание объекта.

Здесь описаны макросы, которые облегчают использование этих возможностей.

```
DECLARE_DYNAMIC(className)
```

Разрешает доступ к динамической информации об объекте класса (должен использоваться при объявлении класса); *className* — действительное C++-имя класса без кавычек.

```
IMPLEMENT_DYNAMIC(className, baseClassName)
```

Генерирует код, необходимый для получения динамической информации 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 динамической информации об этом объекте (должен использоваться при объявлении класса).

**IMPLEMENT_DYNCREATE**(className, baseClassName)

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

При использовании этих макросов вы получаете возможность использовать макрос RUNTIME_CLASS и функцию *CObject::IsKindOf* для определения класса вашего объекта во время выполнения программы.

**DECLARE_SERIAL**(className)

Генерирует C++-код, необходимый для сериализации объекта класса, производного от *CObject* (должен использоваться при объявлении класса) и доступа к динамической информации о классе. Определение этого макроса включает в себя все функциональные возможности, предоставляемые обоими макросами DECLARE_DYNAMIC и DECLARE_DYNCREATE.

**IMPLEMENT_SERIAL**(className, baseClassName, wSchema)

Генерирует C++-код, необходимый для сериализации объекта класса, производного от *CObject* (должен использоваться в реализации класса) и доступа к динамической информации о классе. Параметр *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 в себя макросы и глобальные функции, которые позволяют отслеживать распределение памяти, содержимое дампа объекта и осуществлять печать отладочной информации во время выполнения.

## Основные макросы

**ASSERT**(booleanExpression)

Прерывает выполнение программы (в отладочной версии библиотеки), если вычисляемое выражение *booleanExpression* равно FALSE, и печатает сообщение об ошибке в следующей форме:

`assertion failed in file <имя_файла> in line <номер_строки>`,

где <номер строки> определяет строку, в которой произошла ошибка.

**ASSERT_KINDOF**(className, pObject)

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

**ASSERT_VALID**(pObject)

Используется для оценки доступности внутреннего состояния *pObject* (сначала проверка *pObject* на NULL, затем вызов его метода *AssertValid)*. Если хотя бы одна проверка приводит к возникновению ошибки, то выводится сообщение, аналогичное ASSERT.

**TRACE**(exp)

Позволяет вывести на экран форматированную строку, определяемую выражением *exp*, аналогично функции *printf* для C-программ в консольных программах, 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.

Рассмотренные макросы работают только в отладочной версии библиотеки.

**VERIFY**(booleanExpression)

Действует аналогично макросу ASSERT, но для рабочей версии библиотеки.

## Основные глобальные переменные

CDumpContext **afxDump**

Предопределенная переменная, которая позволяет послать информацию в окно отладчика.

BOOL **afxTraceEnable**

Используется для разрешения или запрещения работы макроса TRACE.

# Форматирование строк и окна сообщений

Эта группа содержит три глобальные функции: две для работы со строками и одну для вывода произвольного сообщения в блок диалога.

```
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 на Visual C++ 6.0>", то после выполнения этой функции *rString* будет содержать строку "Для профессионального программирования 32-разрядных приложений — читайте книгу <Программирование на Visual C++ 6.0>".

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

Аналогична функции *AfxFormatString1*, но используется два шаблона ("%1" и "%2") и две строки для подстановки, соответственно.

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)
```

и

```
int AfxMessageBox (
  UINT nIDPrompt,
  UINT nType = MB_OK,
  UINT nIDHelp = (UINT) -1)
```

Выводит текст в окно сообщений. Первая форма этой функции выводит в окно сообщений 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 в качестве *nIDHelp* используется значение по умолчанию, равное –1, то *nIDPrompt* задает тему справки.

Обе функции возвращают одно из следующих значений:

- IDABORT — нажата кнопка Abort
- IDCANCEL — нажата кнопка Cancel или нажата клавиша <Esc>
- IDIGNORE — нажата кнопка Ignore
- IDNO — нажата кнопка No
- IDOK — нажата кнопка OK
- IDRETRY — нажата кнопка Retry
- IDYES — нажата кнопка Yes
- 0 — окно сообщений не может быть создано.

Параметр *nType* определяет тип окна сообщений и может принимать одно из следующих значений:

- MB_ABORTRETRYIGNORE — содержит три кнопки Abort, Retry и Ignore (Стоп, Повтор и Пропустить)

- MB_OK — содержит только кнопку OK
- MB_OKCANCEL — содержит две кнопки OK и Cancel (OK и Отмена)
- MB_RETRYCANCEL — содержит две кнопки Retry и Cancel (Повтор и Отмена)
- MB_YESNO — содержит две кнопки Yes и No (Да и Нет)
- MB_YESNOCANCEL — содержит три кнопки Yes, No и Cancel (Да, Нет и Отмена)

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

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

**Примечание**

В скобках приведены названия для русскоязычной версии системы Windows.

Помимо кнопок, в окне сообщений можно разместить одну из предопределенных (системных) пиктограмм (рис. 1.1—1.4).

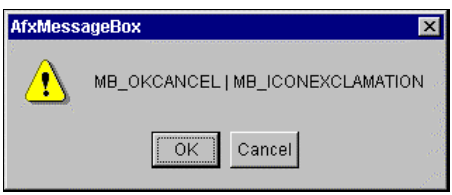



**Рис. 1.1.** MB_ICONEXCLAMATION — восклицательный знак

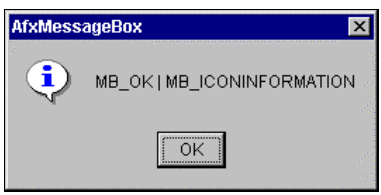



**Рис. 1.2.** MB_ICONINFORMATION — пиктограммы в виде символа "i"

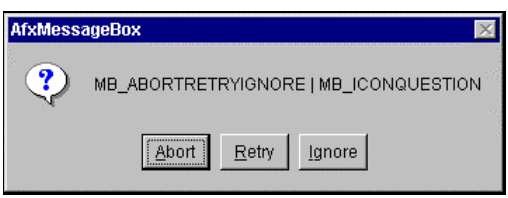



**Рис. 1.3.** MB_ICONQUESTION — знак вопроса

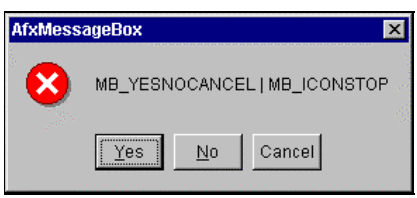



**Рис. 1.4.** MB_ICONSTOP — пиктограмма "стоп"

При использовании в блоке сообщений более одной кнопки можно также установить кнопку по умолчанию:

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

А теперь переходим собственно к структуре библиотеки MFC.

# Иерархия классов MFC

На рис. 1.5 представлена иерархия основных категорий классов библиотеки MFC. Как и положено, 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*. В зависимости от отношения к этому классу все остальные классы библиотеки MFC можно условно разбить на две группы: классы, производные от *Cobject*, и классы, не зависящие от него.

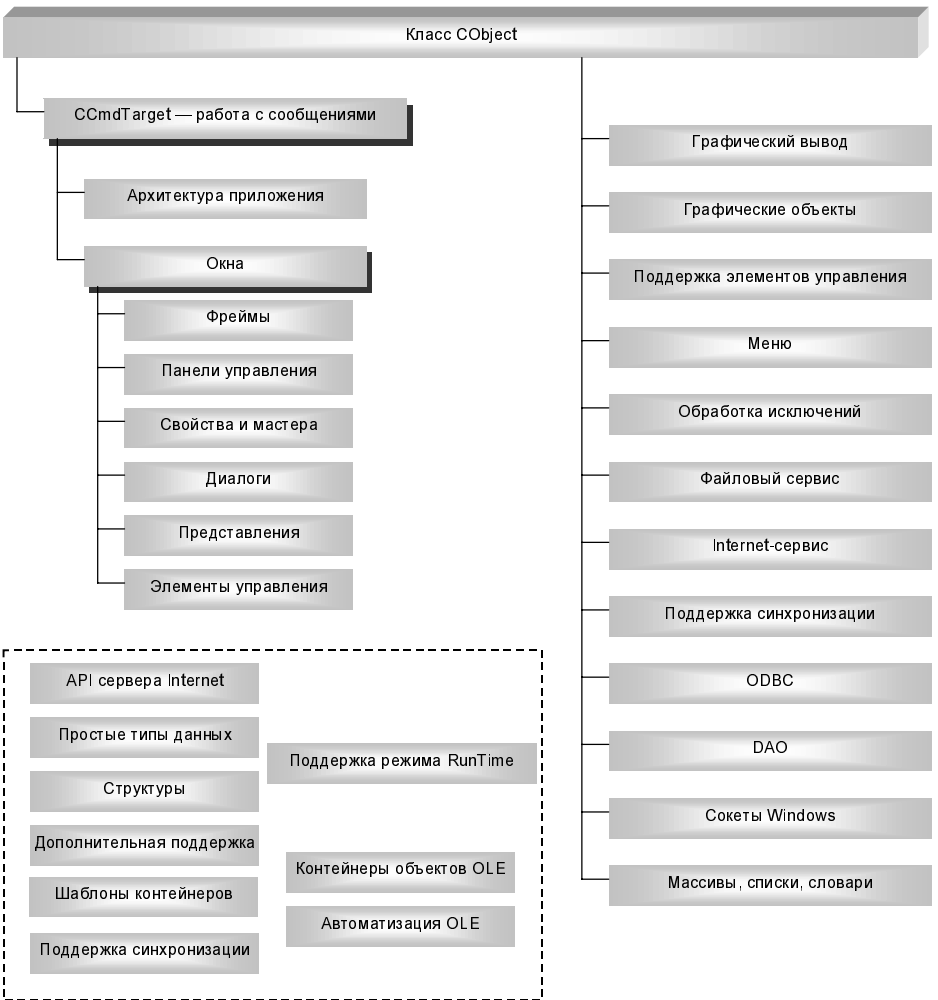



**Рис. 1.5.** Иерархия классов MFC по категориям

О структуре и возможностях библиотеки классов MFC мы будем говорить на протяжении всей книги. Здесь же я приведу описание только вершины представленной иерархии — класса *CObject*, который незримо входит практически во все остальные классы, с которыми мы будем работать.

## *CObject* — вершина иерархии классов

Большинство классов в библиотеке MFC являются производными от единственного класса *CObject* — корня иерархической структуры. Методы и элементы данных этого класса представляют наиболее общие свойства производных от него классов MFC. Основное назначение данного класса заключается в предоставлении всем своим производным классам следующих возможностей:

- хранение информации о классе объекта во время выполнения;
- поддержка сериализации и диагностики объекта.

Единственной переменной класса *CObject* является статическая переменная *classObject* типа *CRuntimeClass*, которая хранит информацию об ассоциированном с классом *Cobject* объекте во время выполнения программы.

Класс *CObject* не поддерживает множественного наследования, т. е. создаваемые вами классы могут иметь только один *CObject* в качестве базового, и он должен быть "самым левым" в иерархии. Однако такой производный класс позволяет иметь "справа от себя" в качестве базовых несколько структур и/или 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 класс, производный от *Cobject*, ассоциируется со структурой *CRuntimeClass*, которую можно использовать для получения информации об объекте или его базовом классе во время выполнения программы.

Для получения динамической информации использование этой структуры должно сопровождаться обязательным включением 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.

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

- базовые функциональные возможности — отсутствует поддержка динамической информации о классе и сериализации, но включается управление диагностикой памяти;
- базовые функциональные возможности плюс поддержка динамической информации о классе;
- базовые функциональные возможности плюс поддержка динамической информации о классе и динамическое создание;
- базовые функциональные возможности плюс поддержка динамической информации о классе, динамическое создание и сериализация.

При создании классов, производных от *CObject*, уровень их функциональных возможностей задается выбором определенных макросов объявления и реализации.

Динамическую информацию о классе *CObject* поддерживает посредством своего метода *IsKindOf*, который 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. Ее аргументом является объект класса *CRuntimeClass*, который можно получить, используя макрос RUNTIME_CLASS с именем класса. Метод *IsKindOf* не поддерживает множественного наследования или виртуальных классов, хотя при необходимости можно использовать множественное наследование для классов, производных от классов библиотеки MFC.

*CObject* также поддерживает динамическое создание, т. е. создание объекта определенного класса во время выполнения программы. Объект создается методом *CreateObject* структуры *CRuntimeClass*, как показано в примере:

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

Надеюсь, теперь у вас сложилось общее впечатление о возможностях и мощи библиотеки классов MFC и можно перейти к рассмотрению вопросов построения приложения на ее базе. Как и любое другое средство, данная библиотека требует выполнения некоторых (очень немногих) условий. Заодно познакомимся с мастером AppWizard, который выполнит за вас большую часть подготовительной работы.



# ГЛАВА 2

# Создание приложений на базе библиотеки классов MFC

Лучше делать новости, чем рассказывать о них.
*Уинстон Черчиль*

Для того чтобы написать какое-либо приложение, прежде всего необходимо создать проект. Разработчики фирмы Microsoft постарались (уже давно) и создали специальные мастера, которые позволяют — после ответов на ряд вопросов — получить работающий каркас приложения. Вам остается только наполнить его конкретным содержанием. Вызвав диалог New, выберите в нем тип приложения MFC AppWizard (exe), а в поле Project Name — имя создаваемого приложения FrameWork. После нажатия кнопки OK запустится мастер AppWizard, и вы увидите его первое окно (рис. 2.1).

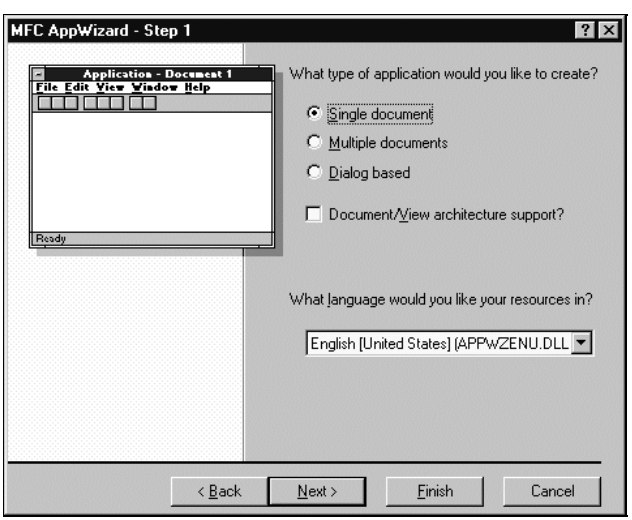



**Рис. 2.1.** Первое окно мастера AppWizard

В этом окне представлены все три основных типа приложений Windows: однодокументное, многодокументное и на базе диалога.

**Примечание**

Обратите внимание на флажок Document/View architecture support? (Нужна поддержка архитектуры документ/представление?), которого вы не увидите в версиях Visual C++ ниже 6.0. В них приходилось вручную удалять фрагменты созданного каркаса приложения, поскольку зачастую автоматически подключаемая поддержка этой архитектуры (безусловно, очень удобной и полезной) не нужна.

Сбросьте флажок Document/View architecture support? и выберите переключатель Single document, поскольку нас сейчас интересуют самые основы.

**Примечание**

Оставьте без изменения поддержку английского языка, т. к. русского все равно нет и неизвестно, когда появится.

Смело дважды нажимайте кнопку Next, чтобы перейти к третьему окну мастера, поскольку во втором окне задается поддержка для работы с базами данных. В третьем окне сбросьте флажок ActiveX Controls (элементы управления ActiveX) и переходите к следующему окну (рис. 2.2), в котором предлагается выбор некоторых дополнительных возможностей.

Примите установки, предложенные по умолчанию, и нажмите кнопку Advanced... (Дополнительно...), чтобы выбрать стили окна (рис. 2.3).

Для демонстрации влияния этих стилей, которые подробно будут описаны ниже, установим флажок Minimized (Свернутое).

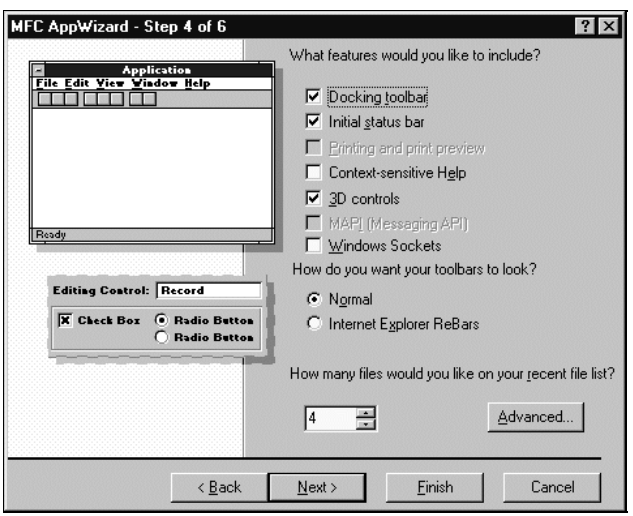



**Рис. 2.2.** Четвертое окно мастера AppWizard

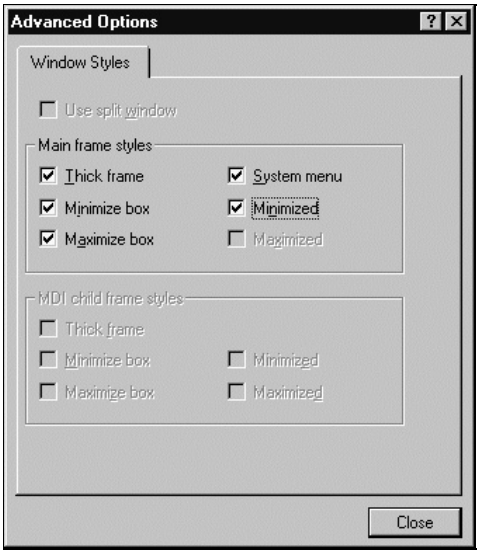



**Рис. 2.3.** Установка стиля основного окна приложения

Вернитесь к четвертому окну мастера и можете нажать кнопку Finish (Готово), поскольку в остальных окнах мы принимаем установки по умолчанию. На экране появится окно с информацией о проекте (рис. 2.4).

**Примечание**

Заданное вами имя проекта служит основой для имени класса приложения, изменить которое можно в пятом окне мастера.

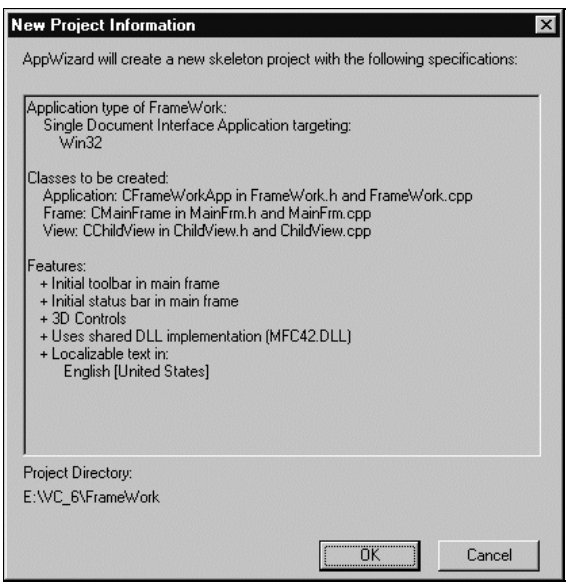



**Рис. 2.4.** Информация о вновь созданном проекте

После того как вы нажмете кнопку OK, мастер создаст для вас каркас приложения и завершит свою работу.

Постройте исполняемый файл, выполнив команду Build, и запустите его. На экране ничего не изменится, но на панели задач появится кнопка минимизированного приложения. Приложение FrameWork строит обычное, традиционное окно, хорошо знакомое всем пользователям системы Windows (рис. 2.5).

Его можно перемещать по экрану, закрыть, минимизировать (уменьшить до размера пиктограммы) или максимизировать (увеличить до размеров экрана), а также изменять его размер. Основное назначение этого примера заключается в том, что он позволяет нам познакомиться с основными принципами, которые положены в основу построения программ на базе библиотеки классов MFC.

Программа имеет три класса объектов: *CFrameWorkApp*, *CMainWnd* и *CChildWnd*. Первый позволяет создать объект "приложение", второй — объект "фрейм окна", и третий — сам объект "окно".

Каждое приложение, использующее библиотеку MFC, может содержать только один объект "приложение", который конструируется вместе с другими глобальными объектами C++ (определение — *CFrameWorkApp theApp*) и уже доступен, когда Windows вызывает функцию *WinMain*. На данном этапе знакомства — это самый существенный момент для успешного программирования под Windows на базе MFC.

Отвлечемся немного от основной темы, чтобы рассмотреть вспомогательные, но тесно связанные с программированием на Visual C++ вопросы.

**Примечание**

Библиотеку классов MFC можно использовать не только для написания полноценных Windows-приложений, но и в консольных приложениях.

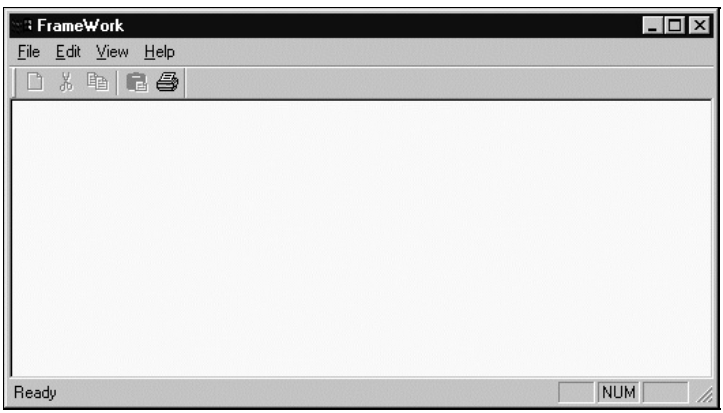



**Рис. 2.5.** Основное окно приложения FrameWork

# Соглашения об именах библиотеки MFC

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

Для имен методов классов используется три способа. При первом способе имя объединяет глагол и существительное, например, *LoadIcon* (Загрузить пиктограмму) или *DrawText* (Нарисовать текст). При втором, имя метода 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 **m**ember — член класса), затем идет префикс, характеризующий тип данных, и завершается все содержательным именем переменной, например, *m_pMainWnd*, где p — префикс, описывающий указатель (для переменных не членов класса префикс m_ не используется, чтобы можно было отличить обычные переменные от элементов любых классов, а не только классов библиотеки MFC). В своих примерах я придерживаюсь аналогичного способа формирования имен, и вам рекомендую поступать также.

# Включаемые файлы

Если вы посмотрите начало любого файла, то заметите, что одной из первых в них стоит строка `#include "stdafx.h"`. Этот включаемый файл служит для подключения к программе необходимых ей библиотечных файлов. Когда вы будете писать большие приложения, вам обязательно потребуется использование заранее откомпилированных заголовочных файлов, что поможет существенно сократить требуемое время компиляции. Прекомпилированные заголовки удобны, если у вас есть несколько заголовков, которые должны быть включены во все или в большинство ваших исходных файлов. Однако их использование полезно только в том случае, если сами они не меняются между компиляциями.

Чтобы продемонстрировать вам 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 и файл <afxwin.h>, содержащий описания основных классов библиотеки и сводящий воедино все включаемые файлы, необходи-

мые для базового функционирования MFC, и <afxext.h>, содержащий описания классов общего назначения, макросы, базовые типы данных MFC и подключающий стандартные библиотеки (run-time library) языка C, и <afxdtctl.h> и <afxcmn.h>, где находятся компоненты поддержки стандартных и совместимых с Internet Explorer 4 элементов управления. 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, каждая Windows-программа должна содержать этот файл. Отличие же заключается в том, что в своем последнем виде он уже не представляет собой огромного "монстра", а разбит на четыре сравнительно небольших файла: <windef.h>, содержащий определения базовых типов, <winbase.h>, описывающий базовые функции 32-разрядного API, <wingdi.h> и <winuser.h>, включающие объявления процедур GDI и USER, а также определения констант и макросов для соответствующих модулей системы.

В заключение темы стандартных включаемых файлов скажу несколько слов об определении

```
#define VC_EXTRALEAN
```

применение которого позволяет исключить из процесса компиляции редко используемые части заголовочных файлов Windows (для C- и C++-программ, не обращающихся к MFC, определение имеет вид #define WIN32_ EXTRA_LEAN).



# Часть II

## Программирование с использованием библиотеки MFC


**Глава 3.** Основные составляющие приложения на базе библиотеки классов MFC

**Глава 4.** Классы окон библиотеки MFC

**Глава 5.** Поговорим о сообщениях...

**Глава 6.** Каждый должен заниматься своим делом...

# ГЛАВА 3



# Основные составляющие приложения на базе библиотеки классов MFC

До первого поворота всякая дорога прямая.
*Владислав Гжещик*

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

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

```
//////////////////////////////////////////////////////
// Файл First.h
// Copyright (c) 2000 Тихомиров Ю.
#ifndef __AFXWIN_H__
  #error сначала включаем '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) 2000 Тихомиров Ю.
#include "stdafx.h"
#include "First.h"
#include "MainFrm.h"

// Может быть создан один и только один объект CFirstApp
CFirstApp theApp;
CFirstApp::CFirstApp()
{
}

// 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 *pFrame = new CMainFrame;
  m_pMainWnd = pFrame;

  pFrame->Create( szWndClass,            //
                  "Первая программа",    //
                  WS_OVERLAPPEDWINDOW,   //
                  CRect(0, 0, 560, 400), //
                  NULL,                  //
                  NULL,                  //
                  0,                     //
                  NULL                   //
                );

  m_pMainWnd->CenterWindow();
  m_pMainWnd->ShowWindow(SW_SHOW);
  m_pMainWnd->UpdateWindow();

  return TRUE;
}

////////////////////////////////////////////////////////
// Файл MainFrm.h
// Copyright (c) 2000 Тихомиров Ю.

class CMainFrame : public CWnd
{
public:
  CMainFrame();
```

```
public:
  virtual ~CMainFrame();
};

/////////////////////////////////////////////////////////
// Файл MainFrm.cpp
// Copyright (c) 2000 Тихомиров Ю.

#include "stdafx.h"
#include "First.h"
#include "MainFrm.h"

// Конструктор/деструктор класса CMainFrame
CMainFrame::CMainFrame()
{
}
CMainFrame::~CMainFrame()
{
}

/////////////////////////////////////////////////////////
// Файл Stdafx.h
// Copyright (c) 2000 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>  // Ядро MFC и стандартные компоненты

/////////////////////////////////////////////////////////
// Файл Stdafx.cpp
// Copyright (c) 2000 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 First.rc
// Copyright (c) 2000 Тихомиров Ю.
...
/////////////////////////////////////////////////////////
// Пиктограмма
IDR_MAINFRAME           ICON    DISCARDABLE     "res\\First.ico"
...
```

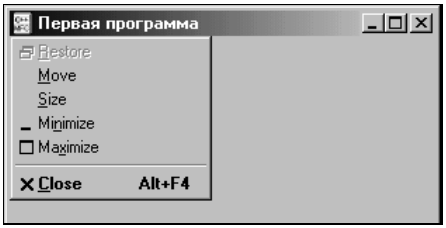



**Рис. 3.1.** Главное окно программы First

Программа First строит обычное, традиционное окно, хорошо знакомое всем пользователям системы Windows (рис. 3.1). Окно можно перемещать по экрану, закрывать, свертывать (уменьшать до размера пиктограммы) или раскрывать (увеличивать до размеров экрана), а также изменять его размер.

Основное, можно даже сказать единственное, назначение этого примера состоит в том, что он позволяет нам познакомиться с принципами построения программ на базе библиотеки классов MFC.

Давайте пройдемся по приведенному тексту, чтобы полностью разобраться в том, какой минимум действий необходимо выполнить для создания некоего каркаса, который в будущем можно будет наполнять конкретным содержанием.

Первое, на что следует обратить внимание — это наличие объекта *CFirstApp*, являющегося объектом-приложением Windows. Основной задачей класса, из которого он образован — *CWinApp*, является создание процесса в системе. Если вспомнить, что процесс — каждый выполняемый экземпляр приложения (в более ранних версиях Windows применялся термин "задача"), то становится понятным, что в приложении может быть определен только один объект класса *CWinApp*. Основными задачами, возложенными на этот класс, являются создание и инициализация главного окна (напоминаю, что Windows — "оконная" система) и опрос системных сообщений.

Итак, еще раз: 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 под Windows на базе MFC, и если вы его усвоили, то можно переходить непосредственно к функции *WinMain*, которая является "точкой входа" в любую написанную под Windows программу (точно так же, как функция main является "точкой входа" в программу, написанную на языке C).

## Функция *WinMain*

Так где же эта обязательная функция? Пусть вас не удивляет, что в представленном листинге вы не найдете даже упоминания о ней. 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 это имеет и оборотную сторону — 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 стандартной (runtime) библиотеки C/C++. Компоновщик Visual C++ знает, что она имеет имя *_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 стандартной библиотеки (runtime library) 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 вместе подробно рассмотрим фрагмент 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())
  {
   // В случае неудачи необходимо корректно завершить приложение ...
    if (pApp->m_pMainWnd != NULL)
    {
      // удалив главное окно, если оно все же было создано
      TRACE0("Warning: Destroying non-NULL m_pMainWnd\n");
      pApp->m_pMainWnd->DestroyWindow();
     }
    nReturnCode = pApp->ExitInstance();
   goto InitFailure;
  }

  // Если все в порядке — запускаем цикл обработки сообщений
  nReturnCode = pApp->Run();
  InitFailure:
  AfxWinTerm();
  return nReturnCode;
}
```

Для лучшего понимания задач, решаемых *WinMain*, рассмотрим структуру этой функции, упрощенное условное изображение которой представлено на рис. 3.2.

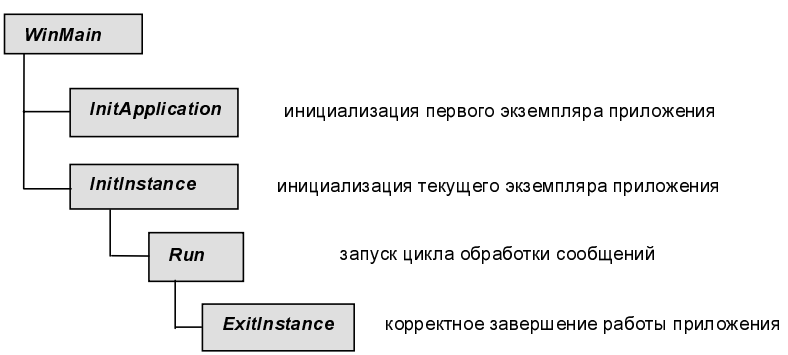



**Рис. 3.2.** Упрощенная структура функции *WinMain*

Все представленные здесь функции определены в классе *CWinApp* — базовом для объекта "приложение", и должны быть переопределены в нашем производном от него классе. Место класса *CWinApp* в иерархии объектов библиотеки MFC показано на рис. 3.3.

Поскольку без использования этого класса нельзя создать ни одно приложение на базе библиотеки MFC, то рассмотрим те основные его функции,

которые необходимо переопределить при создании конкретного приложения. Прежде всего это, конечно же, конструктор:

```
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 компоненте класса *m_pszAppName*.

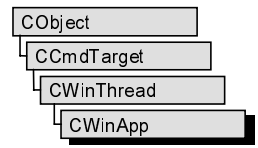



**Рис. 3.3.** Место класса *CWinApp* в иерархии

Следующую важную группу составляют 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 не делает ничего:

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

и наполнение ее содержанием целиком является задачей 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 *CWinApp*, которую вы должны *обязательно* переопределить для того, чтобы ваше приложение что-нибудь делало. Если вы вернетесь к тексту нашего приложения, то увидите, что именно это мы и сделали.

Следующие две функции не требуют обязательного переопределения, но представление о них необходимо иметь.

`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 есть необходимость в проведении специальной обработки при завершении приложения — такой, как возврат памяти, распределенной во время работы, очистка контекста устройства или других ресурсов, то следует переопределить эту функцию. Освобождение стандартных элементов, например, документов и их представлений (view), производится стандартными средствами библиотеки классов MFC и не должно беспокоить разработчиков программного обеспечения.

`virtual int CWinApp::`**`Run`** `()`

Запускает цикл обработки сообщений, в котором получаются и распределяются сообщения Windows 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.

Теперь, после некоторого 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, необходимую для создания главного окна приложения и 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 программу, написанную под Windows. Реализация этой функции в библиотеке MFC выполняет следующие действия:

- определяет указатель на объект "приложение", вызывая для этого глобальную функцию *AfxGetApp*

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

- вызывает функцию, отвечающую за инициализацию первого экземпляра приложения

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

- вызывает функцию, отвечающую за инициализацию текущего экземпляра приложения

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

- запускает цикл обработки сообщений

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

- завершает функцию *WinMain*, а значит, и работу приложения.

При всей своей лаконичности функция *WinMain* имеет все составляющие, необходимые для выполнения программы под Windows. Из этих составляющих 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:

```
pWnd->НекотораяФункцияСозданияОкна(...);
```

После того как объект окна MFC и само окно созданы, указатель на этот объект присваивается переменной *CWinThread::m_pMainWnd* (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.

К этим вопросам мы еще вернемся, а пока нам надо рассмотреть процесс создания собственно окна.

# 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*, базирующийся на одном из предопределенных классов, а именно на *CFrameWnd*, и полностью использует тот прочный фундамент, который заложила библиотека MFC, не добавляя ничего своего:

```
class CMainFrame : public CFrameWnd
{
public:
  CMainFrame();
```

```
  virtual ~CMainFrame();
};
```

В действительности все еще проще: этот класс можно было вообще не определять, поскольку обе его функции пустые. Я включил его только для наглядности. Всю работу выполняет базовый класс нашего окна *CFrameWnd*. Еще раз повторим — окно создается в два-три шага: сначала определяется оконный класс (этот шаг в большинстве случаев необязателен), 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, т. е. расширенный). Поскольку нам надо было создать главное окно программы, которое не может быть дочерним, мы воспользовались функцией *CreateEx*.

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.

Еще раз напомню, что такая последовательность действий при инициализации типична для создания всех приложений Windows, и будем двигаться дальше.

# ГЛАВА 4



# Классы окон библиотеки MFC

Труднее всего научиться общему языку.
*Лешек Кумор*

Вспомним, что Windows — это операционная система, которая прежде всего работает с окнами. Со своей стороны, подавляющую часть библиотеки MFC составляют классы окон. Пришло время подробно познакомиться и с теми и с другими, а также с их взаимодействием.

Объекты оконных классов библиотеки MFC, естественно, отличаются от окон Windows, но тем не менее тесно с ними связаны. Полное и правильное понимание взаимосвязей между ними является залогом успеха при создании приложений.

Как и любой другой объект C++, объект "окно" создается и уничтожается конструктором и деструктором класса, соответственно. 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 переменной *m_hWnd* объекта *CWnd* (рис. 4.1).

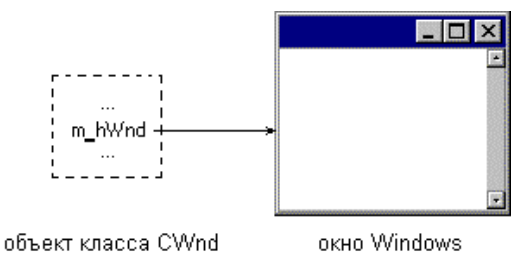



**Рис. 4.1.** Объект класса *CWnd* связан с окном Windows через дескриптор m_hWnd

Несколько возможных способов создания и работы с окнами мы рассмотрели в предыдущей главе, поэтому здесь акцентируем наше внимание на тех типах окон, которые имеются в Windows.

# Окна, определенные в системе Windows

Существует всего три основных типа окон:

- Перекрывающиеся (overlapped)
- Всплывающие или вспомогательные (popup)
- Дочерние (child)

из которых программист может создавать множество самых разнообразных объектов, комбинируя предопределенные биты стиля, имеющие имена с префиксом WS_ (от window style — стиль окна). Рассмотрим более подробно эти биты, их влияние на внешний вид и некоторые свойства окон. Начнем с простого перечисления:

| | |
|---|---|
| WS_BORDER | Задает окно, имеющее рамку без заголовка |
| WS_CAPTION | Задает окно, имеющее заголовок и рамку. Как правило, этот стиль используется для перекрывающихся окон и не может применяться совместно со стилем WS_DLGFRAME |
| WS_CHILD | Устанавливает 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 | Задает неактивное окно, т. е. сразу после его создания окно недоступно для ввода с клавиатуры или с помощью мыши |
| WS_DLGFRAME | Задает окно, имеющее двойную рамку и не имеющее 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 | Задает окно, имеющее горизонтальную полосу прокрутки |
| WS_VSCROLL | Задает окно, имеющее вертикальную полосу прокрутки |

| | |
|---|---|
| WS_MAXIMIZE | Задает отображение окна в развернутом виде |
| WS_MAXIMIZEBOX | Задает окно, имеющее кнопку "Развернуть"; если окно является дочерним окном элемента управления, то этот флаг стиля используется под другим именем — WS_TABSTOP |
| WS_MINIMIZE | Задает отображение окна в виде пиктограммы; используется только со стилем WS_OVERLAPPED |
| WS_MINIMIZEBOX | Задает окно, имеющее кнопку "Свернуть"; если окно является дочерним окном элемента управления, то этот флаг стиля используется под другим именем — WS_GROUP |
| WS_OVERLAPPED | Устанавливает, что окно является перекрывающимся; обычно имеет заголовок и рамку |
| WS_POPUP | Задает всплывающее окно; не может использоваться совместно со стилем WS_CHILD |
| WS_SYSMENU | Задает окно, имеющее пиктограмму 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 | Задает окно, имеющее утолщенную рамку, при помощи которой можно изменять размер окна |
| WS_VISIBLE | Устанавливает режим, когда окно становится видимым сразу после создания; этот стиль в основном используется для окон диалога |

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 на этот знак, то курсор изменяется на указатель в виде знака вопроса; если теперь выбрать какое-либо дочернее окно — 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 | Задает окно, имеющее удвоенную границу, что при использовании совместно со стилем WS_CAPTION позволяет создавать окно с полосой заголовка |
| WS_EX_LEFT | Устанавливает для окна свойство левого выравнивания (по умолчанию) |
| WS_EX_LEFTSCROLLBAR | Задает расположение полосы прокрутки (если она имеется) 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 | Задает дочернее окно MDI |
| WS_EX_NOPARENTNOTIFY | Определяет, что дочернее окно, созданное с этим стилем, не посылает сообщения WM_PARENTNOTIFY своему родительскому окну при создании или уничтожении |
| WS_EX_RIGHT | Устанавливает в зависимости от класса окна групповое свойство правого выравнивания |
| WS_EX_RIGHTSCROLLBAR | Задает расположение полосы прокрутки (если она имеется) с правой стороны рабочей (клиентской) области окна; используется по умолчанию |
| WS_EX_RTLREADING | Определяет порядок ввода и чтения текста справа налево |
| WS_EX_SMCAPTION | Задает окно, имеющее уменьшенную высоту полосы заголовка; используется при создании "плавающих" панелей инструментов |
| WS_EX_STATICEDGE | Определяет окно, имеющее трехмерный стиль границы, заданный для использования в элементах, к которым пользователь не получает доступа (статические элементы управления) |
| WS_EX_TOOLWINDOW | Задает окно, которое используется для создания "плавающей" панели инструментов; такое окно имеет укороченную полосу заголовка и не отображается на панели задач |

| | |
|---|---|
| WS_EX_TOPMOST | Задает окно, которое будет располагаться поверх всех окон (созданных без этого стиля); для установки или сброса этого бита можно использовать функцию SetWindowPos |
| WS_EX_TRANSPARENT | Задает прозрачное окно, т. е. не закрывающее 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 | Задает окно, имеющее 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. Главное окно приложения, как правило, имеет именно этот тип. Для их создания чаще всего используется стиль WS_OVERLAPPEDWINDOW.
- *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 минимизируется, 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 пиктограммы — 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.

Какие же возможности предоставляет нам библиотека MFC в плане создания и работы с окнами? Если обратимся к иерархии классов MFC, то вспомним, что все классы, имеющие какое-либо отношение к окнам, являются производными (прямо или косвенно) от единственного класса *CWnd* (рис. 4.2).

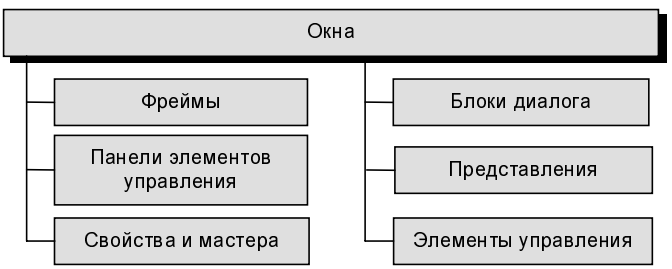



**Рис. 4.2.** Иерархия оконных классов

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.

Класс *CWnd* — это базовый класс для всех окон, созданных на базе библиотеки MFC (рис. 4.3). Помимо предоставления всем своим производным классам большого числа функций для работы с окнами, он также служит для создания разнообразных дочерних окон.

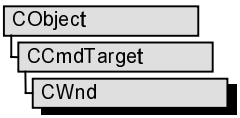



**Рис. 4.3.** *CWnd* — базовый класс всех окон библиотеки MFC

Из имеющихся компонентов класса *CWnd* для нас на данном этапе интерес представляет один:

`HWND CWnd::m_hWnd`

*m_hWnd* является дескриптором 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 и 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 *Create* и *CreateEx*, отвечающие за создание окна Windows (соответствующей ему внутренней структуры) и присоединение его к объекту класса; виртуальная функция *PreCreateWindow*, позволяющая зарегистрировать используемый по умолчанию класс окна "AfxWnd"; функции *GetStyle* и *GetExStyle*, позволяющие узнать текущий обычный или расширенный стиль окна; функции *Attach* и *Detach*, позволяющие присоединить или отсоединить окно Windows от объекта класса *CWnd*; функции *GetSafeWnd* для получения дескриптора окна Windows и *FromHandle* и *FromHandlePermanent* для получения указателя на объект по дескриптору; функции *PreSubclassWindow* и *SubclassWindow* для работы с подклассами окна, 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 пиктограммы, ассоциированной 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 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 дочерние 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 перечислю 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.* Служит 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 (*GetWindowTextLength*), 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 имеют достаточно специфическую направленность и будут обсуждаться в соответствующих разделах книги. К ним относятся следующие категории: функции для работы с полосами прокрутки, каретками, 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 до-

черних 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.

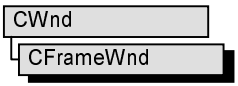



**Рис. 4.4.** Основной класс для создания фреймов

В отличие от своего базового класса *CWnd* класс *CFrameWnd* (рис. 4.4) 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 пользователя с приложением. Оно отражается на экране в виде рамки с необязательной строкой состояния и стандартными элементами управления. Это окно отвечает за управление размещением своих дочерних окон и других элементов рабочей области. Кроме того, фрейм переадресует команды своим представлениям (специальным дочерним окнам) и может отвечать на сообщения от элементов управления.

**Примечание**

Понятие "представление" (view) очень тесно связано с архитектурой "документ/представление", которую мы рассмотрим в одной из следующих глав.

Класс *CFrameWnd* спроектирован таким образом, чтобы взять на себя выполнение всех основных функций приложения Windows на базе интерфейса SDI:

- Передача команды и многих общих извещений в активное текущее представление.
- Управление размещением панелей элементов управления, представлений и других дочерних окон внутри своей рабочей области, обновление панели инструментов и кнопок панелей элементов управления во время цикла ожидания, обработка команд по умолчанию для показа или скрытия панели инструментов и строки состояния.
- Управление 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.
- Возможность использовать специальную *таблицу назначений клавиш* (accelerator table), которая автоматически транслирует комбинации клавиш в команды.

- Наличие необязательного идентификатора справки, который используется для контекстно-зависимой справочной системы. Управление самим механизмом контекстно-зависимой справочной системы (вызов комбинацией клавиш <Shift>+<F1>) и режимом предварительного просмотра перед печатью.
- Поддержка выполнения операций drag-and-drop для работы файлами. На перенос зарегистрированных и ассоциированных с приложением типов файлов класс реагирует генерацией требования открыть канал динамического обмена данными (DDE).
- Предложение сохранить модифицированные документы, выдаваемые в случае, если пользователь закрывает приложение, в котором фрейм является главным окном (определяется значением *CWinThread::m_pMainWnd*).

Из определенных в классе *CFrameWnd* компонентов мы остановимся на одном:

`static AFX_DATA const CRect CFrameWnd::rectDefault`

Компонент определяет константный объект класса *CRect*, который предписывает Windows самостоятельно установить размеры и положение создаваемого окна и задается следующим образом:

```
const AFX_DATADEF CRect CFrameWnd::rectDefault(
  CW_USEDEFAULT,
  CW_USEDEFAULT,
  0,
  0
);
```

Функции этого класса можно сгруппировать 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 командных клавиш — *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 — класс *CFrameWnd* служит базовым для классов *CMDIFrameWnd* и *CMDIChildWnd*, которые созданы специально для организации приложений на базе MDI. Но о них несколько позже.

Еще раз подчеркнем важный момент — на базе этого класса легко и удобно создавать фреймы окон. А мы пока переходим к "развитию" нашего первого приложения. А именно — к созданию и работе с различными типами окон Windows.

# Окна Windows и библиотека MFC

Пора более подробно рассмотреть процесс создания и работы с окнами. В этом разделе мы познакомимся с тем, как

- создать главное окно приложения;
- создать 10 дополнительных окон с различными комбинациями оконных стилей в каждом;
- обрабатывать нажатие левой кнопки мыши.

В результате мы должны получить нечто похожее на представленное на рис. 4.5.

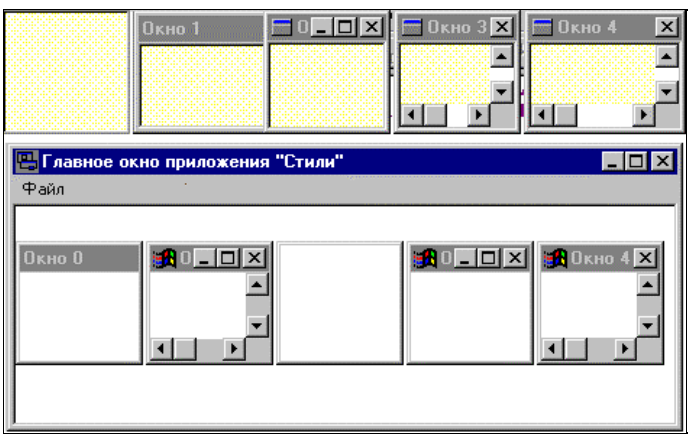



**Рис. 4.5.** Различные типы окон приложения Styles

# Создание главного окна SDI-приложения

В рассмотренной в предыдущей главе программе First мы образовали класс нашего окна из базового *CWnd*. Это допустимый, но очень неэффективный путь. Пришло время познакомить вас с другим возможным подходом для создания главного окна SDI-приложения.

**Примечание**

Такой подход стал поддерживаться только в шестой версии Visual C++. До этого все необходимые действия приходилось выполнять вручную. Но даже если вы пользуетесь Visual C++ 6, рекомендую не пропускать эту главу, поскольку здесь вы найдете подробнейшее описание не только того кода, который генерирует AppWizard, но и много другой полезной информации.

Прежде всего обратим внимание на базовый класс нашего главного окна:

```
class CMainFrame : public CFrameWnd
{
  ...
};
```

Надеюсь, здесь уже нет необходимости объяснять, почему в качестве базового был выбран класс *CFrameWnd*? Действительно, 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. Теперь нам надо подробно рассмотреть функцию *CStylesApp::InitInsnance*, где, как вы помните, и начинается создание главного окна:

```
BOOL CStylesApp::InitInstance()
{
  // Создаем объект класса CMainFrame
  CMainFrame *pMainWnd = new CMainFrame;
  m_pMainWnd = pMainWnd;

  // Создаем собственно окно Windows
  pMainWnd->LoadFrame (IDR_MAINFRAME);

  // Определяем размеры экрана
  int cx = ::GetSystemMetrics(SM_CXSCREEN)/8;
  int cy = ::GetSystemMetrics(SM_CYSCREEN)/4;

  // В зависимости от них устанавливаем размеры окна
  // и выводим его на экран
  pMainWnd->SetWindowPos (NULL,
                          cx,
                          cy,
                          6*cx,
                          2*cy,
                          SWP_NOZORDER | SWP_SHOWWINDOW);
  return TRUE;
}
```

Как видите, процесс создания окна состоит из трех шагов. Прежде всего необходимо создать объект "главное окно":

```
CMainFrame *pMainWnd = new CMainFrame;
```

Следующий шаг состоит в вызове специальной функции, которая создает окно Windows и присоединяет его к нашему объекту:

```
pMainWnd->LoadFrame(IDR_MAINFRAME);
```

И наконец, следует отобразить окно на экране, чтобы пользователь мог с ним взаимодействовать:

```
pMainWnd->SetWindowPos (NULL,
                        cx,
                        cy,
                        6*cx,
                        2*cy,
                        SWP_NOZORDER | SWP_SHOWWINDOW);
```

Создание объекта класса не должно вызвать никаких вопросов, и поэтому рассмотрим процесс создания окна Windows или, используя терминологию MFC, фрейма окна.

Из двух функций, реализованных в классе *CFrameWnd*, в данном случае мы остановили свой выбор на виртуальной функции *CFrameWnd::LoadFrame*, которая берет на себя задачу регистрации 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)
```

Функция возвращает значение TRUE, если создание фрейма прошло успешно, и FALSE — в противном случае. Она имеет следующие параметры: *nIDResource* — идентификатор ассоциированных с фреймом разделяемых ресурсов, который определяет меню, таблицу командных клавиш (accelerator keys), пиктограмму и строку, помещаемую в заголовок окна; *pParentWnd* — указатель на родительское окно данного фрейма (необязательный параметр и для главного окна должен быть равен NULL — ведь главное окно не может иметь "родителя"); *pContext* — указатель на структуру *CCreateContex*, которая определяет объекты, связанные с данным фреймом; *dwDefaultStyle* — стиль фрейма (необязательный параметр). Дополнительно к стандартным стилям окна могут быть использованы еще и следующие:

| | |
|---|---|
| FWS_ADDTOTITLE | В конец заголовка окна будет добавлена дополнительная текстовая строка |

| | |
|---|---|
| FWS_PREFIXTITLE | Перед заголовком окна будет помещено имя документа |
| FWS_SNAPTOBAR | Размеры окна будут подогнаны под размеры панели элементов управления |

Основное назначение функции *LoadFrame* — максимальное упрощение процесса создания фрейма окна. Используя идентификатор ресурсов, функция извлекает из них необходимые параметры и... вызывает функцию *CFrameWnd::Create*, рассмотренную в предыдущей главе. Функция загружает ресурс меню и выполняет функцию *CWnd::CreateEx*, которая регистрирует имя оконного класса, заполняя структуру *WNDCLASS*, и вызывает виртуальную функцию *PreCreateWindow*.

Здесь мы немного приостановимся, поскольку сначала надо разобраться с ресурсами. Вы обратили внимание, что единственным обязательным параметром функции *LoadFrame* является идентификатор ресурсов? Говоря точнее, обязательным является только один ресурс, а именно — меню. И если его не определить, после запуска приложения получим сообщение об ошибке (рис. 4.6).

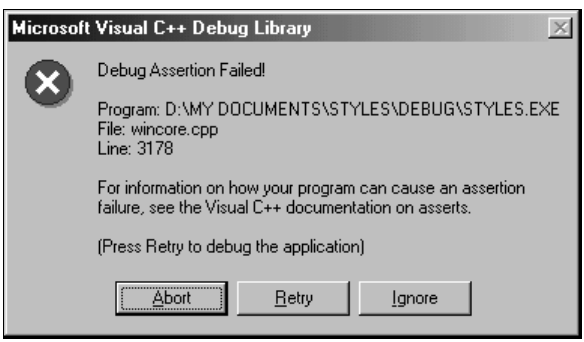



**Рис. 4.6.** Неопределение меню — единственного обязательного ресурса для работы с функцией *LoadFrame* — приводит к ошибке во время выполнения приложения

Поэтому следующим нашим шагом будет создание простейшего меню.

## Создание простейшего меню

Используя мощные возможности Visual C++, создать меню чрезвычайно просто — выполните приведенную ниже последовательность действий, и у вас сразу все заработает.

1. Раскройте вкладку Resource View в окне Workspace, щелкните правой кнопкой мыши на Styles Resource и в появившемся контекстном меню выберите пункт Insert. В результате вы увидите окно Insert Resource (рис. 4.7), в котором необходимо выделить надпись Menu и нажать кнопку New. Результат ваших действий представлен на рис. 4.8.
2. Щелкните правой кнопкой мыши на "пунктирном прямоугольнике" и в появившемся контекстном меню выберите пункт Properties. Введите на-

звание пункта меню (я использовал стандартное — Файл) в поле Caption (рис. 4.9) и обратите внимание, что под исходной полосой меню появился новый "прямоугольник". Выделите его и введите название в поле Caption, а также идентификатор в поле ID (можно выбрать один из стандартных, как мы делаем в данном конкретном случае — ID_APP_EXIT).

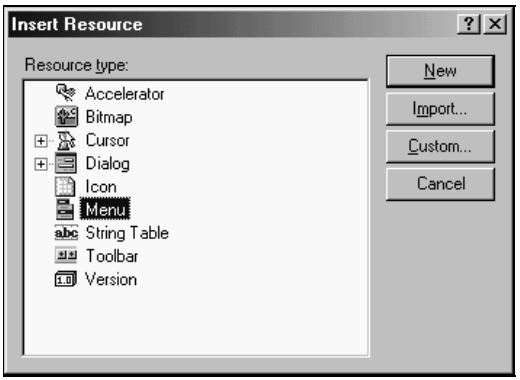



**Рис. 4.7.** Диалог для выбора типа создаваемого ресурса

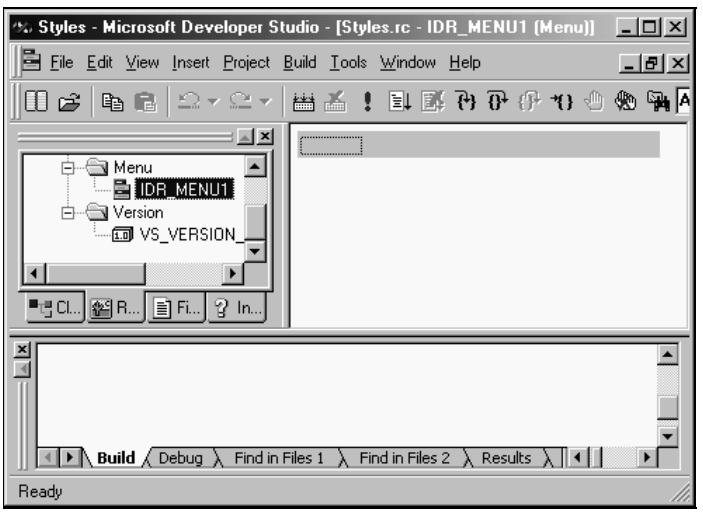



**Рис. 4.8.** Исходная точка для создания нашего меню

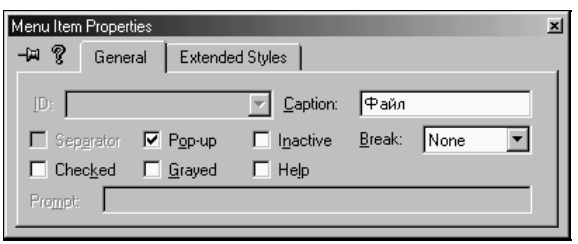



**Рис. 4.9.** В этом окне можно определить параметры элементов меню (да и других ресурсов тоже)

**Примечание**

Для того чтобы каждый раз не приходилось пользоваться правой кнопкой мыши, нажмите кнопку Keep Visible, расположенную в левом верхнем углу окна ... Properties. В результате окно всегда будет доступно.

3. Осталось присвоить соответствующий идентификатор самому созданному ресурсу. Для этого в окне Workspace выделите строку IDR_MENU1 и в поле ID окна Menu Properties введите IDR_MAINFRAME — именно такой идентификатор мы использовали в функции *LoadFrame*.
4. Вновь скомпилируйте и запустите приложение.

Результат, как видите, совершенно другой — все работает, ничего не "падает", и окно ведет себя аналогично тому, как это было в нашем первом примере. Более того, выбрав пункт меню "Выход", вы увидите, что приложение действительно закрылось. И это несмотря на то, что мы не добавили ни строчки кода. Благодаря выбранному нами идентификатору библиотека сама разобралась в том, чего мы хотим, и все сделала.

Итак, на данный момент мы подошли к тому же результату, что и в первом приложении, только "с другой стороны". Однако вернемся к нашему главному окну.

## Создание главного окна приложения *(продолжение)*

Мы остановились на том, что в процессе выполнения функции *PreCreateWindow* система Windows посылает в окно следующие сообщения: WM_GETMINMAXINFO, WM_NCCREATE, WM_NCCALKSIZE и WM_CREATE. 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/или 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 переопределения функции *CFrameWnd::PreCreateWindow*. Для этого:

1. Откройте окно ClassWizard, выбрав пункт меню View | ClassWizard или нажав аналогичную кнопку на панели инструментов (это для тех, кто как и я предпочитает работать с кнопками панели инструментов, а не через меню).
2. В раскрывающемся списке Class name выберите класс CMainFrame, а в списке Messages — PreCreateWindow (рис. 4.10). Нажмите кнопку Add Function (или дважды щелкните левой кнопкой мыши по имени PreCreateWindow) и затем кнопку Edit Code, чтобы непосредственно перейти на то место проекта, где мы будем выполнять дальнейшие действия.

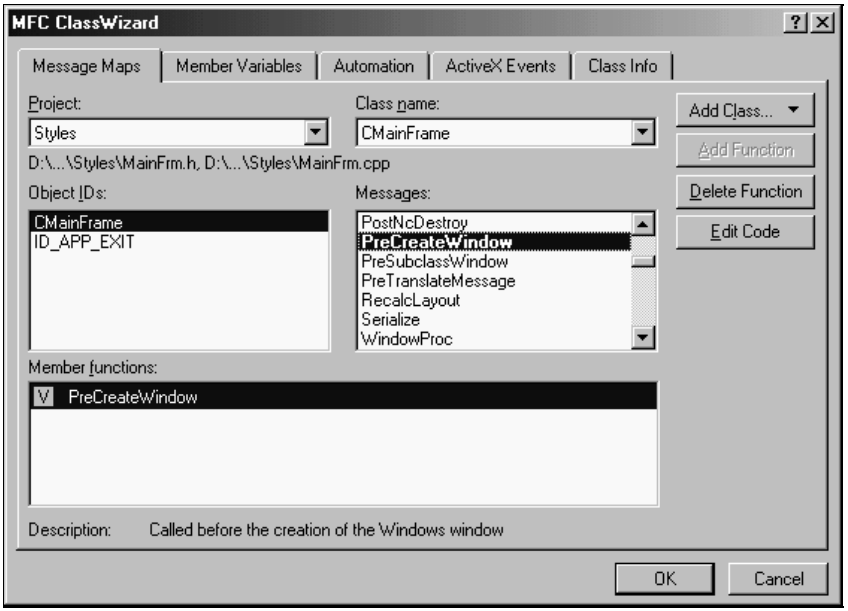



**Рис. 4.10.** ClassWizard существенно облегчает процесс создания новых функций

3. Внесите в переопределенную функцию необходимые изменения. В нашем случае она примет вид:

```
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 функцию *PreCreateWindow*, чтобы не заниматься процедурой регистрации класса окна, возложив эту задачу на библиотеку, поскольку принятые по умолчанию установки нас вполне устраивают. Единственное, что нам следует заменить, это заголовок окна:

```
cs.lpszName = "Главное окно приложения \"Стили\"";
```

Это один из возможных способов присвоения имени окну. После того как окно Windows создано и присоединено к объекту фрейма окна библиотеки MFC, ему посылается сообщение WM_CREATE. Для этого сообщения в библиотеке (точнее, в классе *CWnd*) создан специальный обработчик — *OnCreate*, который можно переопределить под свои потребности:

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
  if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
    return -1;
  return 0;
}
```

Этот обработчик сообщения — одно из возможных мест, где создаются и присоединяются к окну произвольные элементы пользовательского интерфейса, о чем мы обязательно поговорим в дальнейшем.

Для правильного завершения процесса создания окна не забывайте вызывать перед какими-либо своими действиями обработчик базового класса *CFrameWnd::OnCreate*. Вы, конечно же, понимаете, что последняя фраза относится к случаю самостоятельного создания прототипа этой функции — если воспользоваться услугами ClassWizard, то каркас функции будет вам уже предоставлен.

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 | Переместить 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 функцию *SetWindowPos* для указания размеров и расположения окна, а также для того, чтобы сделать его видимым.

**Примечание**

Как я уже говорил, существуют и другие способы решения этой задачи, многие из них также будут рассмотрены.

Следующий вопрос, на котором мы остановимся— как создаются дочерние окна.

**Примечание**

Поскольку любое порожденное окно программисты зачастую называют "дочерним", то часто возникает некоторая путаница. Вы должны совершенно четко отличать стиль 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 и/или для получения команд от пользователя. Каждому "ребенку", как и основному окну, можно приписать конкретный стиль таким образом, чтобы приспособить его внешний вид и назначение к стоящим перед ним задачам. В приложении Styles мы разбили создаваемые окна на два набора. Окна первого располагаются на экране независимо от положения основного окна и 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 создания дочерних окон необходимо проделать следующие действия:

- Создать два новых класса, каждый из которых будет отвечать за "свои" окна.
- Привязать объекты этих классов к главному окну приложения.
- Аналогично главному окну приложения создать по пять экземпляров новых классов.
- Отобразить их на экране.

Есть и другие действия, но к ним мы перейдем постепенно.

Первое из перечисленных в списке действий совершим, воспользовавшись мастером ClassWizard.

1. Откроем окно ClassWizard и в нем нажмем кнопку-меню Add Class, где выберем элемент New (рис. 4.11).

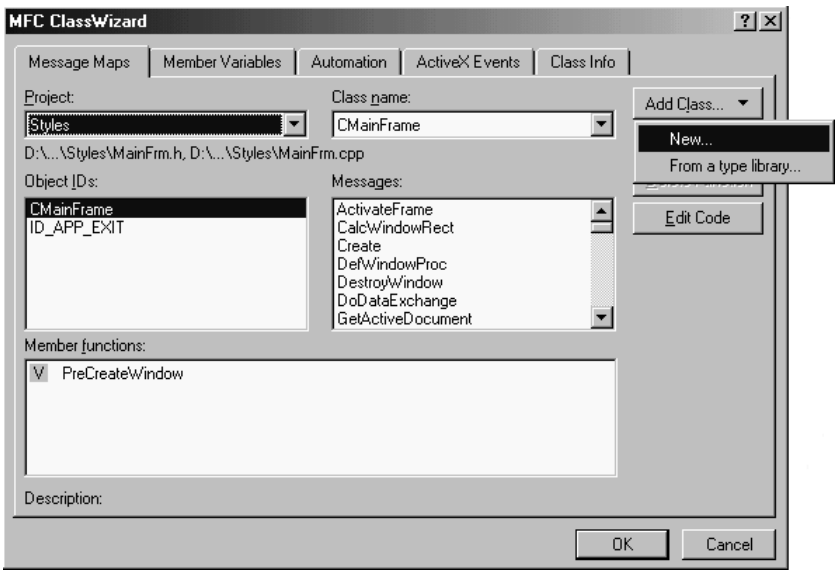



**Рис. 4.11.** Создаем новый класс

2. В результате на экране появится диалоговое окно New Class, в поле Name которого вводим имя вновь создаваемого класса — у меня это CChildWnd, а в раскрывающемся списке Base class — имя базового класса, в данном случае CWnd (рис. 4.12).
3. Нажимаем кнопку OK и новый класс создан.
4. Повторяем действия, описанные в п. 1. Только теперь не будем создавать новых файлов, а воспользуемся уже существующими. Для этого нажмем кнопку Change и в диалоговом окне Change Files выберем файлы <childwnd.h> и <childwnd.cpp> (рис. 4.13).
5. В качестве базового класса в данном случае используем класс *CFrameWnd*, поскольку *CWnd* не стоит использовать для создания всплывающих окон (рис. 4.14).

**Примечание**

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 и был использован в качестве базового.

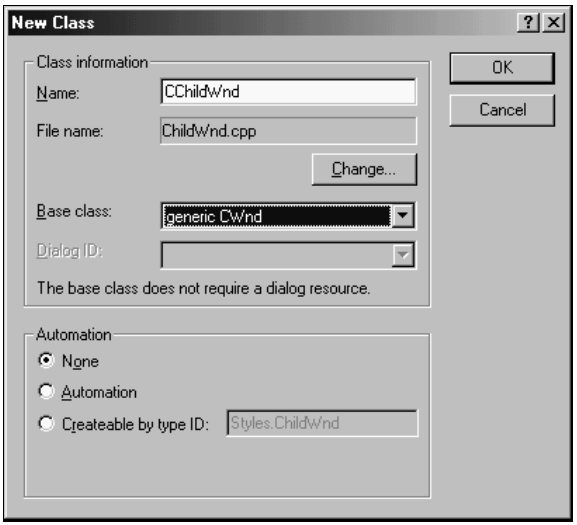



**Рис. 4.12.** Присваиваем имя и выбираем базовый класс для вновь создаваемого

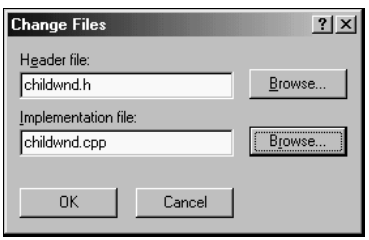



**Рис. 4.13.** Не будем создавать новых файлов

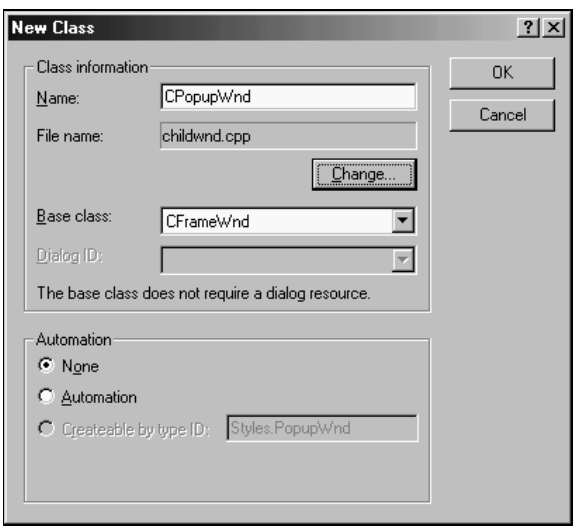



**Рис. 4.14.** Для всплывающих окон в качестве базового используем класс *CFrameWnd*

Как видите, создать новый класс чрезвычайно просто. К сожалению, это высказывание справедливо не всегда — не из всех классов библиотеки MFC можно создавать свои с помощью мастера ClassWizard.

Следующим шагом является создание объектов новых классов и их привязка к главному окну приложения. Последовательность действий при этом следующая:

1. В определении класса *CMainFrame* вводим объекты новых классов

```
class CMainFrame : public CFrameWnd
{
  ...
  CChildWnd *apChildWnd[5];
  CPopupWnd *apPopupWnd[5];
  ...
};
```

2. В конструкторе класса *CMainFrame* устанавливаем указатели на эти объекты в NULL, чтобы иметь гарантии корректности создания окон

```
CMainFrame::CMainFrame()
{
  for(int i = 0; i < 5; i++)
  {
    apChildWnd[i] = NULL;
    apPopupWnd[i] = NULL;
  }
}
```

3. Теперь осталось вернуться к обработчику сообщения WM_CREATE:

```
extern char szNameWnd[128];
...
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
  if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
 return -1;

  // 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);
  }
  // Установка нового курсора 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);
    ::SetWindowLong(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);
    // Отображение всплывающих окон
    for(i = 0; i < 5; i++)
    {
      apPopupWnd[i]->ShowWindow(SW_SHOWNORMAL);
      apPopupWnd[i]->UpdateWindow();
    }
  }
```

Как видите, при ближайшем рассмотрении это не более, чем создание десяти окон, принадлежащих двум новым классам.

В основном процесс создания окон обоих типов стандартен: создаются объекты классов, после чего вызывается некоторая функция для создания окна Windows. Чтобы продемонстрировать как можно больше предоставляемых библиотекой возможностей, в текст программы включены 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 задали режим, когда окно сразу же после создания выводится 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 функцию-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 реализаций можно одновременно изменить размер и положение окна, задаваемые в параметрах *x*, *y*, *nWidth*, *nHeight* или *lpRect*. По-

следний параметр 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, функция посылает еще и сообщение WM_GETMINMAXINFO, которое вы также можете обработать.

**Примечание**

Здесь необходимо отметить, что позиция дочернего окна определяется относительно координат 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 состоит в применении функций API *SetClassLong*, *GetClassLong*, *SetWindowLong* и *GetWindowLong*. Возможно, вы захотите использовать их в своих программах, поэтому рассмотрим эти функции 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, создаваемым 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 пиктограммы, ассоциированной с этим классом |
| 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 или нулевое значение — в случае ошибки. Дополнительную информацию об ошибке можно получить, вызвав функцию API *GetLastError*.

Мы воспользовались функцией *SetClassLong* для того, чтобы установить новые значения для курсора и цвета фона.

Обратную задачу — получение информации из структуры класса окна — выполняет функция *GetClassLong*:

```
DWORD GetClassLong (
  HWND hWnd,
  int nIndex).
```

Для работы с дополнительной памятью в структуре окна интерфейс API предоставляет также две функции:

```
DWORD SetWindowLong (
  HWND hWnd,
  int nIndex,
  LONG dwNewLong)
```

и

```
DWORD GetWindowLong (
  HWND hWnd,
  int nIndex)
```

По принципу действия эти функции 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 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, напр., 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 функцию *SetWindowLong* для записи в дополнительную память номера созданного окна. Другими словами, у первого окна в этом поле будет 0, у второго — 1, у третьего — 2 и т. д. Позже, во время изображения 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 ()
```

Эта функция 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. Нам 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 — ModifyStyle*:

```
BOOL CWnd::ModifyStyle (
  DWORD dwRemove,
  DWORD dwAdd,
  UINT  nFlags = 0)
```

Эта функция 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 *ModifyStyleEx*:

```
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;
    apPopupWnd[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 базового 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 своих размеров даже тогда, когда вы попытаетесь максимизировать 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 максимально возможных размерах и позиции окна, а также о минимальных и максимальных изменяемых размерах:

```
typedef struct tagMINMAXINFO {
  POINT ptReserved;
  POINT ptMaxSize;
  POINT ptMaxPosition;
  POINT ptMinTrackSize;
  POINT ptMaxTrackSize;
} MINMAXINFO;
```

Поля этой структуры имеют следующие назначения: *ptReserved* — зарезервировано для внутреннего использования; *ptMaxSize* — устанавливает размеры окна, которые оно может иметь в развернутом виде; *ptMaxPosition* — устанавливает значение, которое 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 пикселах.

```
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 максимизированного окна |
| SM_CXICON, SM_CYICON | Ширина и высота большой пиктограммы, 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 равен 0, 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 | Предельно 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 | Предельно 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 маленькой пиктограммы, которая 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 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 этой главы и рассмотренного в ней приложения — знакомство с основными классами библиотеки MFC, отвечающими за создание и работу практически всех типов окон Windows. Исключение составляют окна так называемых MDI-приложений — приложений многодокументного (или многооконного) интерфейса, которые мы рассмотрим несколько позже.

Здесь я попытался максимально подробно рассмотреть основные аспекты программирования для Windows с использованием библиотеки классов MFC

и показать, что слухи о сложности этого процесса "несколько преувеличены". Разобравшись в основах, изложенных в данной главе, вы уже прошли большую часть пути: программирование для Windows — это прежде всего работа с окнами.

Единственный механизм, которым мы достаточно активно пользовались, но о котором мало что сказали — работа с сообщениями. Поэтому, на мой взгляд, будет разумно отвлечься на этот чрезвычайно важный вопрос и рассмотреть его более глубоко и "теоретизированно".

# ГЛАВА 5



# Поговорим о сообщениях...

Хороша, плоха ли весть, —
Докладай мне все как есть!
*Леонид Филатов*

После того как мы познакомились со структурой классов библиотеки и рассмотрели минимальную программу на ее основе, поддерживающую однооконный интерфейс, необходимо разобраться в вопросах взаимодействия всех обязательных и вспомогательных объектов между собой.

## Обработка сообщений в библиотеке MFC

В литературе по языку C++ иногда проскальзывает такая терминология — методы класса называются *сообщениями*. И в этом есть некоторый смысл. Благодаря свойству инкапсуляции "общение" с некоторыми компонентами класса осуществляется только посредством определенных методов. Мы не будем приписывать слову "сообщение" еще одно значение, достаточно того, что "класс" и "объект" перегружены настолько, что иногда трудно сформулировать нормальное предложение, описывающее ту или иную операцию. Однако с учетом этого обстоятельства становится очевидным, что использование библиотеки классов MFC должно было оказать влияние на структуру программ для Windows, скрыв в классах многочисленные операторы `switch` или "взломщики" сообщений. Давайте для начала рассмотрим, каким образом реализуется обязательный цикл обработки сообщений при использовании библиотеки MFC.

### Цикл обработки сообщений MFC

При рассмотрении наших первых программ мы только коснулись такого важного вопроса, как обработка сообщений. И дело тут, как вы уже догады-

ваетесь, в том, что библиотека MFC опять взяла на себя решение этой важнейшей задачи. Ниже представлена функция *Run* из класса *CWinThread*, который, как вы помните, отвечает за работу с потоками:

```
int CWinThread::Run()
{
  // для отслеживания состояния времени ожидания
  BOOL bIdle = TRUE;
  LONG lIdleCount = 0;

  // получает и распределяет сообщения,
  // пока не получено сообщение WM_QUIT
  for (;;)
  {
    // фаза1: проверка возможности работы в период ожидания
    while(bIdle && !::PeekMessage(&m_msgCur, NULL,
                                  NULL, NULL, PM_NOREMOVE))
    {
      // вызов OnIdle, пока установлена bIdle
      if (!OnIdle(lIdleCount++))
        bIdle = FALSE; // присвоить состояние "нет ожидания"
    }
    // фаза2: подкачка сообщений, пока возможно
    do
    {
      // подкачка сообщения 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, практически по всем компонентам они совпадают. Сообщения выбираются из очереди при помощи функции API *::PeekMessage*, а работу по поддержке клавиатуры, выбору оконной процедуры и передачи в нее сообщений выполняет недокументированная функция *PumpMessage* из класса *CWinThread* (вы можете

в этом убедиться, если посмотрите ее исходный текст в файле ..\MFC\SRC \THRDCORE.CPP).

Основное отличие заключается в наличии 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 упоминавшаяся функция *PumpMessage*.

И еще несколько слов в заключение вводной части. Представленный цикл включает в себя 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" (довольно неточная формулировка, но, к сожалению, привести конкретные цифры не представляется возможным по вполне понятным причинам), то вы можете получить дополнительные вызовы *OnIdle* в течение одного цикла фоновой обработки. При этом параметр *lIdleCount* позволяет отслеживать номер вызова. Однако 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. как обычно выполняет за вас всю черновую работу.

Рассмотренный цикл не демонстрирует, каким образом происходит собственно обработка сообщений. Неужели необходимо переопределять оконную процедуру? Для ответа на этот вопрос следует рассмотреть, как организована обработка сообщений в библиотеке классов MFC. Но перед этим рекомендую обратиться к приложению 2, чтобы узнать, о каких сообщениях вообще идет речь.

## Типы сообщений MFC

В отличие от рассмотренной классификации сообщений, в библиотеке MFC принята несколько иная. Все возможные сообщения разделены на три основные категории:

- сообщения Windows;
- извещения элементов управления;
- командные сообщения (команды).

В первую категорию входят те сообщения, которые начинаются с префикса WM_, за исключением WM_COMMAND. Все сообщения этой категории предназначены для обработки окнами и представлениями (view) 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 "родителям". 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. Механизм передачи извещений аналогичен другим WM_-сообщениям, с одним единственным исключением. Извещение BN_CLICKED, посылаемое элементом управления BUTTON, когда пользователь щелкает по нему, трактуется как командное 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 интерфейса пользователя, которые включают меню, кнопки панелей инструментов и командные клавиши (accelerator keys). Обработка команд осуществляется способом, отличающимся от обработки других сообщений, и может производиться множеством объектов, включающим документы, шаблоны документов и сам объект "приложение".

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. Например, можно иметь одну функцию-обработчик для сообщения WM_PAINT, другую — для сообщения WM_KEYUP и третью — для сообщения WM_LBUTTONDOWN. 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 из-

меняются от того, к какой категории относится сообщение, однако механизм их вызова различен. Но прежде чем рассматривать механизмы посылки 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 при поступлении сообщения в оконную процедуру соответствующий обработчик вызывается автоматически. Реализация такой взаимосвязи основана на понятии Карты (или Таблицы) сообщений (message map).

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, командных клавиш клавиатуры в функции соответствующих классов, которые их обрабатывают. Эта особенность карты сообщений 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 для того, чтобы 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 используются 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
{
// объявления компонентов класса
protected:
  //{{AFX_MSG_MAP(TheClass)
  afx_msg void OnPaint();
  //}}AFX_MSG_MAP,
DECLARE_MESSAGE_MAP()
};
```

Он используется для объявления трех специальных компонентов класса, о которых нужно знать, но которые практически никогда непосредственно не используются.

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

```
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()
```

**Примечание**

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

Как видите, начинается карта 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_), записанное строчными буквами. 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, своих для каждого конкретного сообщения (например, *nFlags* и *point* — для сообщения WM_LBUTTONUP), отвечает соответствующий макрос. Описание всех обработчиков стандартных сообщений Windows можно найти в файле <afxwin.h> в объявлении класса *CWnd*.

Командные сообщения Windows от меню, командных клавиш и кнопок панелей инструментов обрабатываются макросом:

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

Этот макрос в качестве параметров использует 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 случаях обработчик команды — *memberFn* — будет вызван, только если в оконную процедуру поступит сообщение, которое в параметре *wParam* содержит идентификатор команды, совпадающий с *id*.

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

Для обработки извещений (notification message) от элементов управления применяется макрос

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

Этот макрос в качестве параметров использует код извещения (notification code) *wNotifyCode* от элемента управления, идентификатор элемента управ-

ления *id* и произвольное имя обработчика команды *memberFn*, прототип которой 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 *wNotify* от элемента управления (например, от BN_CLICKED) совпадает с кодом извещения, определенным в компоненте карты сообщений, а значение параметра *id* совпадает с идентификатором элемента управления.

Помимо рассмотренного макроса, общего для всех элементов управления и дочерних окон, для кнопок (BUTTON), элементов редактирования (EDIT), списка (LISTBOX) и комбинированного списка (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>. Их принцип действия полностью совпадает с принципом действия макроса ON_CONTROL.

С появлением в системах Windows 95 и Windows NT 4.0 более сложных дополнительных элементов управления (common controls) имеющихся макросов оказалось недостаточно, т. к. для этих элементов вместе с сообщением зачастую должны быть посланы и 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)
```

Как легко видеть, 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, куда должно быть записано возвращаемое значение.

Ниже приведен список извещений, общих для всех дополнительных элементов управления:

| | |
|---|---|
| NM_CLICK | Пользователь нажал левую кнопку мыши в элементе управления |
| NM_DBLCLK | Пользователь дважды щелкнул левой кнопкой мыши в элементе управления |
| NM_RCLICK | Пользователь нажал правую кнопку мыши в элементе управления |
| NM_RDBLCLK | Пользователь дважды щелкнул правой кнопкой мыши в 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 |

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 в качестве параметров использует два идентификатора *idFirst* и *idLast*, которые составляют непрерывный ряд. Обработчик *memberFn* будет вызываться всякий раз при поступлении в оконную процедуру сообщения, содержащего в параметре *wParam* идентификатор команды, границы которого определяются параметрами *idFirst* и *idLast*. Прототип обработчика должен иметь вид:

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

для первого и

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

для двух последних макросов.

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 также 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 обработчика сообщения — *OnNameMsg*. Очевидно, что необходимо определить как номер сообщения, так и его функцию обработки:

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

Понятно, 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 API *::RegisterWindowMessage*, описанную в этой главе ниже.

Теперь, познакомившись 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 еще раз внимательно посмотреть на иерархию классов библиотеки MFC, приведенную в главе 1, то можно заметить, что все классы, которые могут получать сообщения, являются производными от класса *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. Из всех компонентов этого класса для нас интерес представляет один:

```
virtual BOOL CCmdTarget::OnCmdMsg (
  UINT nID,
  int nCode,
  void* pExtra,
  AFX_CMDHANDLERINFO *pHandlerInfo)
```

Функция устанавливает маршрут и распределяет командные сообщения. Если посмотреть 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 принимает: *nID* — идентификатор команды; *nCode* — код извещения; *pExtra* — определяется значением *nCode*; *pHandlerInfo* — обычно равен NULL, в противном случае функция сама заполняет элементы *pTarget* и *pmf* структуры *pHandlerInfo*.

Если функция нашла адресата, который обработает сообщение, то она возвращает TRUE, в противном случае FALSE.

Таким образом, основным назначением *OnCmdMsg* является распределение команды по другим объектам. В принципе, эту функцию можно переопределить, однако при этом не следует забывать, что на вас перекладывается задача указания следующего получателя команды, чтобы не прервалась цепочка передачи сообщений.

Чтобы продемонстрировать практические аспекты обработки сообщений Windows, вернемся немного назад, в предыдущую главу, и покажем описанный механизм "в действии" на примере сообщения WM_CREATE.

Как вы помните, при создании главного окна приложения Styles необходимо было создать и обработчик сообщения WM_CREATE, в котором формировались дочерние и всплывающие окна, а также устанавливались их параметры. Последовательность действий при этом следующая:

1. Откройте окно мастера ClassWizard и в списке Messages найдите требуемое сообщение — в данном случае это WM_CREATE. Двойной щелчок левой кнопкой мыши приведет к созданию соответствующего обработчика сообщений (рис. 5.1).
2. Нажмите кнопку OK и приступайте к редактированию кода.
3. Обратите внимание на то, что описанные нами макросы уже автоматически добавлены в описание и реализацию класса:

```
// Файл MainFrm.h
class CMainFrame : public CFrameWnd
{
...
// Generated message map functions
protected:
```

```
  //{{AFX_MSG(CMainFrame)
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()
};

// Файл MainFrm.cpp
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
  //{{AFX_MSG_MAP(CMainFrame)
    ON_WM_CREATE()
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

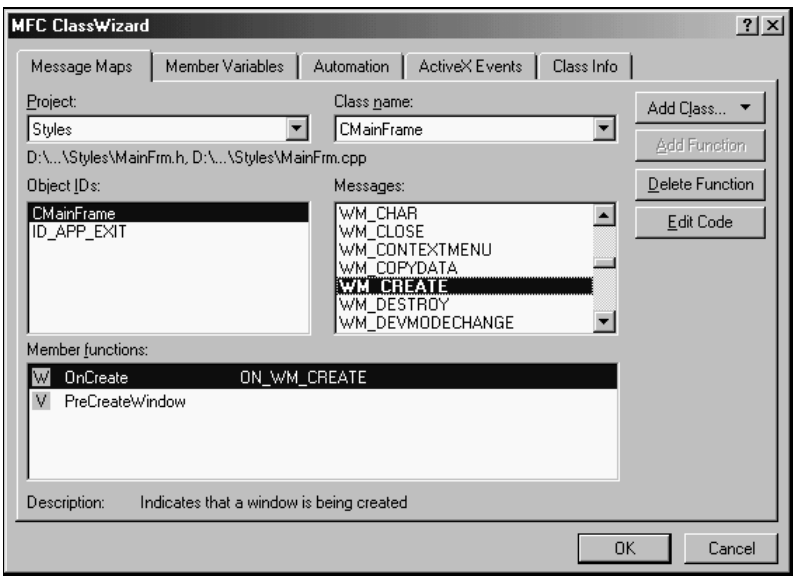



**Рис. 5.1.** Создаем обработчик сообщения WM_CREATE

**Примечание**

Комментарии на английском языке я оставил без перевода, чтобы лишний раз подчеркнуть, что они добавлены не мной, а мастером ClassWizard.

Теперь, после этого небольшого практического отступления от теоретических положений, займемся рассмотрением достаточно важного вопроса — каким образом осуществляется поиск обработчика команды.

## Стандартный маршрут команды

Когда *Run* — функция класса *CwinThread* — получает сообщение, она направляет его в соответствующее окно, являющееся получателем, в котором

сообщение будет обрабатываться (вами или самой библиотекой — сейчас неважно). Несколько иначе все выглядит для команд. Поскольку они формируются в результате взаимодействия пользователя с программой, 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?" Ну 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 либо будет выполнена, либо будет отправлена оконной процедуре, действующей по умолчанию.

Порядок, в котором адресаты передают команды, "зашит" в библиотеке классов MFC и представляет собой следующую последовательность.

- Команду получает фрейм MDI (объект класса *CMDIFrameWnd* или производного от него). Порядок обработки команды:
  - активное окно класса *CMDIChildWnd*;
  - сам объект класса *CMDIFrameWnd*;
  - приложение-объект класса *CWinApp* или производного от него.
- Команду получает фрейм документа (*CFrameWnd*, *CMDIChildWnd* или производные от них). Порядок обработки команды:
  - активное представление (view);
  - текущий фрейм документа;
  - приложение-объект класса *CWinApp* или производного от него.
- Команду получает представление (view). Порядок обработки команды:
  - текущее представление;
  - документ, присоединенный к представлению.
- Команду получает документ. Порядок обработки команды:
  - текущий документ;
  - шаблон документа, присоединенный к нему.
- Команду получает блок диалога. Порядок обработки команды:
  - текущий блок диалога;
  - окно, которое владеет этим диалогом;
  - приложение-объект класса *CWinApp* или производного от него.

Очевидно, что представленные здесь переходы взаимозависимы. Например, если команду получает документ, присоединенный к текущему представлению (view), то следующим объектом, которому будет передана команда, является текущий документ, а затем его шаблон.

**Примечание**

Пусть вас не смущает, что многие перечисленные объекты мы еще не рассматривали и они вам могут быть незнакомы. Каждый из них мы рассмотрим подробно, а пока познакомьтесь с общими принципами, просто приняв их на веру.

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 (view). 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 документа, то команда была бы направлена в шаблон, и в случае неудачи поиск продолжился в объекте класса *CMDIFrameWnd*, а затем в объекте класса приложения. Наглядно описанная последовательность действий представлена на рис. 5.2. Для команды, совершающей этот или любой другой маршрут, вызывается метод *OnCmdMsg* следующего объекта.

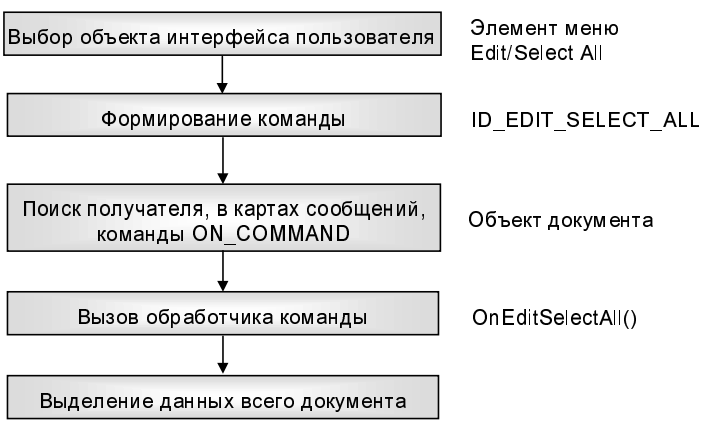



**Рис. 5.2.** Пример обработки команды ID_EDIT_SELECT_ALL

В отличие от команд, стандартные сообщения Windows не используют какого-либо маршрута для поиска обработчика, т. к. они посылаются непосредственно в оконную процедуру того объекта, который должен соответствующим образом отреагировать на них. Но и в этом случае каждый оконный класс, образованный непосредственно или косвенно из класса *CWnd*, имеет собственную карту сообщений, которая устанавливает связи между сообщениями 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, определяемых для стандартных сообщений Windows в классе *CWnd*.

## Команды обновления и класс *CCmdUI*

Когда пользователь раскрывает меню, каждый его элемент "должен знать", как он будет отображаться — доступным, отмеченным или нет. Получатель команды предоставляет ему эту информацию, реализуя обработчик сообщения ON_UPDATE_COMMAND_UI, внутри которого используется специальный класс *CCmdUI*, определяющий или 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 изменить их внешний вид. Маршрут поиска обработчика совпадает с тем, который действует для "обычных" командных сообщений (рис. 5.3).

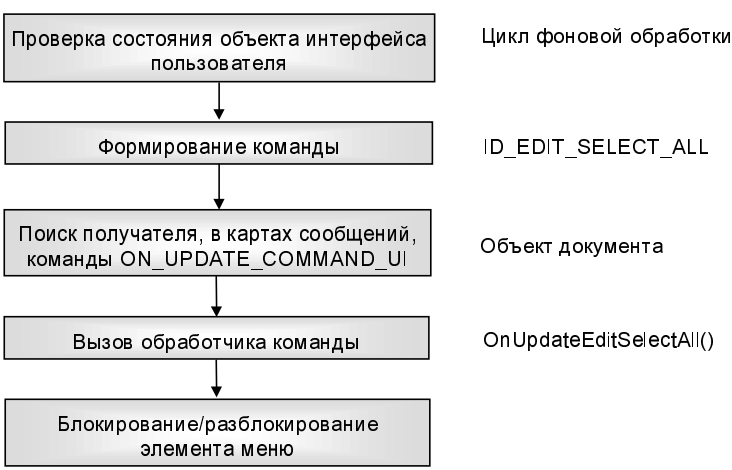



**Рис. 5.3.** Пример обработки команды обновления для ID_EDIT_SELECT_ALL

Рассмотрим возможности, заложенные в классе *CCmdUI* для обновления элементов интерфейса пользователя, поскольку многими из них нам при-

дется активно пользоваться буквально в следующей главе. Начнем с элементов данных 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 (notification), например, панель 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 методов.

`virtual void CCmdUI::`**`Enable`** `(BOOL bOn = TRUE)`

Блокирует (bOn равно FALSE) или разблокирует (bOn равно 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 передаваемого параметра: 0 — не помечено, 1 — помечено и 2 — неопределенное состояние. Последнее состояние применимо только к кнопкам панелей инструментов.

`virtual void CCmdUI::`**`SetRadio`** `(BOOL bOn = TRUE)`

Блокирует (bOn равно FALSE) или разблокирует (bOn равно TRUE) группу переключателей.

`virtual void CCmdUI::`**`SetText`** `(LPCTSTR lpszText)`

Устанавливает текст для этого 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.

Как я уже сказал, практические аспекты использования перечисленных здесь возможностей будут подробно рассмотрены в следующих главах при обсуждении вопросов, связанных с организацией интерфейса пользователя.

В заключение общего обзора принципов работы с сообщениями, заложенными в библиотеке классов MFC, и несколько забегая вперед, рассмотрим основные функции, которые для этого предназначены.

# Функции для работы с сообщениями

Понимание работы с сообщениями крайне важно для написания правильных и эффективных программ для Windows. Поэтому рассмотрим те 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 можно извлечь 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. Зарегистриро-

ванное 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. 5.4).

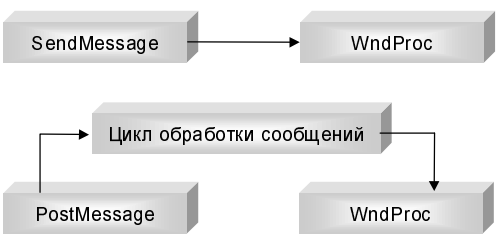



**Рис. 5.4.** Различие между посылкой сообщений функциями *SendMessage* и *PostMessage*

Основная цель этих функций — послать сообщение в окно своего или некоторого другого приложения. Но на этом их совпадения заканчиваются. Дело в том, что функция *SendMessage* явно вызывает процедуру того окна, которое ассоциировано с вызывающим ее объектом 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*. Выполнение самой функции *PostMessage* при этом завершается, и работа приложения может быть продолжена.

Но от теории, пусть даже и очень важной, вернемся к практике — пора развить наше приложение, добавив в него механизмы "общения с пользователем".

# ГЛАВА 6



# Каждый должен заниматься своим делом…

Никогда не удается делать что-то одно.
*Закон Хардина*

Теперь, когда мы познакомились с общей структурой приложений, написанных с применением библиотеки классов MFC, и рассмотрели типы окон, используемых в Windows, попытаемся разделить задачи, за которые должен отвечать некоторый отдельный объект. Что я имею в виду? На данном этапе знакомства с программированием под Windows мы можем выделить три задачи, присутствующие в любом приложении:

1. Взаимодействие с операционной системой.
2. Организация взаимодействия с пользователем.
3. Представление информации пользователю.

Для решения первой задачи в библиотеке реализован класс *CWinApp*, который настолько успешно справляется с возложенной на него задачей, что для его использования не требуется никаких дополнительных усилий.

Взаимодействие с пользователем — это основная задача подавляющего большинства приложений. Более того, сама система Windows задумывалась и реализовывалась именно для упрощения этого процесса. В библиотеке за эту задачу отвечают классы фреймов, базирующиеся на *CFrameWnd*.

В принципе, даже если ограничиться только этими двумя составляющими, можно писать достаточно сложные и разнообразные приложения. Однако по мере развития такого приложения разбираться в нем будет все труднее и труднее. Представьте себе, например, летчика пассажирского самолета, который помимо выполнения своих основных функций будет еще отвечать за обслуживание пассажиров. Думаю, далеко не каждый захочет лететь на таком самолете.

Именно по этой причине я и разделил задачи взаимодействия с пользователем и представления ему информации. Необходимо создать еще один "оконный" класс, подчинив его основному фрейму. Этот класс будет отвечать за некоторое дочернее окно, в которое мы в дальнейшем будем направлять весь вывод необходимой пользователю информации.

Итак, чтобы попутно научиться ценить свое время, откроем проект MyWizard3, имеющийся на прилагаемой дискете, и скомпилируем его. Обратите внимание на сообщение компилятора, которое он выводит в окне Output после построения приложения

```
Copying custom AppWizard to Template directory...
        1 file(s) copied
// Копирование настраиваемого мастера в каталог шаблонов ...
//       скопирован 1 файл
```

Как вы уже догадались — только что мы создали свой новый Wizard, на основе которого и будем создавать наше следующее приложение. Теперь вам не придется вручную набивать минимально необходимые тексты или "выкидывать" лишнее из предлагаемых Visual C++ проектов.

**Примечание**

Если вы пользуетесь версией Visual C++ 6.0, то там предусмотрено создание проекта, аналогичного нашему. Однако оно, с одной стороны, более развернутое (в частности, есть меню и блок диалога), а с другой — несколько ограниченнее: чтобы заменить курсор и/или пиктограмму необходимо добавить свой код. Так что выбирайте сами.

Теперь вам осталось только выбрать MyWizard3 в качестве мастера для создания каркас и указать имя приложения. Вот как это выглядит у меня (рис. 6.1).

Два раза нажмите кнопку OK, и каркас проекта готов (рис. 6.2).

По сравнению с нашей первой программой здесь появился класс *CChildWnd*. Именно он и будет отвечать за поддержку решения третьей задачи, о которой мы говорили в начале главы. Изменения в классах *CClockApp* (отвечает за взаимодействие с системой) и *CMainFrame* (как обычно, должен отвечать за взаимодействие с пользователем) минимальные, поэтому сразу переходим к новому для нас классу — *CChildWnd*.

Уже по названию можно догадаться, что объект этого класса представляет в приложении дочернее окно. Вот его определение, сгенерированное AppWizard:

```
class CChildWnd : public CWnd
{
// Construction
public:
  CChildWnd();
```

```
// Attributes
public:
// Operations
public:
// 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)
// NOTE - the ClassWizard will add and remove member functions here.
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
};
```

Как видите, мастер упростил использование этого класса, вставив в него необходимые макросы. Но больше ничего необычного здесь нет — конст-

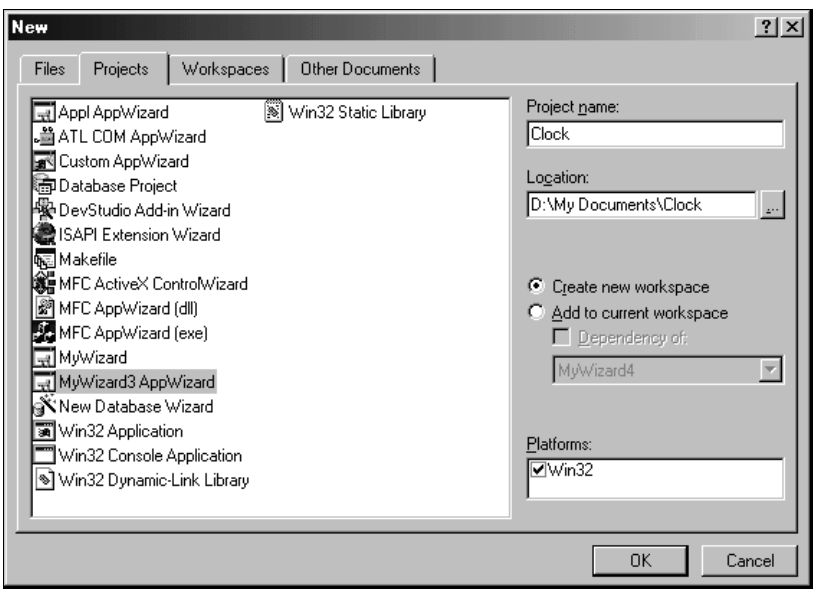



**Рис. 6.1.** Создаем проект Clock, воспользовавшись вновь созданным мастером MyWizard3

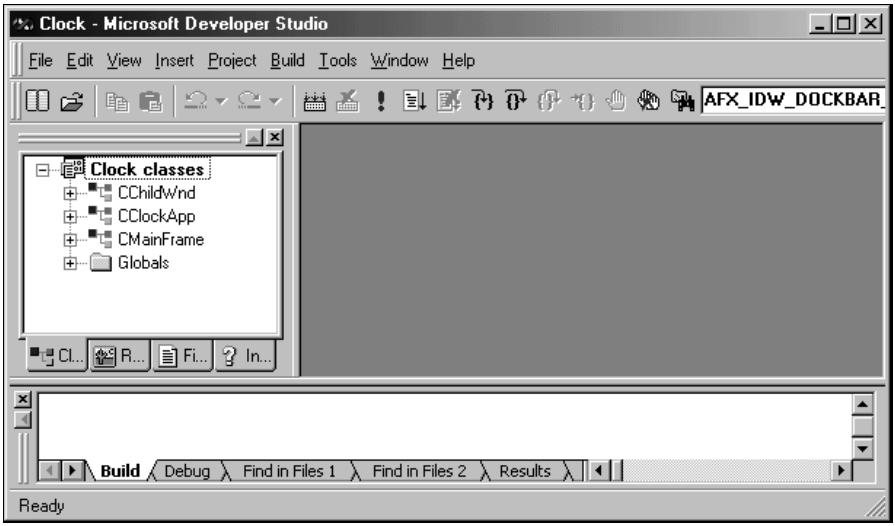



**Рис. 6.2.** Теперь можно наполнять уже работающее приложение "своим" содержимым

руктор, деструктор и переопределенная виртуальная функция *PreCreateWindow*. Обратите также внимание, что в качестве базового мы используем класс *CWnd*, практически идеально подготовленный для создания дочерних окон.

Пока ничего нового, по сравнению с предыдущими примерами, мы не видим. Теперь вопрос: "Когда будет вызвана эта переопределенная версия функции?" Правильно, после того, как мы создадим объект класса и вызовем одну из имеющихся функций создания окна Windows. В главе 4 мы уже обсуждали эти вопросы, и поэтому приведу только соответствующие фрагменты кода.

```
class CMainFrame : public CFrameWnd
{
...
protected:  // этот фрейм владеет дочерним окном
  CChildWnd   m_wndChild;
...
};

int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
  if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
    return -1;
  // Создаем дочернее окно Windows
  // в качестве клиентской области фрейма
  if (!m_wndChild.Create(NULL,
                         NULL,
```

```
                        AFX_WS_DEFAULT_VIEW,
                        CRect(0, 0, 0, 0),
                        this,
                        AFX_IDW_PANE_FIRST,
                        NULL))
  {
    TRACE0("Ошибка при создании дочернего окна \n");
    return -1;
  }
  return 0;
}
```

Что здесь для нас интересного? Конечно же, это параметры функции *CWnd::Create*, которые мы использовали:

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

Параметры функции нам уже практически знакомы, поэтому только перечислю их: *lpszClassName* — имя класса окна Windows; *lpszClassName* — заголовок окна; *dwStyle* — стиль окна; *rect* — координаты и положение окна на экране; *pParentWnd* — указатель на родительское окно; *nID* — идентификатор окна; *pContext* — некоторый контекст, связанный с окном.

А вот как это выглядит у нас:

```
m_wndChild.Create(
  NULL,                 // Нам пока не нужен специальный класс
  NULL,                 // Согласитесь, что окно,
                        // представляющее клиентскую область,
                        // будет странно выглядеть с заголовком
  AFX_WS_DEFAULT_VIEW,  // Используем специальную константу
                        // для создания клиентской области окна
  CRect(0, 0, 0, 0),    // Размерами займемся позже
  this,                 // Родителем является фрейм,
                        // в котором окно создается
  AFX_IDW_PANE_FIRST,   // Еще одна предопределенная константа для
                        // идентификации того, что это окно первое
  NULL)                 // Не связываем окно ни с каким контекстом
```

Другими словами, при создании клиентской области окна нам не надо практически ни о чем задумываться — главное не ошибиться в числе пара-

метров. Основную работу сделает сам класс *CChildWnd*. Посмотрите, как выглядит реализация этого класса:

```
//////////////////////////////////////////////////////////////////////
CChildWnd
// Пустые конструктор и ...
CChildWnd::CChildWnd()
{
}
// деструктор
CChildWnd::~CChildWnd()
{
}
// Операторные скобки, куда ClassWizard будет вставлять макросы
BEGIN_MESSAGE_MAP(CChildWnd, CWnd)
  //{{AFX_MSG_MAP(CChildWnd)
  // NOTE — the ClassWizard will add and remove mapping macros here.
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()

//////////////////////////////////////////////////////////////////////
CChildWnd message handlers — обработчики сообщений класса

// Единственная виртуальная функция, которую мы переопределили
BOOL CChildWnd::PreCreateWindow(CREATESTRUCT& cs)
{
  // Что творится в недрах библиотеки, зачастую неважно, однако
  // надо предоставить ей возможность выполнить свою часть работы
  if (!CWnd::PreCreateWindow(cs))
    return FALSE;
  // Теперь можно изменить стиль, ...
  cs.dwExStyle |= WS_EX_CLIENTEDGE;
  cs.style &= ~WS_BORDER;

  // а также класс окна,
  // что позволит определить свои цвет фона, курсор и пиктограмму
  cs.lpszClass = AfxRegisterWndClass(
                    CS_HREDRAW | CS_HREDRAW | CS_DBLCLKS,
                    AfxGetApp()->LoadCursor(IDR_MAINFRAME),
                    (HBRUSH)(COLOR_WINDOW + 1),
                    AfxGetApp()->LoadIcon(IDR_MAINFRAME));
  return TRUE;
}
```

**Примечание**

Пиктограмма и курсор для вас уже созданы (посмотрите в ресурсах), но вы с легкостью можете изменить их внешний вид.

Может возникнуть вопрос: "Почему мы стали проделывать все эти изменения здесь, а не установили все необходимые стили при создании дочернего окна?" Однозначный ответ дать невозможно, но мне представляется, что "каждый должен заниматься своим делом": фрейм создает себе клиентскую область, а она уже "сама решает, как ей выглядеть".

Итак, мы получили некоторый каркас, в котором три объекта отвечают каждый за свою часть работы. Единственное, что я бы изменил в этом каркасе — это заголовок основного окна приложения, чтобы он более соответствовал назначению программы. Посмотрите на рис. 6.3 и внесите необходимые изменения в код, чтобы заголовок выглядел аналогичным образом (название выбрано именно таким, поскольку на базе этого каркаса мы будем с вами создавать аналоговые часы).

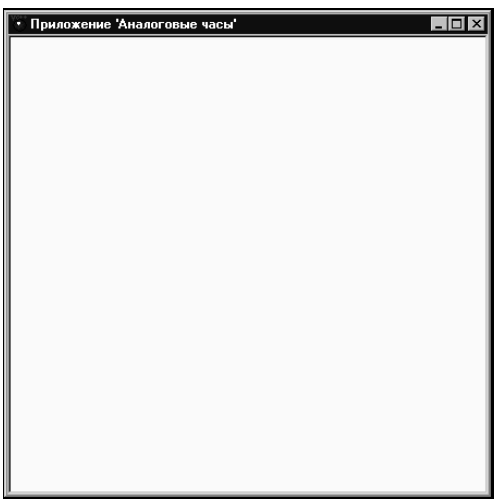



**Рис. 6.3.** На основе этого каркаса мы будем создавать аналоговые часы



# Часть III

## MFC и графический вывод


**Глава 7.** Управление графическим выводом

**Глава 8.** Кисти, карандаши и многое другое...

**Глава 9.** Класс поддержки рисования

**Глава 10.** Рисуем графические объекты

# ГЛАВА 7



# Управление графическим выводом

Мастерство — это когда "что" и "как"
приходят одновременно.
*NN*

Мы наконец-то подошли к тому моменту, когда пора показать что-нибудь пользователю — рассматривать стандартное окно Windows, пусть даже и созданное самим, уже не интересно. Начать новую тему, конечно же, следует с изложения общих представлений о том, что и как должно работать.

## Идеология графического вывода

Система Windows создавалась прежде всего как графическая оболочка (это в последнее время начали говорить о ней как о полноценной операционной системе). Соответственно, главное ее отличие (по крайней мере на первый взгляд) от MS-DOS — это графическое представление информации.

**Примечание**

У меня нет полной уверенности в том, что надо продолжать сравнивать, как это делается уже много лет, MS-DOS и Windows. Однако такое сопоставление все же более понятно, чем сравнение вывода информации в консольных и полноценных приложениях Windows.

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. Windows обеспечивает универсальность представления информации как на экране, так и на других устройствах вывода, на-

пример на принтере. Причем, что чрезвычайно важно, используются для этого 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 в качестве первичного устройства вывода приложения используют 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 логическое устройство вывода, что исключает одновременный доступ к последнему — перед началом процесса вывода приложение изменяет параметры вывода, подстраивая их под себя.

# Аппаратно-независимый графический вывод

Как вы уже поняли, одна из главных особенностей Windows — независимость графического вывода от устройства. Программное обеспечение, которое поддерживает независимость, содержится в двух библиотеках динамической компоновки (DLL, Dynamic Link Library). Первая, GDI.DLL, обеспечивает графический интерфейс устройства (Graphics Device Interface), а вторая, зависящая от конкретного устройства, является драйвером этого устройства. Например, если приложение осуществляет вывод в область пользовательского окна на дисплее VGA, это библиотека VGA.DLL, а если на принтер Epson FX-80, то — EPSON9.DLL.

Перед операцией вывода на некоторое устройство приложение должно запросить GDI о загрузке соответствующего драйвера (обычно это происходит автоматически и не требует от программиста дополнительных действий). 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. Система Windows обеспечивает хранение этих и других параметров в специальной структуре, называемой *контекстом устройства*.

# Контексты устройств

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 образы 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.

Сразу же необходимо подчеркнуть — приложение не имеет прямого доступа к контексту устройства, как это было для большинства структур Win32 API. Настройка параметров осуществляется посредством вызова специальных функций.

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.

В Windows предусмотрено несколько типов контекстов устройств, к знакомству с которыми мы и переходим.

**Примечание**

При дальнейшем изложении материала я часто буду пользоваться формулировками типа "Win32 API предоставляет" или "Win32 API обеспечивает". Несмотря на то, что предметом наших исследований является библиотека классов MFC, такие обороты не являются оговоркой — классы MFC, говоря откровенно, не более чем надстройка над Win32 API. Здесь же я должен еще раз оговориться — хотя и надстройка, но настолько профессионально сделанная, что зачастую об этом забываешь.

## Типы контекстов устройств

В Windows определены четыре основных 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.

**Примечание**

Когда мы говорим о рисовании на экране, то имеется в виду, конечно же, некоторое окно.

## Контексты экрана

Win32 API предоставляет три типа контекста устройства для экрана: *контекст класса*, *общий и частный контексты*. Контекст класса и частные контексты устройства используются в приложениях, выполняющих многочисленные операции рисования. Это, в частности, программы автоматизированного проектирования и настольные издательские системы, т. е. такие приложения, которые самостоятельно и постоянно осуществляют вывод и, соответственно, затрачиваемое на эти операции время критично с точки зрения производительности приложения. Общие контексты устройства используются в приложениях, лишь иногда выполняющих операции рисования, и где время, затрачиваемое на этот процесс, не столь существенно.

Приложение получает контекст устройства экрана, вызывая функцию *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 функцию *EndPaint* или *ReleaseDC*, соответственно.

### Контекст класса

Контекст класса поддерживается только для совместимости с предыдущими версиями Windows. При создании Win32-приложения вместо контекстов класса следует использовать частные контексты.

### Общий контекст

Общий контекст устройства — контекст устройства экрана, связанный с компонентом управления окна Win32 API и находящийся в области администратора окна. Приложение получает общий контекст устройства, вызывая функцию *GetDC*, *GetDCEx* или *BeginPaint*. Система Windows инициализирует общие контексты устройства значениями по умолчанию, которые можно менять по мере необходимости при помощи специальных функций. Количество общих контекстов в системе ограничено и, следовательно, после завершения операции вывода необходимо сообщить системе об этом, вызвав функцию *ReleaseDC* или *EndPaint*. В результате вызова одной из этих функ-

ций те изменения параметров контекста, которые были произведены приложением, будут отменены. Каждый раз параметры необходимо устанавливать именно таким образом.

### Частный контекст

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. Для получения частных контекстов окон, имеющих стиль CS_OWNDC, используются те же функции: *GetDC*, *GetDCEx* или *BeginPaint*.

## Контекст принтера

Win32 API предоставляет 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, исключения мигания, Win32 API предоставляет возможность приложениям полностью подготавливать изображение в памяти до его вывода на экран дисплея и только затем осуществлять непосредственно 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 (или 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 работу 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 для восстановления или получения заданных по умолчанию параметров 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 объекты

Система 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;
- регионы (regions) — области окна, которые могут быть ограничены прямоугольником, многоугольником, эллипсом или их комбинацией для выполнения операций заполнения, заливки, инверсии и т. п.; кроме того, служат для определения местоположения курсора;
- логические палитры (logical palettes) — для осуществления интерфейса между приложением и цветным устройством вывода, таким как дисплей; содержат список цветов, необходимых приложению;
- контуры (paths) — для целей заполнения или выделения контура различных фигур;
- шрифты (fonts) — для задания параметров вывода текста, включая имя шрифта, размер символов и т. п.

## Графические режимы

Система 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 (pens), кистей (brushes), текстовых объектов и растровых изображений с цветом фона поля вывода (окна).

**Режим масштабирования.** Определяет преобразования логических координат при графическом выводе в окна или на 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. Шрифты TrueType

Несмотря на то, что шрифты являются графическими объектами, их применение имеет свои существенные особенности. Именно поэтому их следует рассмотреть отдельно. Операционная система Windows может работать со шрифтами следующих трех типов:

- Растровые
- Векторные
- Шрифты TrueType

Отличие между видами 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 используется при уменьшении или увеличении масштаба отображения.

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

Система Windows обеспечивает разработчиков широким набором функций для шрифтового оформления своих приложений, начиная с того, что каж-

дый 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.

Теперь, когда мы в общих чертах познакомились с графическим выводом, самое время вернуться к библиотеке MFC и рассмотреть те же самые вопросы с этой точки зрения.

# Классы графического интерфейса

Ввиду важности вопроса еще раз напомню, что весь вывод в системе Windows реализован так, чтобы унифицировать работу с такими физически различными устройствами, как экран дисплея, принтеры, плоттеры и т. п. Для всех устройств вывода 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.

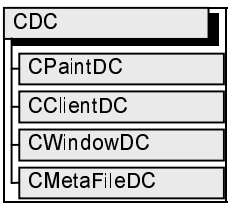
Win32 API имеет в своем составе большое количество функций графического вывода и настройки режимов рисования, однако их использование требует соблюдения определенного набора правил, т. к. в противном случае могут возникнуть ошибки во время выполнения программы, сбои при рисовании, потеря части содержимого окон и постепенное исчерпание системных ресурсов, что повлияет на работу не только самого неправильно созданного приложения, но и других приложений и всей системы в целом. Последняя неприятность сплошь и рядом случалась в 16-разрядных версиях Windows, где не освобожденные в явном виде графические ресурсы не освобождались и после завершения приложения. В 32-разрядных версиях, где учет и распределение ресурсов более строги, такое может произойти в случае, если неверно написанное приложение продолжает использоваться, однако после его завершения, 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 правиль-

ное использование ресурсов (захват и освобождение) в неявном для разработчика виде. С одной стороны, библиотека предоставляет программисту мощные средства графического вывода, но уже в виде функций соответствующих классов, причем имеются и такие, у которых нет аналогов в Win32 API (но это не означает, что в конечном итоге они обходятся без 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. Как и во многих других операциях, при выводе графики появилась возможность больше сконцентрироваться на самом процессе рисования, а не на подготовке к нему.

Перед тем как перейти к описанию тех возможностей, которые предоставляет библиотека классов MFC, напомню, 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.

## Классы контекстов устройств




Посмотрите внимательно на рис. 7.1, на котором представлены все классы библиотеки, имеющие отношение к контекстам устройств.

**Рис. 7.1.** Классы контекстов устройств библиотеки MFC

- *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*, вызываемой библиотечными функциями MFC.

- *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 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. Используются 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 возможен вывод их содержимого только с учетом настроек контекста того устройства, на которое осуществляется собственно вывод. Метафайлы второго типа (*контекстно-независимые*) 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 (от Windows MetaFile), а расширенных метафайлов — Enhanced MetaFiles — расширение .EMF.

**Примечание**

В отличие от класса *CpaintDC,* для объектов класса *CMetaFileDC* автоматический вызов функции *OnPrepareDC* не производится — ответственность за ее вызов лежит на разработчике приложения.

# Графические объекты

Настройка параметров графического вывода проводится двумя путями. Одни параметры настраиваются при помощи вызова функций Win32 API или функций класса *CDC* и производных от него, другие — путем замены так называемых *графических объектов*.

Библиотека MFC предоставляет разработчикам все необходимые классы, которые инкапсулируют соответствующие графические объекты Windows (рис. 7.2). Кроме того, библиотека имеет в своем составе дополнительные классы, не имеющие непосредственного отношения к графике, но значительно облегчающие решение ряда задач (классы *CPoint*, *CSize*, *CRect*, *CRectTracker*). Все классы имеют один и тот же базовый класс, который позволяет унифицировать работу с графическими объектами как таковыми (*CGdiObject*). Вот краткий обзор имеющихся классов.

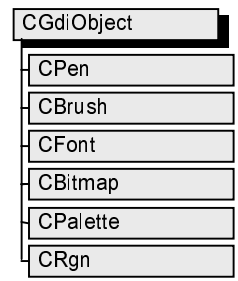



**Рис. 7.2.** Классы графических объектов библиотеки MFC

- *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 назначен для контекста устройства и использоваться для определения типа и цвета линий или границ фигур.
- *CBrush* — инкапсулирует объект Windows "кисть", который может быть назначен для контекста устройства и использоваться для определения типа и цвета заливки внутренних областей замкнутых фигур.
- *CFont* — инкапсулирует объект Windows "шрифт", который может быть назначен для контекста устройства и использоваться при операциях вывода текстовой информации.
- *CBitmap* — инкапсулирует объект Windows "битовый массив", который обеспечивает приложение рядом функций. 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 уст-

ройства 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. 7.1 показано соответствие представленных классов библиотеки MFC и типов дескрипторов объектов Windows.

***Таблица 7.1.*** *Соответствие дескрипторов объектов Windows и классов MFC*

| Класс MFC | Объект Windows | Тип дескриптора |
|---|---|---|
| CPen | "карандаш" | HPEN |
| CBrush | "кисть" | HBRUSH |
| CFont | "шрифт" | HFONT |
| CBitmap | "битовый массив" | HBITMAP |
| CPalette | "палитра" | HPALETTE |
| CRgn | "регион" | HRGN |

Помимо того, что каждый из вышеперечисленных классов имеет конструктор, что вполне естественно, в нем также определена специальная функция (функции) инициализации, имеющая префикс *Create*.

Создание графического объекта любого класса, например *CPen* или *CBrush*, возможно двумя способами. Первый способ заключается в использовании конструктора как для создания собственно объекта, так и для его инициализации. Второй способ, помимо использования конструктора, только создающего объект класса, дополнительно требует вызова инициализирующей функции.

Очевидно, что 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, своевременного их обнаружения, потребуется использовать механизм обработки исключений.

**Примечание**

Описание механизма обработки исключений выходит за рамки книги. Для тех же читателей, которые заинтересуются этим вопросом, могу порекомендовать книгу А. Мешкова и Ю. Тихомирова "Visual C++ и MFC. т. 3".

При использовании графического объекта для настройки соответствующих параметров рисования необходимо выполнить следующие действия:

- создать графический объект одним из двух описанных способов;
- заменить в контексте устройства вновь созданным графическим объектом текущий, сохранив при этом указатель на последний;

**Примечание**

Естественно, что при использовании функций Win32 API приложение манипулирует не указателями, а дескрипторами, не имеющими, в частности, деструктора. Освобождение ресурса необходимо проводить в явном виде вызовом соответствующей функции API.

- закончив операции рисования, восстановить графический объект в контексте устройства при помощи сохраненного указателя;
- обеспечить удаление созданного объекта по выходу из области видимости, что происходит автоматически для объектов, созданных в стеке приложения.

В случае, если один и тот же объект используется приложением достаточно часто, так что его создание и затем удаление существенно сказывается на производительности приложения в целом, возможно создание и сохранение такого объекта на все 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 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);
      // Рисование 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.

В этой главе я ничего не сказал о поддержке графических режимов библиотекой MFC, но к этому вопросу мы еще вернемся в соответствующем месте, а теперь пора переходить непосредственно к работе с представленными классами.



# ГЛАВА 8

# Кисти, карандаши и многое другое...

В проблеске гениальности видишь
свою бездарность.
*Станислав Ежи Лец*

Начиная с этой главы, мы будем заниматься разработкой приложения, задача которого — отображать текущее время в виде аналоговых часов (т. е. часов с циферблатом и стрелками). Естественно, что основная цель преследуется несколько иная, а именно — показать использование большей части того, что предоставляет библиотека классов MFC в плане работы с графикой. По этой причине многие фрагменты кода написаны "не оптимально" — но это не должно вас смущать: благодаря такому подходу мы рассмотрим различные варианты решения одной и той же задачи (где-нибудь это вам обязательно пригодится).

Основа для нашего приложения у вас уже есть — это пример из главы 5, который, как вы помните, создавался именно для этой цели. Однако все-таки начнем по порядку. Класс приложения *CClockApp* пока трогать не будем, поскольку то, что создал мастер, нас вполне устраивает — связь с системой установлена.

Таким образом, у нас есть два класса — *CMainFrame* и *CChildWnd*, каждый из которых отвечает за свою часть работы приложения: первый — за общий вид, а второй за то, что пользователь будет видеть в окне.

Займемся сначала общим видом.

## Изменение начального размера окна

Взгляните на следующий код, где полужирным шрифтом выделен фрагмент, который необходимо добавить к каркасу:

```
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
  if(!CFrameWnd::PreCreateWindow(cs))
    return FALSE;
  // Определяем размеры окна в зависимости от размеров экрана
  int nSize = min(GetSystemMetrics(SM_CXMAXIMIZED),
                  GetSystemMetrics(SM_CYMAXIMIZED)) * 3 / 5;
  // Задаем горизонтальный и вертикальный размеры окна
  cs.cx = nSize;
  cs.cy = nSize;

  cs.dwExStyle &= ~WS_EX_CLIENTEDGE;
  cs.lpszName  = "Приложение 'Кварцевые часы'";
  // Устанавливаем свою пиктограмму
  cs.lpszClass = AfxRegisterWndClass(
                       CS_HREDRAW | CS_HREDRAW | CS_DBLCLKS,
                       ::LoadCursor(NULL, IDC_ARROW),
                       HBRUSH (COLOR_WINDOW + 1),
                       AfxGetApp()->LoadIcon(IDR_MAINFRAME));
  return TRUE;
}
```

Как видите, сначала мы определяем ширину и высоту экрана, выбираем из них минимальное значение и устанавливаем размеры окна равными трем пятым этой величины.

Вот, собственно, пока и все, что необходимо добавить в код класса *CMainFrame*, поскольку никакого взаимодействия с пользователем в данном приложении не предполагается.

Для вывода надписей нам понадобятся шрифты, а для создания внешнего вида часов — кисти и карандаши. Чтобы не создавать их каждый раз при выводе рисунка (и, соответственно, не удалять после окончания вывода), сделаем это непосредственно в классе. В этом случае они будут освобождены при удалении объекта класса *CChildWnd*.

```
class CChildWnd : public CWnd
{
...
private:
  ...
  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;
...
};
```

Зачем нам так много графических объектов — три шрифта и по пять кистей и карандашей — мы разберемся по ходу создания приложения, а пока вспомним, что объекты Windows создаются в два этапа: сначала объект некоторого класса библиотеки MFC, а затем, вызовом соответствующей функции с префиксом Create, непосредственно объект Windows. Пока мы выполнили только первый этап — создали объекты соответствующих классов. Поэтому переходим ко второму этапу.

# Создание графических объектов Windows

Как вы думаете, где лучше всего реализовать действия второго этапа по созданию графических объектов Windows? Надеюсь, что вы догадались. Действительно, в обработчике сообщения WM_CREATE — функции *OnCreate*. Поэтому первое, что надо сделать — это определить обработчик *OnCreate*. Как это сделать, мы уже рассматривали. После этого можно переходить непосредственно к созданию графических объектов Windows. И начнем мы со шрифтов.

## Создание шрифтов

Для работы со шрифтами в библиотеке MFC реализован класс *CFont* (рис. 8.1), и для более ясного понимания дальнейших действий рассмотрим его несколько подробнее.

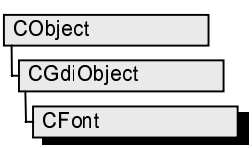



**Рис. 8.1.** Место класса *CFont* в иерархии библиотеки MFC

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 его в кон-

кретный 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. Именно с такого подхода мы и начнем. В данном случае это связано еще и с тем, что структура *LOGFONT*, используемая для создания шрифта, достаточно часто применяется не только вместе с классом *CFont*, но и, например, при выборе шрифта в блоке диалога, созданного при помощи класса *CFontDialog*.

Итак, для создания шрифта можно использовать функцию

```
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 *LOGFONT* 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 рассчитывается как разность между полной высотой знакоместа и размером области для отображения специальных знаков, например, ударения. Это поле структуры *LOGFONT* может интерпретироваться, в зависимости от знака, следующим образом:

| | |
|---|---|
| > 0 | Значение преобразуется в единицы устройства вывода и используется для выбора из имеющегося набора шрифта, соответствующего по высоте знакоместа |
| 0 | Для нахождения шрифта используется значение высоты по умолчанию |
| < 0 | Значение преобразуется в единицы устройства вывода и используется для выбора из имеющегося набора шрифта, соответствующего по высоте символа |

При поиске 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 задаваемого в точках (points) шрифта следующим образом:

```
lfHeight = -MulDiv(PointSize,dc.GetDeviceCaps(LOGPIXELSY),72);
```

**Примечание**

Режимы отображения мы рассмотрим несколько позже.

Поле *lfWidth* определяет среднюю ширину символов в логических единицах. Нулевое значение поля *lfWidth* обеспечивает выбор из имеющегося набора шрифта, 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.

К сожалению, в рамках Windows 95/98 значения обоих полей *lfEscapement* и *lfOrientation* должны быть равны, т. е. существует возможность вывода под наклоном только текста 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 задании нуля 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.

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.

При задании в явном виде имени шрифта особенно важно правильно задать таблицу кодировки, т. к. именно кодировка в **первую очередь определяет набор установленных шрифтов**, среди которых будет произведен поиск по имени (рис. 8.2).

| SYMBOL_CHARSET и "Times New Roman" | RUSSIAN_CHARSET и "Wingdings" |
|---|---|
| ?oU±?, Ul?l??C<br>?lol ??l?O?a?? | Текст, который<br>можно прочитать |

**Рис. 8.2.** 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 | Определяет, что при установке в системе нескольких шрифтов с одним и тем же названием выбирается системный шрифт |
| OUT_OUTLINE_PRECIS | Только для Windows NT: выбор осуществляется из набора векторных или шрифтов TrueType |
| OUT_RASTER_PRECIS | Определяет, что при установке в системе нескольких шрифтов с одним и тем же названием выбирается 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 шрифтов TrueType, если ни один такой шрифт не установлен, используется механизм выбора, принятый по умолчанию |
| OUT_TT_PRECIS | Определяет, что при установке в системе нескольких шрифтов с одним и тем же названием выбирается 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 шрифтов TrueType |
| CLIP_MASK | Не используется |
| CLIP_EMBEDDED | Указывается при использовании внедренных шрифтов |
| CLIP_LH_ANGLES | Указывает, что отсчет направления угла поворота (по часовой стрелке 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 несущественно |
| 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 наиболее подходящий по размеру шрифт; при необходимости возможно создание синтетических начертаний шрифтов — полужирного, курсивного, подчеркнутого и перечеркнутого |

Поле *lfPitchAndFamily* определяет тип и семейство шрифтов. Это поле позволяет выбрать наиболее похожий на заданный по имени шрифт 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*).

Таким образом, для создания шрифта, с помощью которого мы будем выводить название, нам необходимо заполнить поля структуры *LOGFONT*:

```
LOGFONT lf = {
  130,                          // Высота шрифта
                                // в логических единицах
```

```
  0,                              // В качестве ширины символов
                                  // выбирается значение, наиболее
                                  // подходящее конкретному устройству
  0,                              // Текст будем выводить
  0,                              // без наклона
  FW_BOLD,                        // полужирный,
  0,                              // не курсивный,
  0,                              // не подчеркнутый
  0,                              // и не перечеркнутый шрифт
  DEFAULT_CHARSET,                // Кодировка, установленная
                                  // по умолчанию
  OUT_DEFAULT_PRECIS,             // Механизм соответствия заданным
                                  // параметрам по умолчанию
  CLIP_DEFAULT_PRECIS,            // Механизм отсечения,
                                  // используемый по умолчанию
  PROOF_QUALITY,                  // Важно качество шрифта
  VARIABLE_PITCH | FF_DONTCARE,   // Пропорциональный шрифт, а какое
                                  // семейство — не имеет значения
  "Arial"                         // Имя шрифта
};
```

После того как структура заполнена, для создания соответствующего объекта Windows осталось только вызвать функцию *CreateFontIndirect*, передав ей в качестве параметра указатель на эту структуру.

```
m_Font_1.CreateFontIndirect(&lf);
```

Кроме этой функции, для создания шрифта можно воспользоваться функцией *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)
```

Воспользуемся этой функцией для создания второго шрифта:

```
m_Font_2.CreateFont(
  100,
  0,                              // В качестве ширины символов
                                  // выбирается значение, наиболее
                                  // подходящее конкретному устройству
  0,                              // Текст будем выводить
  0,                              // без наклона
  FW_BOLD,                        // полужирный,
  0,                              // не курсивный,
  0,                              // не подчеркнутый
  0,                              // и не перечеркнутый шрифт
  DEFAULT_CHARSET,                // Кодировка, установленная
                                  // по умолчанию
  OUT_DEFAULT_PRECIS,             // Механизм соответствия заданным
                                  // параметрам по умолчанию
  CLIP_DEFAULT_PRECIS,            // Механизм отсечения,
                                  // используемый по умолчанию
  PROOF_QUALITY,                  // Важно качество шрифта
  VARIABLE_PITCH | FF_SWISS,      // Пропорциональный шрифт
                                  // без засечек
  "Arial"                         // Имя шрифта
);
```

Как видите, особой разницы в рассмотренных способах создания шрифта нет. Однако в ряде случаев все же предпочтительнее пользоваться функцией *CreateFontIndirect* и вот почему.

Обратите внимание, что при создании двух шрифтов практически все их параметры совпадают. Конечно, это не обязательное условие, но в большинстве случаев именно так и бывает. Поэтому, чтобы не повторять каждый раз описание всех параметров, можно просто заменить значения в соответствующих полях уже созданной структуры LOGFONT и вызвать функцию *CreateFontIndirect*. Как вы помните, нам надо создать еще один, последний, шрифт. Воспользуемся для этого предложенным методом:

```
// Изменяем высоту
lf.lfHeight = 80;
// Делаем шрифт самым "тонким"
lf.lfWeight = FW_THIN;
// Устанавливаем пропорциональный шрифт без засечек
lf.lfPitchAndFamily = VARIABLE_PITCH | FF_SWISS;
```

Осталось только вызвать соответствующую функцию, что мы и делаем:

```
m_Font_3.CreateFontIndirect(&lf);
```

Большой разницы в рассмотренных двух способах нет — какой вам больше понравился, таким и пользуйтесь. Мне же осталось рассказать еще о двух функциях.

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 в пунктах (points). А пункт — это вовсе не экранная точка, т. е. пиксел, а единица длины, равная 1/72 дюйма. Класс *CFont* имеет в своем составе функцию *CreatePointFont*, 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)
```

Первый 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*, при задании нуля выбирается системно-независимый шрифт.

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 *CreateFontIndirect* — функция *CreatePointFontIndirect* — в качестве параметра также принимает указатель на структуру *LOGFONT*, однако единицами измерения в поле *lfHeight* этой структуры являются пункты.

Мы не воспользовались этими функциями в частности еще и потому, что у нас пока не определен контекст устройства.

Итак, мы подготовили шрифты и переходим к следующему необходимому нам объекту — кисти.

## Создание кистей

Процесс создания кисти значительно проще, чем создание шрифта. В библиотеке MFC реализован класс *CBrush* (рис. 8.3), который инкапсулирует графический объект Windows "кисть" для использования в качестве основы 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()*, который 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. Рассмотрим некоторые из этих функций более подробно.

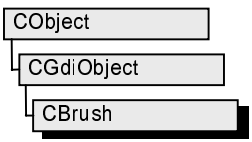



**Рис. 8.3.** Место класса *CBrush* в иерархии библиотеки MFC

```
BOOL CBrush::CreateSolidBrush(COLORREF crColor)
```

Используется для создания однотонной кисти, цвет которой задается параметром *crColor*. Тип COLORREF определяет 32-разрядное значение, задающее цвет в системе RGB (красный, зеленый, синий) и имеющее формат 0x00bbggrr.

Для задания параметра *crColor* можно воспользоваться определенным в Win32 API макросом RGB, который комбинирует интенсивности (в диапазоне от 0 до 255) трех цветов — красного, зеленого и синего и определяется следующим образом:

```
COLORREF RGB(int red, int green, int blue)
```

Итак, для создания однотонной кисти, которой мы будем закрашивать фон наших часов, нам необходимо задать ее цвет:

```
#define COLOR_BCKGR   RGB(224, 224, 224)
```

и вызвать функцию *CreateSolidBrush*:

```
m_wndBrush.CreateSolidBrush(COLOR_BCKGR);
```

Аналогичную операцию проделаем и для других кистей, за исключением одной:

```
#define COLOR_CLOCK   RGB(64, 128, 128)
#define COLOR_ROUND   RGB(64, 128, 128)
#define COLOR_CRNER   RGB(64, 128, 128)
...
m_clkBrush.CreateSolidBrush(COLOR_CLOCK);
m_rndBrush.CreateSolidBrush(COLOR_ROUND);
m_crnBrush.CreateSolidBrush(COLOR_CRNER);
```

Для фона стрелок наших часов используем кисть со штриховкой. Создать ее так же просто, как и однотонную кисть:

```
BOOL CBrush:: CreateHatchBrush(
  int nIndex,
  COLORREF crColor)
```

Здесь параметр *nIndex* определяет один из возможных вариантов штриховки:

| | |
|---|---|
| HS_BDIAGONAL | Штриховка слева направо под углом 45° и сверху вниз |
| HS_CROSS | Горизонтальная и вертикальная штриховка крест накрест |
| HS_DIAGCROSS | Штриховка крест накрест под углом 45° |
| HS_FDIAGONAL | Штриховка слева направо под углом 45° и снизу вверх |
| HS_HORIZONTAL | Горизонтальная штриховка |
| HS_VERTICAL | Вертикальная штриховка |

а *crColor* — значение цвета линий штриховки.

Вот как это будет в нашем случае:

```
m_arrBrush.CreateHatchBrush(
  HS_DIAGCROSS,              // Штриховка крест накрест под  углом 45°
  COLOR_ARROW);              // цвет линий штриховки
```

Кисть для фона реализуем с помощью специального шаблона. В этом случае необходимо сначала создать или загрузить битовый массив, а затем вызвать функцию

```
CBrush:: CreatePatternBrush (CBitmap* pBitmap)
```

В качестве параметра *pBitmap* функции передается указатель на объект класса *CBitmap*, представляющего в библиотеке объект Windows "битовый массив".

При использовании этой функции надо иметь в виду, что размеры битового массива должны быть не меньше чем 8×8 пикселов, а сама кисть создается на основе левой верхней квадратной области битового массива размером также 8 × 8 пикселов.

Я думаю, вам теперь не составит труда самостоятельно разобраться с остальными имеющимися в классе *CBrush* функциями, а нам надо прерваться, чтобы познакомиться с классом *CBitmap*.

## Создание битовых массивов

Для работы с битовыми массивами в библиотеке MFC реализован класс *CBitmap* (рис. 8.4), который инкапсулирует графический объект Windows "битовый массив".

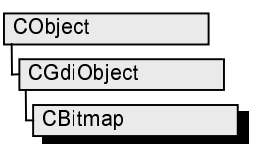



**Рис. 8.4.** Место класса *CBitmap* в иерархии библиотеки MFC

Как обычно, для работы с объектом класса необходимо создать сам объект, а затем создать собственно битовый массив Windows путем вызова одной из функций *CreateBitmap*, *CreateBitmapIndirect*, *CreateCompatibleBitmap* или *CreateDiscardableBitmap*, загрузить его из соответствующего ресурса (функции *LoadBitmap*, *LoadOEMBitmap*, *LoadMappedBitmap*) или установить связь с ранее созданным (загруженным) битовым массивом (функция *Attach*).

Кроме этого, класс предоставляет набор функций, позволяющих менять и узнавать содержимое (*SetBitmapBits* и *GetBitmapBits*) и размеры (*SetBitmapDimension* и *GetBitmapDimension*) массива.

Для нашего примера мы воспользуемся функцией загрузки битового массива, созданного в качестве ресурса:

```
BOOL CBitmap::LoadBitmap (LPCTSTR lpszResourceName)
```

или

```
BOOL CBitmap:: LoadBitmap (UINT nIDResource)
```

> Функции позволяют загрузить битовые массивы, заданные текстовой строкой (параметр *lpszResourceName*), т. е. по имени, или числовым идентификатором (параметр *nIDResource*). Загрузка битовых массивов может осуществляться прямо из ресурсов исполняемого модуля текущего приложения.

Очевидно, что сначала требуется создать соответствующий ресурс. Вот как он будет выглядеть у нас (рис. 8.5).

Затем ресурс надо загрузить в объект класса *CBitmap*:

```
CBitmap bmp;
bmp.LoadBitmap(IDB_BMP_BK);
```

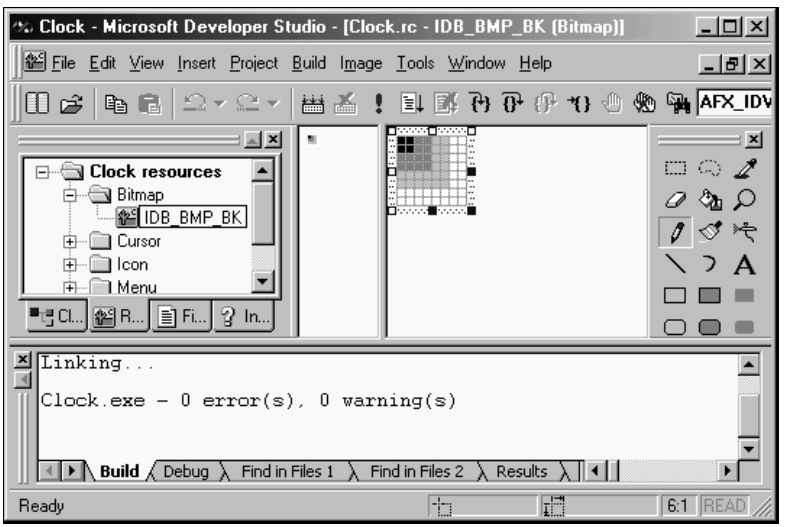



**Рис. 8.5.** На основе этого шаблона мы будем создавать "цвет фона" кисти часов

Конечно же, в классе *CBitmap* есть и другие функции, но рассмотренный подход — один из наиболее простых, и поэтому мы пока им и ограничимся — пора возвращаться к нашему примеру.

После того как объект класса *CBitmap* создан, нам осталось передать указатель на него в функцию *CreatePatternBrush*:

```
m_clkBrush.CreatePatternBrush(&bmp);
```

Результатом наших действий будет кисть на основе этого шаблона.

Теперь, когда мы научились создавать не только шрифты и кисти, но битовые массивы, перейдем к последнему (используемому нами, но не последнему из имеющихся) объекту, а именно, к карандашам.

## Создание карандашей

Класс *CPen* инкапсулирует графический объект Windows "карандаш" (рис. 8.6). Карандаши используются для рисования различных линий и контуров объектов. Эти объекты имеют наиболее богатые возможности для их создания — в Windows имеется целый набор различных предопределенных стилей, свойств и типов, которые позволяют использовать карандаши просто и эффективно и, соответственно, проводить с их помощью самые разнообразные линии.

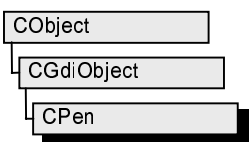



**Рис. 8.6.** Место класса *CPen* в иерархии библиотеки MFC

Как и все другие классы графических объектов, класс *CPen* допускает как создание объектов класса, не связывая их при этом с объектами Windows, так и одновременное создание объекта класса и соответствующего графического объекта.

Что касается последней возможности, то она дублируется наличием функций создания карандаша Windows уже после создания объекта класса. Поскольку мы уже создали объекты этого класса, то не будем рассматривать имеющиеся в классе конструкторы, а подробно остановимся только на специальных функциях создания соответствующих объектов Windows.

```
CPen::CreatePen (
    int nPenStyle,
    int nWidth,
    COLORREF crColor)
```

и

```
CPen::CreatePen (
    int nPenStyle,
```

```
    int nWidth,
    const LOGBRUSH* pLogBrush,
    int nStyleCount = 0,
    const DWORD* lpStyle = NULL)
```

Давайте рассмотрим отдельно для каждой версии функции *CreatePen* возможные значения параметра *nPenStyle*.

В результате использования первой версии функции создаются 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 | Штрихпунктир, толщина которого равна 1 |
| PS_DASHDOTDOT | Двойной штрихпунктир, 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 пиксела и имеющих цвет, недоступный для текущего устройства. В этом случае линии будут состоять из пикселов доступных цветов, совокупность которых наиболее близко соответствует заданному цвету |

Для наглядности продемонстрируем линии различных стилей. Обратите внимание, что при выводе линий одной длины некоторые штриховые линии фактически оканчиваются раньше сплошных (рис. 8.7), что вполне естественно — эти линии завершились в месте пустого пространства между штрихами. Такой эффект может негативно сказаться при решении конкретных задач.

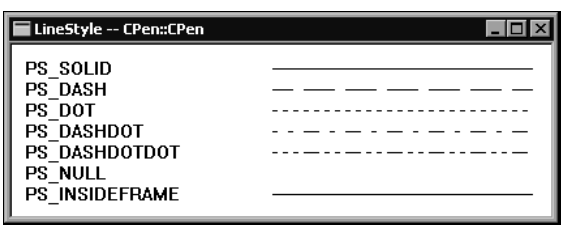



**Рис. 8.7.** Различные стили линий

Использование второй версии функции *CreatePen* позволяет комбинировать в параметре *nPenStyle* стиль, тип, режим рисования концов линии и режим

соединения линий. Каждый атрибут рисования может быть представлен только одним значением, заданные значения комбинируются при помощи оператора OR (|).

Карандаши Windows бывают двух типов — PS_GEOMETRIC и PS_COSMETIC, отличающихся использованием кистей для рисования линий и требованиями к заданию ширины. Кроме того, от выбранного типа зависит интерпретация параметров, задающих размеры штрихов для пользовательских стилей.

Вторая версия функции *CreatePen* поддерживает два дополнительных стиля — PS_ALTERNATE (может использоваться только совместно с типом PS_COSMETIC) и PS_USERSTYLE, который для рисования линий требует задания массива длин штрихов.

Для карандашей также возможно задание режима рисования концов (естественно, это имеет смысл при рисовании широких линий). Таких режимов три, для первых двух 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 |

Примеры различных типов, стилей рисования и соединения линий представлены на рис. 8.8. Одно из требований при применении режима рисования концов линий и соединений заключается в том, что рисование линий при помощи карандашей, созданных с флагами PS_ENDCAP_SQUARE, PS_ENDCAP_FLAT, PS_JOIN_BEVEL и PS_JOIN_MITER, можно осуществлять только при помощи механизма использования контуров. Обратите внимание, что для наглядности по оси каждой линии, имеющей тип PS_GEOMETRIC, проведены сплошные тонкие линии с теми же координатами.

При создании карандаша необходимо также задать его ширину с помощью параметра *nWidth*. В первой версии функции *CreatePen* заданием ширины, равной 0, гарантируется вывод линии шириной в один пиксел независимо от режима вывода (mapping mode). При использовании второй версии функ-

ции ширина задается в логических единицах — для карандашей типа PS_GEOMETRIC, а для карандашей типа PS_COSMETIC ширина должна быть равна 1.

Для задания цвета карандаша используется параметр *crColor*.

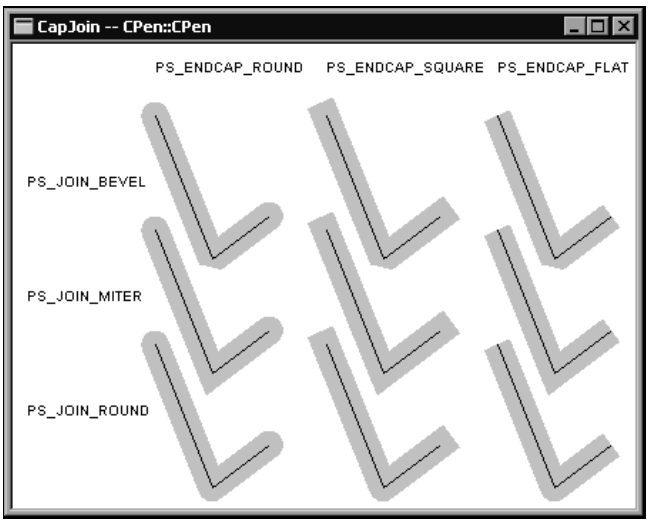



**Рис. 8.8.** Различные стили рисования и соединения линий

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 быть равен BS_SOLID.

Параметр *lpStyle* указывает на массив, содержащий *nStyleCount* чисел, определяющих попарно длину штриха и расстояние между штрихами для карандашей, имеющих стиль PS_USERSTYLE.

Кроме рассмотренных, для создания карандашей могут быть использованы функции *CreatePenIndirect* или *CreateStockObject*. При помощи первой из них возможно создание только обычных карандашей, т. е. таких же, какие можно создавать при помощи первой версии функции *CreatePen*.

Как и в случае создания кистей, никаких сложностей при создании карандашей нет. Вот как мы осуществляем это в нашем примере:

```
#define COLOR_BCKGR   RGB(224, 224, 224)
#define COLOR_BLACK   RGB(  0,   0,   0)
#define COLOR_ARROW   RGB(  0, 128, 128)
#define COLOR_CRNER   RGB( 64, 128, 128)
...
m_wndPen.CreatePen(PS_SOLID, 1, COLOR_BCKGR);
m_arrPen.CreatePen(PS_SOLID, 1, COLOR_ARROW);
```

```
m_clkPen.CreatePen(PS_SOLID, 1, COLOR_BLACK);
m_crnPen.CreatePen(PS_SOLID, 1, COLOR_CRNER);
```

Единственное исключение мы сделаем для карандаша, которым будем рисовать "метки" для обозначения цифр. В этом случае используем вторую версию функции *CreatePen*. Для нее, как вы помните, нам необходим указатель на структуру *LOGBRUSH*, которая содержит параметры кисти, применяемой при рисовании "толстых" линий. Посмотрите, как можно получить параметры кисти, используемой в качестве основы карандаша:

```
LOGBRUSH lb;
m_rndBrush.GetLogBrush(&lb);
```

После того как поля структуры заполнены, нам осталось только вызвать функцию создания карандаша, передав ей в качестве параметра указатель на эту структуру:

```
m_rndPen.CreatePen(
  PS_SOLID | PS_GEOMETRIC |  // "Сплошные",
  PS_ENDCAP_SQUARE,          // с прямоугольными концами линии
  20,                        // Ширина 20 пикселов
  &lb);                      // Параметры кисти
```

**Примечание**

Задание прямоугольных концов для линий, для которых создаем карандаш, в нашем примере не влияет на рисование, т. к. это свойство проявляется, когда рисование осуществляется при помощи создания контуров, а данный карандаш будет использоваться для рисования дуг, которые функцией *CDC::StrokePath* не отображаются.

Здесь мне хотелось бы рассмотреть еще один графический объект, который мы пока не создавали, но который понадобится нам при рассмотрении вопросов рисования. Речь идет о регионах.

## Работа с регионами

Класс *CRgn* инкапсулирует объект Windows "регион", представляющий собой совокупность прямоугольных и эллиптических областей (рис. 8.9).

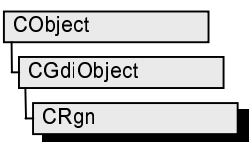



**Рис. 8.9.** Место класса *CRgn* в иерархии классов библиотеки MFC

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. 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 версия функции получает в качестве параметров координаты левого верхнего и правого нижнего углов прямоугольной области. Вторая версия функции получает указатель на структуру *RECT*, в которой указываются также координаты прямоугольной области.

## Создание эллиптических регионов

```
BOOL CRgn::CreateEllipticRgn (
    int x1,
    int y1,
    int x2,
    int y2)
```

и

```
BOOL CRgn::CreateEllipticRgnIndirect (LPCRECT lpRect)
```

Обе версии функции получают информацию, аналогично версиям предыдущей функции, о прямоугольной области, в которую вписан эллипс.

## Создание сложных регионов на базе многоугольников

```
BOOL CRgn::CreatePolygonRgn (
    LPPOINT lpPoints,
    int nCount,
    int nPolyFillMode)
```

Функция принимает в качестве информации о многоугольнике массив координат (*lpPoints*) и количество его вершин (*nCount*).

```
BOOL CRgn::CreatePolyPolygonRgn (
    LPPOINT lpPoints,
    LPINT lpPolyCounts,
    int nCount,
    int nPolyFillMode)
```

Функция принимает в качестве информации массив координат вершин многоугольников (*lpPoints*) и массив, содержащий количество вершин для каждого из них (*lpPolyCounts*). Параметр *nCount* задает количество многоугольников.

Параметр *nPolyFillMode* в этих функциях определяет режим закраски ALTERNATE или WINDING (см. следующую главу).

### 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 параметры *x3* и *y3* определяют 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)
```

Функция 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:

| | |
|---|---|
| RGN_AND | Регион будет являться пересечением обоих заданных, т. е. определяться 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 являться объединением обоих заданных, т. е. будет включать все области, относящиеся и к первому, и ко второму региону |
| RGN_XOR | Регион будет включать области первого региона (pRgn1) и второго региона (pRgn2), кроме тех, которые принадлежат и первому и второму одновременно |

Несмотря на то, что Windows для осуществления комбинирования требует наличия объектов для всех трех регионов, 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, упомянем еще несколько. Класс *CRgn* имеет в своем составе функцию копирования *CopyRgn*, функцию преобразования текущего контура (path) в регион (*CreateFromPath*), функцию сравнения двух регионов (*EqualRgn*).

Для иллюстрации использования регионов приведем процедуру, в которой в результате комбинирования трех регионов производится закрашивание части замкнутых областей одним вызовом функции рисования (рис. 8.10).

```
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
  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);
  // Восстанавливаем кисть
  dc.SelectObject(pBrush);
}
```

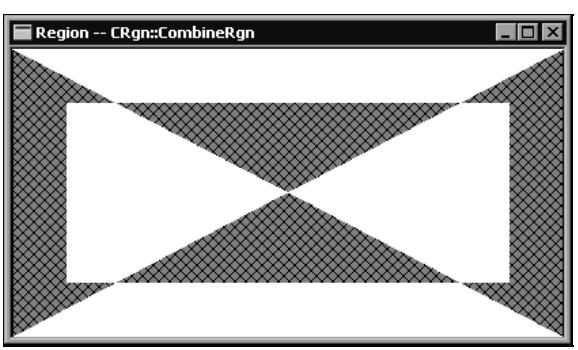



**Рис. 8.10.** Результат работы функции *CRegionFrame::OnPaint*

К сожалению, даже на таком простом примере при выполнении программы видно, как медленно происходит рисование: такое впечатление, что для каждой точки проверяется возможность ее закрашивания.

На этом подготовительные работы по созданию необходимых нам графических объектов закончены, и мы переходим к следующему вопросу.

# Изменение размеров окна

Вообще-то за изменение размеров окна в приложениях отвечает класс фрейма (у нас это *CMainFrame*). И более того, для поддержки этой возможности нам не пришлось писать ни строчки кода — все сделала библиотека MFC. Нас же вопрос изменения размеров интересует несколько с другой точки зрения: мы хотим, чтобы размер наших часов всегда совпадал с размерами окна.

Реализовать это чрезвычайно просто. Прежде всего необходимо определить две переменные, которые будут хранить текущие значения ширины и высоты окна:

```
class CChildWnd : public CWnd
{
...
private:
  int m_cxClient; // ширина клиентской области окна
  int m_cyClient; // высота клиентской области окна
...
};
```

А для отслеживания текущих значений ширины и высоты необходимо определить обработчик сообщения WM_SIZE. Вот как он будет выглядеть:

```
void CChildWnd::OnSize(UINT nType, int cx, int cy)
{
  // Не вмешиваемся в работу библиотеки
  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;
}
```

Вот собственно и все. Теперь, как только нам понадобятся параметры размеров часов, мы всегда можем получить их текущие значения.

# Обработка сообщения WM_PAINT

Многие из вас, конечно же, ожидали, что в конце концов мы придем к созданию обработчика сообщения WM_PAINT. И вы не ошиблись, хотя в нашем конкретном случае вполне можно было бы обойтись и без него. Дело в том, что рисование часов обладает особенностью — их надо перерисовывать не в зависимости от "поврежденности" окна, а тогда, когда истечет выбранный нами отрезок времени (через секунду, если будем рисовать секундную стрелку, или через минуту, если секундную стрелку рисовать не будем).

Однако для демонстрации "работы" этого наиболее часто используемого обработчика я решил определить в нем перерисовку фона. Вот как выглядит обработчик *OnPaint* в нашем примере:

```
void CChildWnd::OnPaint()
{
  CPaintDC dc(this); // device context for painting
                     // контекст устройства для перерисовки
  // Закрашиваем клиентскую область,
  // для чего создаем однородную кисть и ...
  CBrush brush(COLOR_BCKGR);
  // Рисуем прямоугольник
  dc.FillRect(CRect(0, 0, m_cxClient, m_cyClient), &brush);
  // Циферблат, текст и стрелки рисовать не будем,
  // этим займется функция CClockApp::OnIdle
  // Do not call CWnd::OnPaint() for painting messages
  // Это едва ли не единственное место,
  // где не надо вызывать переопределенную функцию базового класса
}
```

**Примечание**

Английский текст в обработчике *OnPaint* вставляется мастером ClassWizard.

Вы наверняка обратили внимание на вызов функции *FillRect*, которая занимается закрашиванием прямоугольника размером с клиентскую область окна. И уж точно догадались, что она принадлежит одному из классов, отвечающих за контекст устройства. И это действительно так. Поэтому здесь мне бы хотелось прервать самого себя и рассмотреть более подробно класс *CDC*, поскольку ни одно действие по выводу чего-либо в окно (рисованию в окне) не обходится без участия этого класса (прямо или опосредованно).

Однако задержимся еще ненадолго, чтобы закончить со всеми остальными, отличными от рисования, вопросами.

# Организация процесса рисования

Ясно, что нам нужна некоторая функция, которая будет заниматься собственно рисованием в окне. Назовем ее, например, *Redraw* (нам же периодически придется заниматься именно перерисовкой). Очевидно, что она должна принадлежать классу *CChildWnd*, отвечающему за окно, в котором мы будем рисовать.

Теперь необходимо определить, где и когда следует вызывать эту функцию. Первое, что приходит на ум: создать таймер и с заданной периодичностью вызывать *Redraw*. Безусловно, это один из вариантов. Однако я предлагаю рассмотреть другой способ, а работу с таймером мы разберем при обсуждении строки состояния.

Вернемся немного назад — к классу *CWinApp*, который мы рассматривали в главе 3. Как вы помните, на базе этого класса создается объект приложения. Некоторые основные его функции мы рассмотрели, нас же сейчас интересует еще одна, а именно:

```
virtual BOOL CWinApp::OnIdle (LONG 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" приложения.

Функция *CWinApp::OnIdle* вызывается в цикле обработки сообщений, когда очередь сообщений приложения *пуста*. Функция, выполняемая по умолчанию, обновляет команды объектов интерфейса пользователя, 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*.

Именно этой функцией мы и воспользуемся (естественно, сначала ее переопределив). Рассмотрим необходимые действия:

1. Откройте Class Wizard и в раскрывающемся списке Class name выберите класс нашего приложения — *CClockApp*. В окне Messages найдите и выделите виртуальную функцию *OnIdle* (рис. 8.11).
2. Нажмите последовательно кнопки Add Function и Edit Code и в созданную заготовку внесите следующий код:

```
BOOL CClockApp::OnIdle(LONG lCount)
{
  CWinApp::OnIdle(lCount);
  // Перерисовываем окно
  ((CMainFrame *)m_pMainWnd)->Redraw();
```

```
  // чтобы не закончить цикл вызова, функции OnIdle возвращаем TRUE
  return TRUE;
}
```

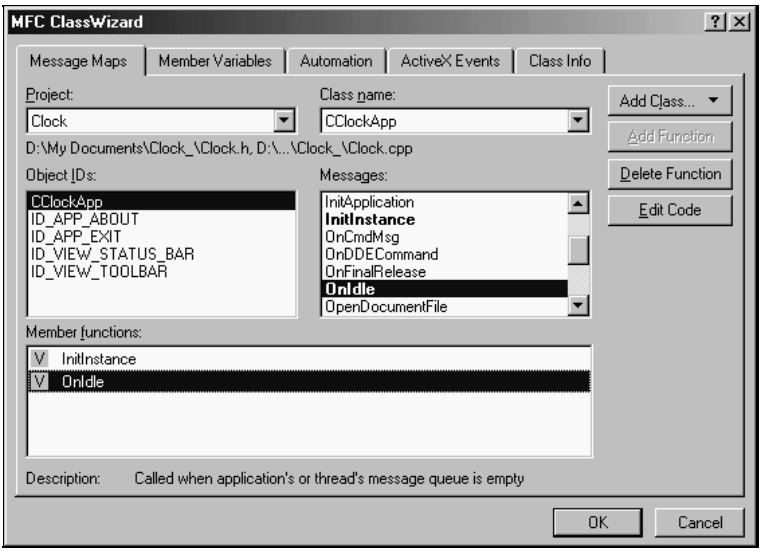



**Рис. 8.11.** Переопределяем виртуальную функцию *OnIdle*

3. В классе *CMainFrame* определяем функцию *Redraw*:

```
class CMainFrame : public CFrameWnd
{
...
// Operations
public:
  void Redraw();
...
};

// Функция перерисовки часов может вызываться
// практически в любой функции программы
// У нас это будет происходить во время "простоя" приложения
void CMainFrame::Redraw()
{
  m_wndChild.Redraw();
}
```

4. В классе *CChildWnd* определяем свою функцию *Redraw*:

```
class CChildWnd : public CWnd
{
...
```

```
// Operations
public:
  void Redraw();
...
};

void CChildWnd::Redraw()
{
  // Именно здесь будем реализовывать рисование часов
}
```

Вот как будет выглядеть окно приложения после того, как мы реализуем функции отображения часов (рис. 8.12).

И теперь мы спокойно можем переходить к знакомству с одним из основополагающих классов библиотеки MFC, а именно, классом *CDC*.




**Рис. 8.12.** Окно приложения "Аналоговые часы" при начальных размерах окна

# ГЛАВА 9



# Класс поддержки рисования

Лишь тот заслуживает памятника,
кто в нем не нуждается.
*Уильям Гэзлитт*

Мы уже говорили о важности такого понятия, как контекст устройства, и той роли, которую играет класс *CDC*. В этой главе мне хотелось бы познакомить вас с возможностями, предоставляемыми библиотекой MFC в плане работы с контекстами устройств. Глава несколько выпадает из общего построения книги и носит скорее справочный характер, однако такое решение продиктовано рядом соображений. И основным является то, что в рамках одной книги, какая бы большая она не была, невозможно рассмотреть все вопросы. Например, в разрабатываемом нами приложении "Аналоговые часы" мы коснемся достаточно многих аспектов, связанных с графическим выводом — но, естественно, не всех. Может быть, кто-то из вас в процессе знакомства с кодом найдет решение некоторых проблем, отличающееся от предложенного. Для таких читателей мне хотелось бы дать более широкий обзор возможностей. Другими словами, при последовательном чтении эту главу можно пропустить без ущерба для понимания дальнейшего материала. Итак, начнем.

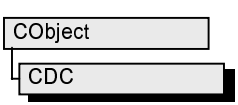



**Рис. 9.1.** Место класса *CDC* в иерархии библиотеки MFC

Базовым для всех классов, представляющих в библиотеке MFC контексты устройств, является класс *CDC* (рис. 9.1). Он предлагает разработчикам набор функций для работы с устройствами, такими как экран дисплея или принтер, а также функций, позволяющих работать с окнами Windows в це-

лом или только с клиентской их частью. Еще раз подчеркну — все операции рисования выполняются *только* для объектов класса *CDC*. Из всех классов, предоставляющих интерфейс с контекстами устройств, только этот класс 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, число функций для взаимодействия с которыми несоизмеримо с количеством функций для всех других объектов. Речь идет об окнах и контекстах устройств. Естественно, что и библиотека классов MFC обеспечивает соответствующие классы не менее широким набором функций-членов. Об окнах и классе *CWnd* мы подробно говорили в предыдущих главах, сейчас же проведем обзор функциональных возможностей класса *CDC*.

# Создание объекта класса и его связь с контекстом устройства

Все классы библиотеки MFC, непосредственно связанные с объектами Windows, обеспечивают программиста более удобным и понятным интерфейсом. Часто класс имеет в своем составе конструктор по умолчанию, т. е. без параметров, который только создает объект класса, никак не связанный с объектом Windows. Так как работа с объектами Windows происходит только через дескрипторы, класс должен иметь возможность для их хранения. Рассмотрим упомянутые механизмы и их поддержку в реализации класса *CDC*.

## Компоненты класса

Класс *CDC* имеет в своем составе даже не один, а два компонента данных для хранения дескрипторов, что обеспечивает известную гибкость.

`HDC CDC::m_hDC`
: Дескриптор контекста устройства графического вывода.

`HDC CDC::m_hAttribDC`
: Дескриптор контекста устройства, используемый 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. Аналогичный ме-

ханизм используется и для режима предварительного просмотра печати (print preview), реализованного в большом количестве приложений. Для этих целей могут применяться функции *SetOutputDC* и *SetAttribDC*, которые позволяют приложению связывать объект класса с различными контекстами устройств для графического вывода и получения текущих параметров.

Тем не менее, иногда возникает необходимость получения параметров вывода текста от обоих контекстов. В табл. 9.1 показаны пары функций, используемых в таких случаях.

***Таблица 9.1.*** *Примеры пар функций для работы с различными контекстами*

| 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 контекста устройства Windows, который, возможно, следует 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 — CreateDC*.

```
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; если значение параметра равно NULL, используются установки, заданные в системе по умолчанию.

**Примечание**

Фирма 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 расположенный в памяти контекст устройства, подобный (совместимый) 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 пиксел × 1 пиксел. Таким образом, операции графического вывода могут производиться только в том случае, 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 вывести ее полностью на экран, для чего можно воспользоваться функцией *BitBlt*, осуществляющей перенос битовых массивов между контекстами. Давайте проиллюстрируем это 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);
  dcMem.Rectangle(...) // вызов функции рисования
  // 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();
}
```

## 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 эти функции могут возвращать указатели на временные объекты соответствующих классов.

## Функции контекста устройства

Функции этой группы обеспечивают доступ к дескрипторам контекстов (*GetSafeHdc*), осуществляют сохранение и восстановление параметров контекстов устройств (*SaveDC* и *RestoreDC*). К функциям этой группы относится также 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 создания приложения, универсального с точки зрения графического вывода на различные устройства, эту функцию просто необходимо использовать. Различие устройств в поддержке тех или иных возможностей графического вывода может потребовать помощи 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 (и то не всегда) в качестве параметров только координаты вывода. Все остальные параметры, такие как цвет, стиль и т. д., задаются отдельно и действуют до их смены. Поэтому можно задавать параметры вывода сразу для группы последовательно вызываемых функций. Альтернативой такому способу могло бы быть наличие функций со многими параметрами, что затруднило бы сам процесс программирования. Кроме того, 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 хоть и велико, но ограниченно. Создание таких объектов требует определенных дополнительных расходов, тогда как возможность использования текущих настроек для нескольких функций рисования такие расходы, естественно, 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**

Так как кисти представляют собой прямоугольную область размером 8×8 пикселов, закрашивание с их помощью областей большего размера требует определенного согласования, иначе закраска не будет ровной. Например, линии штриховки будут иметь размер не больше 8 пикселов. 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.

Основная функция установки графических объектов — *SelectObject*. Библиотека MFC в классе *CDC* определяет пять функций для различных 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 соответствующего типа, ранее установленный для контекста устройства, и NULL — в противном случае. Причем указатель может указывать на временный объект и использоваться во время обработки одного сообщения Windows.

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 функцией *CBitmap::CreateCompatibleBitmap*.

Кроме создаваемых в приложениях графических объектов, существуют и предопределенные системные, которые также можно применять. Для таких объектов следует использовать функцию *SelectStockObject*, устанавливающую в контексте устройства системные графические объекты различных типов.

```
virtual CGdiObject* CDC::SelectStockObject (int nIndex)
```

Возвращает указатели на использованные ранее объекты в случае успешной замены и NULL — в противном случае. Новый объект задается параметром *nIndex*, имеющим следующие значения:

| | |
|---|---|
| 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, зависимый 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 могут использоваться только для окон, имеющих стили CS_HREDRAW и CS_VREDRAW. Связано это с тем, что базовые координаты системных кистей не могут настраиваться, 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. Далее следует описание наиболее часто используемых функций.

`int CDC::SetBkMode (int nBkMode)`

Используется для установки режима закрашивания фона. Возможные режимы задаются параметром *nBkMode*, который может принимать следующие значения:

| | |
|---|---|
| OPAQUE | (Непрозрачный) фон закрашивается текущим цветом фона при выводе текста, несплошных линий или при использовании штриховых кистей; задано по умолчанию |
| TRANSPARENT | (Прозрачный) фон не меняется |

`int CDC::SetPolyFillMode (int nPolyFillMode)`

Используется для установки режима закрашивания многоугольников. Возможные режимы задаются параметром *nPolyFillMode* (рис. 9.2):

| | |
|---|---|
| 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 |

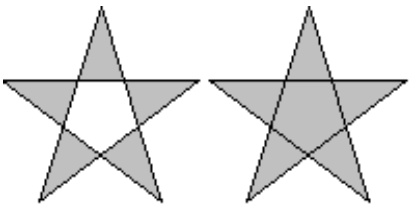


**Рис. 9.2.** Отличие режимов закрашивания: ALTERNATE — слева, WINDING — справа

Для установки режима рисования или заливки используется функция *SetROP2*. Название функции расшифровывается как "установка режима растровых (пиксельных) операций, использующих в качестве параметров два пиксела". Такими пикселами 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 | Пикселы будут черного цвета |
| R2_WHITE | Пикселы будут белого цвета |
| R2_NOP | Пикселы останутся без изменений |
| R2_NOT | Пикселы будут иметь инверсный по отношению к экрану цвет |
| R2_COPYPEN | Пикселы будут иметь цвет, заданный для текущего карандаша |
| R2_NOTCOPYPEN | Пикселы будут иметь инверсный цвет по отношению к цвету текущего карандаша |
| R2_MERGEPENNOT | Цвет пикселов получается в результате комбинирования цвета карандаша и инверсии цвета экрана ("цвет пиксела" = (NOT "цвет пиксела экрана") OR "цвет карандаша") |
| R2_MASKPENNOT | Цвет пикселов получается в результате комбинирования цвета карандаша и инверсии цвета экрана ("цвет пиксела" = (NOT "цвет пиксела экрана") AND "цвет карандаша") |
| R2_MERGENOTPEN | Цвет пикселов получается в результате комбинирования цвета экрана и инверсии цвета карандаша ("цвет пиксела" = (NOT "цвет карандаша") OR "цвет пиксела экрана") |

| | |
|---|---|
| R2_MASKNOTPEN | Цвет пикселов получается в результате комбинирования цвета экрана и инверсии цвета карандаша ("цвет пиксела" = (NOT "цвет карандаша") AND "цвет пиксела экрана") |
| R2_MERGEPEN | Цвет пикселов получается в результате комбинирования цвета экрана и цвета карандаша ("цвет пиксела" = "цвет карандаша" OR "цвет пиксела экрана") |
| R2_NOTMERGEPEN | Цвет пикселов получается в результате инверсии R2_MERGEPEN цвета ("цвет пиксела" = NOT("цвет карандаша" OR "цвет пиксела экрана")) |
| R2_MASKPEN | Цвет пикселов получается в результате комбинирования цвета экрана и цвета карандаша ("цвет пиксела" = "цвет карандаша" AND "цвет пиксела экрана") |
| R2_NOTMASKPEN | Цвет пикселов получается в результате инверсии R2_MASKPEN цвета ("цвет пиксела" = NOT("цвет карандаша" AND "цвет пиксела экрана")) |
| R2_XORPEN | Цвет пикселов получается в результате комбинирования цвета экрана и цвета карандаша ("цвет пиксела" = "цвет карандаша" XOR "цвет пиксела экрана") |
| R2_NOTXORPEN | Цвет пикселов получается в результате инверсии R2_XORPEN цвета ("цвет пиксела" = NOT("цвет карандаша" XOR "цвет пиксела экрана")) |

**Примечание**

Вышеперечисленные режимы рисования линий поддерживаются только растровыми устройствами.

Для выбора режима сжатия битовых массивов используется функция *SetStretchBltMode*:

`int CDC::SetStretchBltMode (int nStretchMode)`

Задает режим сжатия/увеличения изображений при использовании функции *StretchBlt*, в результате выполнения которой возможна потеря фрагментов изображения или их искажение. Возможные режимы задаются параметром *nStretchMode*, принимающим следующие значения:

| | |
|---|---|
| STRETCH_ANDSCANS или BLACKONWHITE | Цвет пикселов получается в результате комбинирования цвета удаляемых пикселов и остающихся с использованием операции AND; для монохромных изображений черные пикселы имеют приоритет над белыми |
| STRETCH_DELETESCANS или COLORONCOLOR | Режим простого удаления пикселов без попыток сохранить информацию об их цвете |
| STRETCH_HALFTONE или HALFTONE | При сжатии цвет пикселов результирующего блока "в среднем" сохраняется |

| | |
|---|---|
| STRETCH_ORSCANS или WHITEONBLACK | Цвет пикселов получается в результате комбинирования цвета удаляемых пикселов и остающихся с использованием операции OR; для монохромных изображений белые пикселы имеют приоритет над черными |

К этой же группе 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*).

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 просветов между символами текста. Для линий, выведенных с помощью карандашей, созданных с флагами PS_GEOMETRIC или PS_COSMETIC, цвет фона не используется.

# Режим 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?

Система Windows дает программистам возможность создавать свою собственную систему координат. Для этого, манипулируя различными параметрами, необходимо задать 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 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), ось $x$ направлена слева направо, а ось $y$ — сверху вниз. Единица измерения — один пиксел (режим MM_TEXT). Кстати сказать, именно поэтому на различных

устройствах из-за различий в размерах пиксела фактический (визуальный) размер по оси *x* может не соответствовать размеру по оси *y*.

Для настройки системы координат Windows использует два понятия — *физическая область вывода* (viewport), координаты и размеры которой задаются в физических единицах — пикселах, и *логическая область вывода* (window), координаты и размеры которой задаются в логических единицах, определяемых режимом рисования.

Давайте теперь познакомимся с режимами отображения, поддерживаемыми Windows. Всего их восемь, и основная функция, которая позволяет выбрать режим, а значит, и систему координат — *SetMapMode*.

`virtual int CDC::SetMapMode (int nMapMode)`

Эта функция устанавливает направления осей и определяет логические единицы, т. е. единицы измерения для задаваемой системы координат. Возможные режимы задаются параметром *nMapMode*, который принимает следующие значения:

| | |
|---|---|
| MM_TEXT | Одна логическая единица равна одному пикселу, ось *x* направлена вправо, ось *y* — вниз; режим задан по умолчанию |
| MM_HIENGLISH | Одна логическая единица равна 0.001 дюйма, ось *x* направлена вправо, ось *y* — вверх |
| MM_HIMETRIC | Одна логическая единица равна 0.01 миллиметра, ось *x* направлена вправо, ось *y* — вверх |
| MM_LOENGLISH | Одна логическая единица равна 0.01 дюйма, ось *x* направлена вправо, ось *y* — вверх |
| MM_LOMETRIC | Одна логическая единица равна 0.1 миллиметра, ось *x* направлена вправо, ось *y* — вверх |
| MM_TWIPS | Одна логическая единица — твипс (twips) — равна 1/20 пункта (point) или 1/1440 дюйма, ось *x* направлена вправо, ось *y* — вверх |
| MM_ANISOTROPIC | Режим позволяет настраивать (с помощью функций *SetWindowExt* и *SetViewportExt*) размерность (отдельно для каждой из 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*, т. е. 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, т. е. его система координат, соответствует способу чтения, принятому в европейских языках — слева направо и сверху вниз.

**Примечание**

Исторически в полиграфии используются такие единицы измерения, как *пункты* (points) и *твипсы* (twips), размеры которых составляют (приблизительно) 1/72 и 1/1440 дюйма, соответственно. Для обозначения точки экрана мы используем понятие *пиксел* (pixel).

Отметим, что только режимы MM_TWIPS, MM_HIENGLISH, MM_HIMETRIC, MM_LOENGLISH и MM_LOMETRIC обеспечивают приложения, с одной стороны, физически понятными единицами измерения, а с другой — определенной независимостью от устройств вывода.

Для первых шести режимов можно менять только начало координат, соответствующее по умолчанию левому верхнему углу области вывода. Для режимов MM_ANISOTROPIC и MM_ISOTROPIC дополнительно можно настраивать направление осей и 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).
```

В свою очередь, для настройки начала координат логической области вывода используются 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. Причем 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 настраивать начало координатной сетки есть смысл только для одной из областей, т. к. система координат логической области вывода привязана к системе координат физической области. Поясним сказанное на примерах.

После настроек функции *OnPaint* классов *CVenusFrame* и *CEarthFrame* будут работать с координатной сеткой, начало координат которой располагается в середине клиентской области (рис. 9.3).

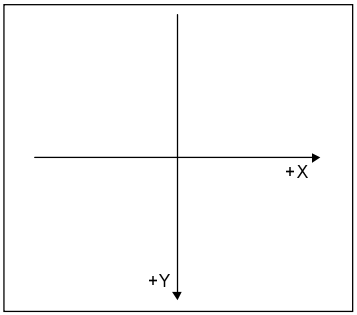



**Рис. 9.3.** Более привычная, но все еще перевернутая система координат

```
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);
  ...
}
```

В следующем примере попытка настройки систем координат обеих областей вывода в классе *CMarsFrame* приводит к несколько неожиданным результатам (рис. 9.4), так что такую последовательность, скорее всего, можно считать ошибочной.

```
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);
  ...
}
```

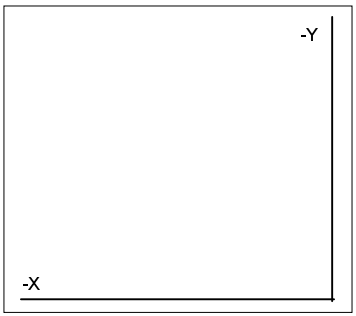



**Рис. 9.4.** С такой системой координат работать будет совсем непривычно

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 находится как отношение соответствующих размеров областей вывода. Однако для большинства режимов отображения коэффициент масштабирования изменить невозможно.

**Примечание**

Размеры области, куда будет осуществляться вывод, определяются размерами клиентской части 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. Эти значения используются для преобразования логиче-

ских 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, то 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 или типа устройства.

Ниже приводится код, 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. 9.5). При этом вывод может осуществляться и по координатам, большим 100 или меньшим −100, соответственно.

**Примечание**

Режим MM_ISOTROPIC предполагает отношение единиц по обоим направлениям как 1:1. Таким образом, задание одинаковых размеров области вывода в окне, имеющем прямоугольную форму (не квадрат), приводит к тому, что по одной из осей фактически размер области вывода может быть больше заданного. С другой стороны, никто не запрещает вызывать функции рисования вне клиентской области окна — они не окажут 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);
  ...
}
```

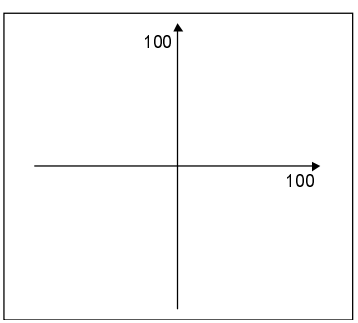



**Рис. 9.5.** Вот теперь мы имеем привычную систему координат

Кроме вышеперечисленных функций для установки начала координат и размеров осей, класс *CDC* содержит ряд функций для модификации 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 продолжают понимать только физические единицы устройства — пикселы — и для них 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* — используются 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, некоторые функции группы, например *GetClipBox* и *PtVisible*, позволяют определить 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 процесс вычисления (например, при выводе 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*), установленных в 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.

## Базовые функции рисования

Функции этой группы осуществляют рисование закрашенных простейших объектов, прежде всего прямоугольников (*FillRect*, *FillSolidRect* и *InvertRect*), с возможностью задания различного типов рамок (*FrameRect*, *Draw3dRect* и *DrawEdge*). К группе относятся функции рисования пиктограмм и битовых массивов (*DrawIcon* и *DrawState*) и элементов управления — кнопок, меню, заголовков окон и полос прокрутки (*DrawFrameControl*). Большинство функций используют для рисования текущие карандаш и кисть.

Остановимся на функции, вернее, на функциях *DrawState* немного подробнее.

### Битовые массивы

```
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)
```

### Пиктограммы

```
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)
```

### Строки

```
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

Общие два первых параметра 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 | Пиктограмма; в последних двух версиях функции младшее слово параметра *lData* содержит дескриптор пиктограммы |
| 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 нулевым 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 нулевым символом |

Из названия функции *DrawState* не совсем ясно, причем тут состояние (state). Параметр *nFlags* может комбинировать со значением типа изображения следующие флаги состояния, 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. Для выделения прямоугольной области можно использовать функцию *DrawFocusRect*, которая выполняет над содержимым области операцию XOR.

Здесь мне хотелось бы обратить ваше внимание на одну особенность рисования "гладких" линий на примере функции *Pie*:

```
BOOL CDC::Pie(
  HDC hDC,          // дескриптор контекста устройства
  int nLeftRect,    // x-координата верхнего левого угла
  int nTotRect,     // y-координата верхнего левого угла
  int nRightRect,   // x-координата нижнего правого угла
  int nBottomRect, // y-координата нижнего правого угла
  int nXStart,      // x-координата начала сектора
  int nYStart,      // y-координата начала сектора
  int nXEnd,        // x-координата конца сектора
  int nYEnd);       // y-координата конца сектора
```

Для соединения точки (*nXStart, nYStart*) с центром эллипса система использует воображаемую линию. В точке пересечения этой линии с ограничивающим прямоугольником система Windows начинает рисовать дугу эллипса в направлении против часовой стрелки. Рисование дуги завершается в точке пересечения воображаемой линии, соединяющей точку (*nXEnd, nYEnd*) с центром эллипса, и ограничивающего прямоугольника (рис. 9.6).

Точность рисования при таком подходе, конечно же, невысокая, но при этом требуется значительно меньше вычислений, и функция работает достаточно быстро.

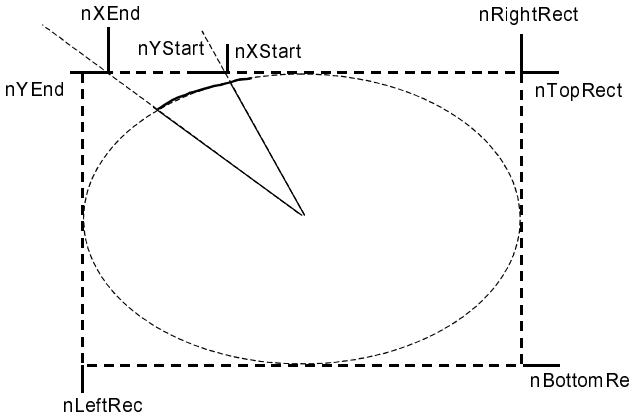



**Рис. 9.6.** Сектор, нарисованный с помощью функции *Pie*

## Контуры

*Контур* представляет собой совокупность прямых и изогнутых линий, которая хранится в системе и используется для создания сложных графических объектов. В то же время контур может применяться для создания регионов, например, для ограничения области отсечения.

Чтобы создать контур, необходимо сообщить системе о том, что все последующие вызовы функций рисования должны быть использованы для этой операции. В начале процесса нужно вызвать функцию *CDC::BeginPath*, а по завершении создания контура — *CDC::EndPath*. Между вызовами этих двух функций реальный вывод не осуществляется. Для обвода или закраски контура служат функции *CDC::StokePath* и *CDC::FillPath*, а чтобы обвести и одновременно закрасить замкнутый контур, необходимо воспользоваться функцией *CDC::StrokeAndFillPath*.

Только часть функций рисования может быть использована для создания контура. Вот они:

| | | |
|---|---|---|
| `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, сюда относятся 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 (рис. 9.7), что может быть чрезвычайно 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);
  // т. к. 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. 9.7.** Без использования регионов добиться аналогичного результата было бы затруднительно

Немного 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. Ниже перечислены возможные режимы (они также используются и в функции создания новой области отсечения с помощью 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

Графические возможности Windows были бы существенно ограничены, если бы допускали рисование только геометрических фигур или только отображение и копирование растровых (точечных) изображений. Система Windows и вместе с ней класс *CDC* предоставляют набор достаточно мощных функций, которые позволяют выводить любые рисунки, дополнительно масштабируя их или осуществляя вращение. (К сожалению, последняя возможность есть только в Windows NT — функция *PlgBlt*.)

## Функции, использующие битовые массивы

Функции этой группы применяются 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 в результате 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.

Возможно использование следующих растровых операций:

| | |
|---|---|
| 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* |

**Примечание**

Данная функция выполняется существенно быстрее для окон, имеющих стили CS_BYTEALIGNWINDOW или CS_BYTEALIGNCLIENT. Для указания этих стилей необходимо зарегистрировать класс окна при помощи глобальной функции MFC *::AfxRegisterWndClass*.

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 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)
```

Функция 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 эти параметры 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 определяющий область закрашивания

пиксел, координаты которого заданы параметрами *x* и *y*, имеет цвет, совпадающий с *crColor*, — при режиме закрашивания FLOODFILLBORDER, имеет цвет, не совпадающий с *crColor*, — при режиме закрашивания FLOODFILLSURFACE или если пиксел 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 возможны 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 |

**Примечание**

Функция *CDC::ExtFloodFill* 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 я рассмотрю возможности вывода текста, получения информации о выбранных шрифтах и т. п. С самими объектами рисования — шрифтами — мы познакомились в предыдущей главе.

## Вывод текста

Функции предназначены для организации вывода текстовых строк. Группа включает функции, выполняющие все необходимые действия от простейшего вывода и настройки режимов выравнивания до 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 осуществлять вывод текстовой строки 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 версий состоит в механизме задания строки символов. В первой версии строка задается параметром *lpszString*, а ее длина — параметром *nCount*; если последний равен –1, то длина строки вычисляется. Во второй версии строка задается ссылкой на объект класса *CString*, определяемый параметром *str*.

Параметр *lpDxWidths* указывает на массив чисел, определяющих расстояние между символами; если параметр равен 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 которой должен происходить вывод текста, и может быть равен NULL.

Параметр *nOptions* может принимать одно, оба или не принимать ни одного из следующих значений:

| | |
|---|---|
| ETO_CLIPPED | Вывод текста осуществляется только в области, заданной 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* игнорируются, вывод осуществляется с текущей позиции, а по завершении вывода текущая позиция устанавлива-

ется либо в конец выведенной строки, либо в соответствии с последним элементом массива (параметр *lpDxWidths*) в зависимости от того, какое значение больше.

Функция *DrawText* осуществляет текстовый вывод с различными способами 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*, содержащие координаты (в логических единицах) прямоугольной области, в которую, собственно, и происходит форматный вывод текста.

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 | Служит для определения ширины и высоты прямоугольной области вывода; для текста в несколько строк используется ширина прямоугольной области, заданной параметром *lpRect,* и настраивается высота так, чтобы поместить весь текст; 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 выравнивание по центру (по горизонтали) заданной области |
| DT_EXPANDTABS | Производит при выводе замену 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 | Указывает 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* содержится количество 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 выравнивание по центру (по вертикали) 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 текущие шрифт, цвет 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 игнорируется — вывод осуществляется с текущей позиции, а по завершении вывода текущая позиция устанавливается в конец выведенного текста. При задании флага TA_UPDATECP флаг DT_WORDBREAK игнорируется.

К этой же группе относятся функции *GetOutputTextMetrics* и *GetTextMetrics*, которые поставляют приложению основную информацию о текущем шриф-

те 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)
```

Обе функции возвращают TRUE в случае успешного завершения и FALSE в противном случае. В качестве единственного параметра используется указатель на структуру *TEXTMETRIC*, которая и будет содержать, собственно, запрашиваемую информацию. Важность последней требует подробного описания структуры. 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 представлена на рис. 9.8:

| | |
|---|---|
| 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 |

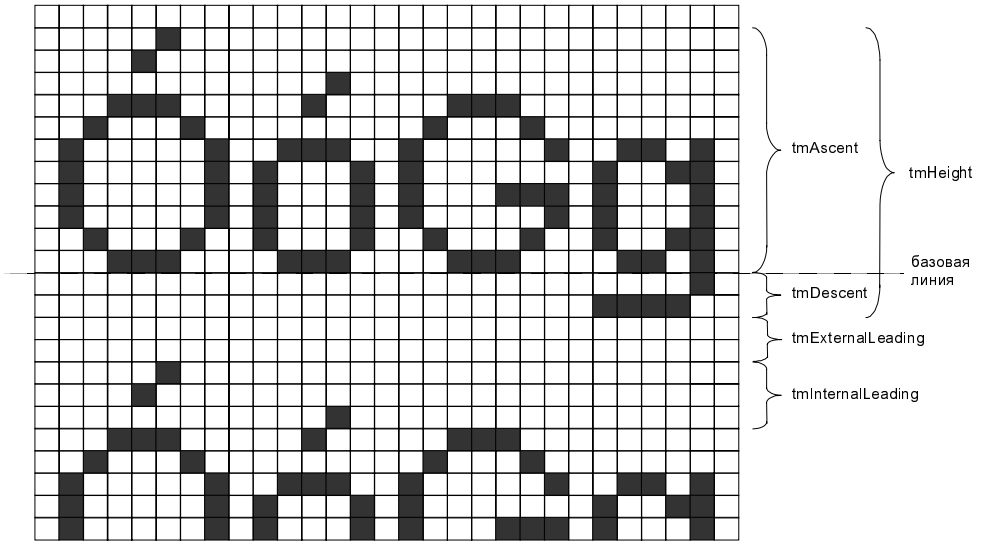



**Рис. 9.8.** Определение основных параметров шрифта с помощью элементов структуры *TEXTMETRIC*

| | |
|---|---|
| tmExternalLeading | Размер области между строками текста; при выводе эта область никак не заполняется ни в режиме OPAQUE (непрозрачный), ни в режиме TRANSPARENT (прозрачный) |
| 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.<br>Старшие четыре бита определяют семейство шрифтов: FF_DECORATIVE, FF_DONTCARE, FF_MODERN, FF_ROMAN, FF_SCRIPT, FF_SWISS.<br>Младшие четыре бита определяют тип шрифта: |
| TMPF_FIXED_PITCH | Пропорциональный шрифт, если бит установлен, и моноширинный (все символы шрифта имеют одинаковую ширину), если сброшен |
| TMPF_VECTOR | Векторный шрифт |
| TMPF_TRUETYPE | Если бит установлен, то шрифт является шрифтом TrueType |
| TMPF_DEVICE | Если бит установлен, то шрифт является шрифтом устройства |
| tmCharSet | Определяет таблицу символов шрифта |

**Примечание**

В русском языке тоже есть символы, использующие область InternalLeading, например, "Й", который выше "И" как раз на величину tmInternalLeading.

## Информация о шрифтах

Функции этой группы служат для получения полной информации о шрифтах (*GetFontData*, *GetKerningPairs*, *GetOutlineTextMetrics*, *GetGlyphOutline*, *GetCharABCWidth*, *GetCharWidth* и *GetOutputCharWidth*), что позволяет 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 непосредственно не занимаются 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

(видеокартами). Такие функции (*Escape* и *DrawEscape*, соответственно) дают возможность программе выполнять действия, не реализованные 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 10



# Рисуем графические объекты

Вы умеете рисовать? Очень жаль.
Я, к сожалению, тоже не умею...

А буквы вы умеете? Тоже не умеете? Совсем нехорошо...
*Остап Бендер*

Теперь, когда мы подготовили наше приложение и познакомились с особенностями рисования в Windows на основе библиотеки MFC, можно приступать непосредственно к "рисованию" наших часов.

В главе 8 мы остановились на том, что создали функцию *CChildWnd::Redraw*, не наполнив ее никаким содержимым. Пора восполнить этот пробел.

Сам по себе алгоритм достаточно прост:

1. Проверим, не свернуто ли окно — нет смысла рисовать, если все равно никто ничего не увидит.
2. Поскольку мы реализовали механизм перерисовки содержимого окна во время "простоя" приложения, то вполне возможна ситуация, когда текущее время еще не успело измениться. Очевидно, что и в этом случае незачем заниматься перерисовкой.
3. Получим доступ к контексту клиентской области окна — без него мы ничего нарисовать не сможем.
4. Установим систему координат, относительно которой мы будем рисовать.
5. Теперь нарисуем циферблат.
6. Нарисуем стрелки "по старому месту" — перед выводом нового значения времени необходимо стереть старые стрелки.
7. Поскольку надписи на циферблате могут перекрываться стрелками, то обновить их можно только теперь.

8. Выводим стрелки по их новому месту.
9. Осталось только "прибрать за собой" — восстановить измененные параметры контекста устройства и записать новое значение текущего времени.

Вот, собственно, и все. Посмотрите, как это выглядит в коде:

```
// Функция перерисовки часов может вызываться
// практически в любом месте программы
//
void CChildWnd::Redraw()
{
  static CTime m_tmPrev; // переменная статическая,
                         // поэтому ее значение не будет изменяться
                         // между вызовами функции
  // Если окно свернуто, нет необходимости что-либо перерисовывать
  if (IsIconic())
    return;
  // Поскольку мы рисуем часы — без получения текущего времени
  // нам не обойтись
  CTime tmCurrent = CTime::GetCurrentTime();
  // Моменты времени, когда мы будем рисовать, у нас произвольные,
  // поэтому сначала проверяем, изменились ли текущие значения
  if (tmCurrent != m_tmPrev)
  {
    // Получаем доступ к контексту клиентской области окна
    CClientDC dc(this);
    // Устанавливаем нашу систему координат
    SetMode(&dc);
    // Рисуем циферблат
    DrawClock(&dc);
    // Закрашиваем цветом фона ранее выведенные стрелки,
    // другими словами — стираем их
    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);
    // Восстанавливаем параметры контекста устройства,
    // которые у него были перед тем, как мы их начали менять
    dc.SelectObject(pBrush);
    dc.SelectObject(pPen);
  }
```

```
  // Изменяем значение текущего времени
  m_tmPrev = tmCurrent;
}
```

А мне осталось только привести некоторые комментарии.

# Проверка того, что окно свернуто

Для проверки "состояния" окна — не свернуто ли оно, мы воспользовались функцией класса *CWnd* — *CWnd::IsIconic*, которая возвращает TRUE, если окно свернуто, и FALSE — в противном случае.

# Установка системы координат

Для установки системы координат я реализовал функцию *SetMode*, код которой приведен ниже:

```
// Функция формирования системы координат
//
void CChildWnd::SetMode(CDC *pDC)
{
  // Устанавливаем масштаб, начало и направление
  // 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);
}
```

Прежде всего необходимо установить режим отображения:

```
pDC->SetMapMode(MM_ISOTROPIC);
```

Почему выбран именно этот режим — MM_ISOTROPIC? Причина должна быть вам уже понятна: в этом случае по осям *x* и *y* устанавливается одинаковая размерность.

Далее задаем размеры логической

```
pDC->SetWindowExt(1000, 1000);
```

и физической областей вывода

```
pDC->SetViewportExt(m_cxClient / 2, -m_cyClient / 2);
```

Знак "минус" у второго параметра сообщает, что ось *y* будет направлена вверх.

И, наконец, начало физических координат помещаем в центр окна:

```
pDC->SetViewportOrg(m_cxClient / 2,  m_cyClient / 2);
```

В результате начало координат — в середине клиентской области окна, ось *x* направлена влево, ось *y* — вверх, а размер окна — 2000×2000 логических единиц:

```
X: [-1000; +1000]
Y: [-1000; +1000]
```

# Рисование циферблата часов

Аналогично следует поступить (т. е. реализовать отдельную функцию, чтобы не загромождать код) и с функцией для рисования циферблата часов — *CChildWnd::DrawClock*. Но сначала о том, что мы собираемся делать (я не буду описывать необходимые для реализации нашего замысла математические выкладки — это выходит за рамки поставленных задач):

1. Нарисовать отметки для каждого часа, причем для 12, 3, 6 и 9 часов будем рисовать прямоугольники со скругленными углами, а для остальных — круги.
2. Нарисовать дуги между часовыми отметками. Их диаметр следует сделать немного больше диаметра окружности, на которой расположены часовые отметки.
3. Нарисовать "углы" часов. Для демонстрации различных подходов сделаем это тремя способами.

А вот и программная реализация описанного алгоритма:

```
// Функция рисования циферблата часов
//
void CChildWnd::DrawClock(CDC *pDC)
{
  CPen *pPen;
  CBrush *pBrush;
  int  nAngle;

  // Рисуем отметки для каждого часа,
  // причем для 12-ти, 3-х, 6-ти и 9-ти часов
  // отметки — прямоугольники,
  // для остальных — круги
  //
  // Выбираем в контекст устройства кисть, созданную для циферблата,
  pBrush = pDC->SelectObject(&m_clkBrush);
  // и шрифт для названия часов
  CFont *pFont = pDC->SelectObject(&m_Font_1);
```

```
// Рисуем отметки для каждого часа
for (nAngle = 0; nAngle < 360; nAngle += 30)
{
  // Не буду загромождать текст пояснениями математических
  // выкладок — примите на веру или проверьте самостоятельно
  POINT pt[2] = { 0, 900 };
  POINT pt1[1] = {0, 850};
  // Поскольку вращение точек необходимо в разных местах,
  // выделяем это действие в отдельную функцию
  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;
  // В обеих использованных ниже функциях pt[0] и pt[1] задают
  // размеры ограничивающего фигуру прямоугольника
  if (nAngle % 90)
    // Для 12, 3, 6 и 9 часов рисуем круги
    pDC->Ellipse(pt[0].x, pt[0].y, pt[1].x, pt[1].y);
  else
    // для остальных — скругленные прямоугольники
    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 диаметра окружности, на которой расположены
// часовые отметки
//
// Для рисования дуг между часовыми отметками
// выбираем в контекст устройства специальный карандаш
pPen = pDC->SelectObject(&m_rndPen);
POINT ptPrev;
// Опять опускаю математическое обоснование
for (nAngle = -30; nAngle < 360; nAngle += 30)
{
  POINT pt[2] = { -45, 950, 45, 950 };
  RotatePoint(pt, 2, nAngle);
  if (nAngle != -30)
    // Рисуем дугу, радиус которой, как мы договорились,
    // несколько меньше радиуса окружности, где расположены
    // часовые отметки
    pDC->Arc(-980, -980, 980, 980,
             pt[0].x, pt[0].y, ptPrev.x, ptPrev.y);
```

```
  ptPrev = pt[1];
}
// Рисуем "углы" часов, причем тремя различными способами
//
CRect round(-1100, -1100, 1100, 1100);
CRgn rgn;
CRect rect;
// Для рисования "углов" выбираем в контекст устройства
// специальные карандаш и кисть
pDC->SelectObject(&m_crnPen);
pDC->SelectObject(&m_crnBrush);

// 1-й способ:
// Рисуем левый верхний угол
//
rect = CRect(-950, 950, -550, 550);
// Преобразуем логические координаты прямоугольника в физические
pDC->LPtoDP(&rect);
// Формируем область отсечения,
// для чего нужны координаты в единицах
// устройства вывода
rgn.CreateRectRgnIndirect(&rect);
// Выбираем сформированный регион в контекст устройства
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);
// Поскольку функция LineTo не изменяет значение текущих
// координат, сразу можно рисовать обе линии
pDC->LineTo(-950, 550);
// Делаем доступной для рисования всю клиентскую часть окна
pDC->SelectClipRgn(NULL);
// До сих пор был задействован только текущий карандаш,
// теперь же в дело вступает текущая кисть
// Заполняем замкнутую область
// с внутренней точки до границы
pDC->ExtFloodFill(-900, 900, COLOR_CRNER, FLOODFILLBORDER);
// Рисуем правый верхний угол,
// здесь все действия аналогичны предыдущим.
// 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
// внутренней точки
pDC->ExtFloodFill(900, 900, COLOR_BCKGR, FLOODFILLSURFACE);

// 2-й способ:
// Рисуем правый нижний угол
// Создаем два региона:
// первый — прямоугольный, в него вписан
// радиус окружности
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 полученных регионов создаем
// комбинированный с нужной нам формой и ...
rgn.CombineRgn(&rgn1, &rgn2, RGN_DIFF);
// ... закрашиваем его
pDC->PaintRgn(&rgn);
// Убираем за собой
rgn.DeleteObject();
rgn1.DeleteObject();
rgn2.DeleteObject();

// 3-й способ:
// Рисуем левый нижний угол,
// с помощью контура и кривых Безье
// 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, аналогично первым двум способам
    pDC->MoveTo(-950, -550);
    pDC->LineTo(-950, -950);
    pDC->LineTo(-550, -950);
  pDC->EndPath();
  // Закрашиваем пространство, ограниченное
  // сформированным контуром
  pDC->FillPath();

  // Мы нарисовали все, что хотели, поэтому
  // восстанавливаем параметры контекста устройства
  pDC->SelectObject(pPen);
  pDC->SelectObject(pBrush);
}
```

Приведенный код функции *DrawClock* я снабдил большим числом комментариев, которые, надеюсь, избавили меня от необходимости в каких-либо дополнительных пояснениях.

Нам осталось рассмотреть всего два момента: обновление надписей на циферблате и рисование стрелок.

# Вывод текста

Как вы помните, мы предусмотрели три различных шрифта для надписей на циферблате часов. Один (самый крупный) будем использовать для названия, второй (средний) — для надписи о том, что это кварцевые часы, и третий — для вывода некоторой дополнительной информации. Но перейдем непосредственно к коду:

```
// Функция рисования строк текста на циферблате
//
void CChildWnd::DrawLabel(CDC *pDC)
{
  // Надписи будем выводить по центру
  pDC->SetTextAlign(TA_CENTER);
  // Первые две строки рисуем так, чтобы
  // промежутки заполнялись цветом фона
  COLORREF color = pDC->SetBkColor(COLOR_BCKGR);
  // Выбираем в контекст устройства шрифт для надписи
  CFont *pFont = pDC->SelectObject(&m_Font_1);
```

```
  // ... и выводим ее в окно
  // (текст задаем непосредственно в параметре)
  pDC->TextOut(0,  480, _T("Петербург"), 9);
  // Для вывода надписи о том,
  // что часы кварцевые, используем другой способ
  // Получаем параметры текущего шрифта
  TEXTMETRIC tm;
  pDC->GetTextMetrics(&tm);
  // Выбираем в контекст устройства соответствующий шрифт
  pDC->SelectObject(&m_Font_2);
  // Выводим надпись, сместив ее на расстояние,
  // равное сумме высоты символов и размера
  // области между строками текста
  // (все величины взяты для предыдущего шрифта)
  pDC->ExtTextOut(0,  480-(tm.tmHeight+tm.tmExternalLeading),
                  ETO_OPAQUE, NULL,
                  _T("Кварцевые"), 9, NULL);
  // Восстанавливаем в контексте устройства цвет фона
  pDC->SetBkColor(color);
  // Последнюю строку рисуем так, чтобы
  // промежутки между контурами символов
  // оставались без изменения — цвет
  // фона влияния на эти области не оказывает
  pDC->SetBkMode(TRANSPARENT);
  // Выбираем в контекст устройства соответствующий шрифт
  pDC->SelectObject(&m_Font_3);
  // и выводим его в окно
  pDC->TextOut(0, -500, _T("Противоударные"));
  // Восстанавливаем "старый" шрифт
  pDC->SelectObject(pFont);
}
```

На мой взгляд, и здесь от меня не требуется дополнительных комментариев.

# Рисование стрелок

Для рисования стрелок я использовал функцию класса *CDC — Polygon*. Это единственное место, которое отвечает за отображение. Весь остальной код — это подготовка к выводу трех многоугольников.

```
// Функция вывода стрелок установленными вне
// ее текущими карандашом и кистью
//
void CChildWnd::DrawArrows(CDC *pDC, CTime &tm)
```

```
{
  // Массив точек, которые определяют контуры стрелок
  static POINT pt[3][11] =
  {
    // часовая стрелка
      0,  -50, -50,    0, -5,   40, -5,  100, -30,  120,
      0,  600,  30,
    120,    5, 100,    5, 40,   50,  0,    0, -50,
    // минутная стрелка
      0,  -40, -40,    0, -5,   30, -5,   90, -20,  110,
      0,  800,
     20,  110,   5,   90,  5,   30, 40,    0,   0,  -40,
    // секундная стрелка
      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];
  // Вычисляем углы поворота каждой из трех стрелок
  // в зависимости от текущих значений часов, минут и секунд
  nAngle[0] = (tm.GetHour()   * 30) % 360 + tm.GetMinute() / 2;
  nAngle[1] = (tm.GetMinute() * 60 + tm.GetSecond()) / 10.0;
  nAngle[2] = (tm.GetSecond() *  6);
  // Чтобы не "потерять" исходные значения стрелок,
  // создаем временный массив
  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);
  }
}
```

Вот пока и все, что я хотел вам рассказать о выводе на экран графической информации. Естественно, что тема еще не закончена, и мы к ней обязательно вернемся. А сейчас перейдем к рассмотрению дополнительных возможностей взаимодействия пользователя и приложения.



# Часть IV

## MFC и взаимодействие с пользователем

**Глава 11.** Вводим элементы пользовательского интерфейса

**Глава 12.** В глубине меню

**Глава 13.** Создание многодокументных приложений

# ГЛАВА 11



# Вводим элементы пользовательского интерфейса

Нехорошо человеку быть одному
и сотворим ему помощника соответственного ему.
*Бытие, 2,18*

Начиная с этой главы, мы с вами начнем разработку нового приложения — простейшего графического редактора, которое позволит нам познакомиться со следующими вопросами:

1. Создание панелей инструментов и работа с ними.
2. Создание строки состояния и работа с ней.
3. Динамическое изменение курсора.
4. Вычерчивание геометрических фигур с помощью мыши.
5. Работа с меню.

Итак, прежде всего необходим каркас приложения. Те из вас, кто работает с Visual C++ 6.0, должны создать стандартное MFC-приложение, отказавшись от поддержки архитектуры "документ/представление". Для тех же читателей, которые используют Visual C++ 5.0, я подготовил специального мастера — MyWizard4, чтобы не начинать все практически с нуля.

**Примечание**

Внешний вид приложений, созданных стандартным мастером Visual C++ 6.0 и моим, несколько отличается друг от друга, но это никак не повлияет на работу приложения.

Прежде всего посмотрите на состав проекта (рис. 11.1) и убедитесь, что основные изменения, по сравнению с нашим предыдущим приложением, коснулись ресурсов. Добавилось меню и панель инструментов. Вопросы,

связанные с меню, мы частично (на необходимом на данный момент уровне) уже рассматривали и будем еще развивать эту тему в дальнейшем. Сейчас же займемся панелями инструментов.

# Работа с панелями инструментов

Любое современное профессионально написанное приложение имеет в своем главном окне как минимум два дополнительных инструментальных средства, 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 со своей стороны поддержку этих элементов пользовательского интерфейса. На самом деле создать панель управления не сложнее, чем обыкновенное окно, тем более используя классы MFC. Давайте подробнее познакомимся с этими удобными и полезными компонентами приложения.

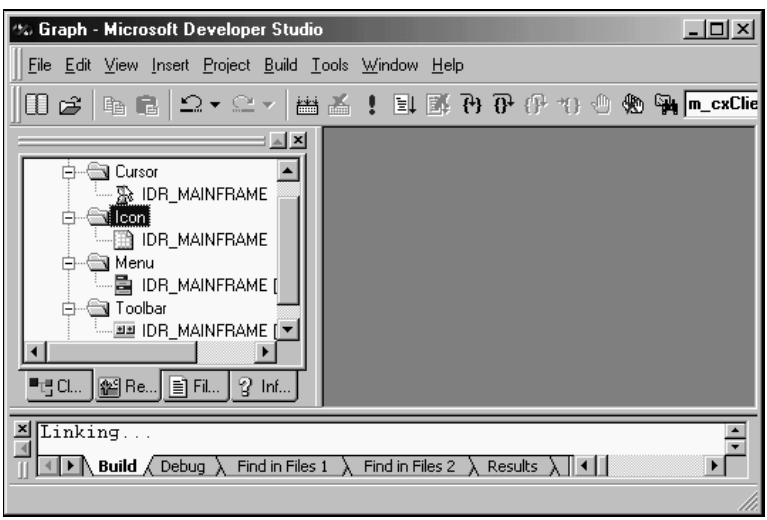



**Рис. 11.1.** В приложении появились новые ресурсы

Для того чтобы можно было работать с панелью инструментов, ее необходимо прежде всего создать. Сделать это очень просто (особенно, если вы умеете рисовать). Щелкните правой кнопкой мыши на папке Toolbar в Resource View, а затем выберите из контекстного меню Insert Toolbar (Вставить панель инструментов), и Visual C++ автоматически создаст новую панель инструментов. Выведите на экран окно свойств панели инструментов и в поле ID введите какой-либо содержательный идентификатор (рис. 11.2).

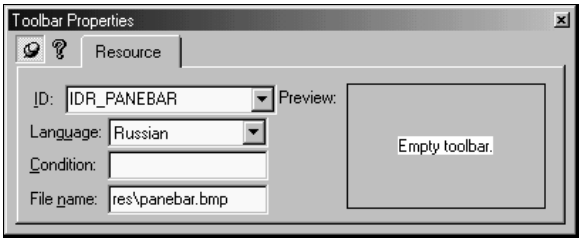



**Рис. 11.2.** Свойства панели инструментов

Теперь осталось заполнить панель кнопками. Сделать это несложно — выделите "пустую" кнопку и творите. Вот что получилось у меня (рис. 11.3). После того как кнопка "нарисована", необходимо присвоить ей идентификатор. Для этого дважды щелкните на кнопке, чтобы вывести окно свойств, и в нем введите необходимую информацию, например, так, как это показано на рис. 11.4.

Вот и все. Новая панель инструментов создана, и каждой кнопке присвоен идентификатор, интерпретируемый как команда — совершенно аналогично команде меню.

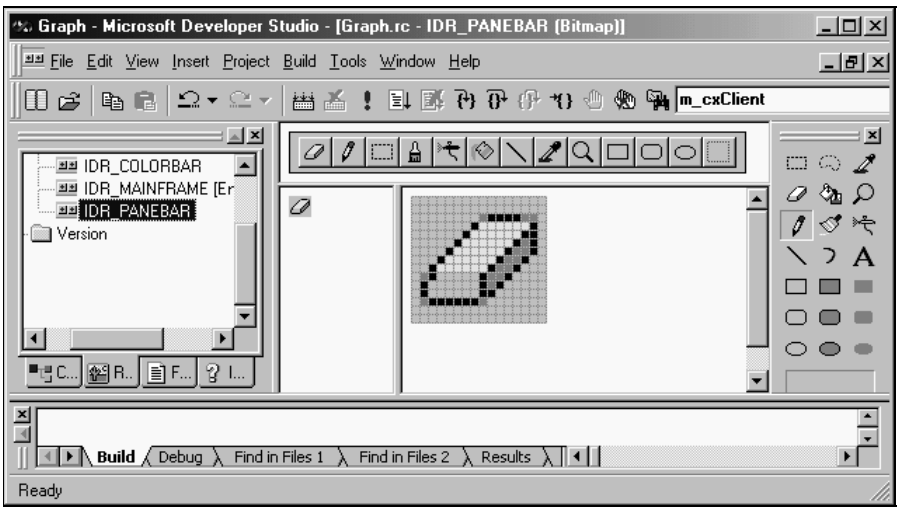



**Рис. 11.3.** Пример панели инструментов рисования

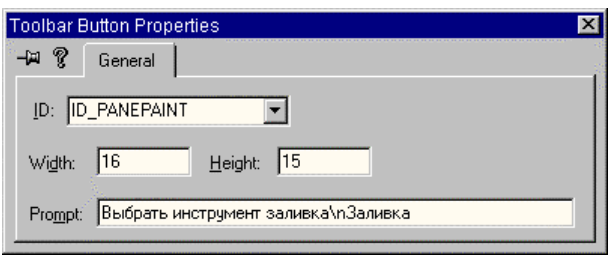



**Рис. 11.4.** Свойства кнопки панели инструментов

Теперь необходимо добавить код, который создаст объект Windows для панели инструментов. Как вы помните, панель инструментов лучше всего ассоциировать с классом *CMainFrame*. Библиотека MFC для панелей инструментов предлагает два класса, являющихся производными от класса *CControlBar* (рис. 11.5). Мы кратко рассмотрим их в следующем разделе.

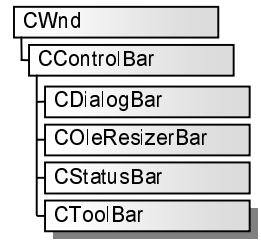



**Рис. 11.5.** Иерархия объектов инструментов

## Класс *CToolBar*

Этот класс определяет поведение панели инструментов (toolbar) — панели управления, отображающей полосу растровых кнопок (все они должны быть одного размера — стандартно 23×22 пиксела) и разделителей. Кнопки активизируют команды, т. е. нажатие на кнопку 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 инструментов либо привязана к какой-либо стороне родительского фрейма, либо является плавающей. В последнем случае пользователь может изменять ее размер. Класс *CToolBar* поддерживает и работу с всплывающей подсказкой — маленьким окном, содержащим текстовую строку.

Для изображения всех кнопок панели инструментов создается один битовый массив, который должен содержать один и только один образ для каждой кнопки. Обычно порядок расположения образов совпадает с тем, как они отображаются на экране. Все образы должны иметь один и тот же размер (обычно 16×15 пикселов) 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 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 следующие функции: *CCmdUI::Enable*, *CCmdUI::SetCheck* и *CCmdUI::SetRadio*.

Теперь, когда у вас сложилось общее представление о панели инструментов, мы, как обычно, рассмотрим основные функциональные возможности, предоставляемые классом *CToolBar*.

Поскольку панель инструментов — это окно, то для ее создания необходимо проделать стандартные шаги:

1. Создать объект класса (в классе *CMainFrame*)

```
CToolBar m_wndPaneBar;
```

Конструктор класса

```
CToolBar::CToolBar ()
```

создает объект класса и устанавливает размеры по умолчанию: размер кнопок — 23×22, размер рисунков на них (образов) — 16×15, высота верхней и нижней рамки — 3 пиксела.

2. На базе предопределенного оконного класса TOOLBARCLASSNAME создать окно Windows. Сделать это лучше всего в обработчике *OnCreate* сообщения WM_CREATE класса *CMainFrame*:

```
if (!m_wndPaneBar.Create(this,
                    WS_CHILD | WS_VISIBLE | CBRS_SIZE_FIXED |
                    CBRS_TOP | CBRS_TOOLTIPS) ||
    !m_wndPaneBar.LoadToolBar(IDR_PANEBAR))
{
  TRACE0("Ошибка при создании панели инструментов рисования\n");
  return -1;      // при создании произошла ошибка
}
```

**Примечание**

Пусть вас не пугают слова "... создать на базе предопределенного оконного класса TOOLBARCLASSNAME...". Они приведены только для информации — ничего специально делать не надо. Фактически мы создаем новое окно, а о том, как это делается, мы говорили уже достаточно.

Для создания окна Windows и присоединения его к объекту предназначены две функции класса *CToolBar*:

- "Старая", использованная нами и применявшаяся в предыдущих версиях

```
BOOL CToolBar::Create (
  CWnd *pParentWnd,
  DWORD dwStyle = WS_CHILD | WS_VISIBLE | CBRS_TOP,
  UINT nID = AFX_IDW_TOOLBAR)
```

В качестве аргументов передаются параметры: *pParentWnd* — указатель на родительское окно панели инструментов, *nID* — идентификатор дочернего окна (напомню, что панель инструментов является дочерним окном) и *dwStyle* — стиль панели инструментов, который может принимать следующие значения:

| | |
|---|---|
| CBRS_TOP | Расположить панель вверху рабочей области родительского окна |
| CBRS_BOTTOM | Расположить панель внизу рабочей области родительского окна |

| | |
|---|---|
| CBRS_NOALIGN | Не переопределять размеры панели в случае, если изменились родительские |
| CBRS_TOOLTIPS | Выводить краткое описание кнопок |
| CBRS_SIZE_DYNAMIC | Разрешает пользователю при помощи мыши менять размер панели инструментов, если она находится в плавающем состоянии |
| CBRS_SIZE_FIXED | Запрещает менять размер панели инструментов |
| CBRS_FLOATING | Панель инструментов плавающая |
| CBRS_FLYBY | В строке состояния отображать информацию о кнопке |
| CBRS_HIDE_INPLACE | Сделать панель невидимой для пользователя |

Помимо вышеперечисленных можно задавать также стили окон Windows и общие стили для всех панелей управления:

| | |
|---|---|
| CBRS_ALIGN_TOP | Панель управления привязывается к верхней стороне рабочей области фрейма |
| CBRS_ALIGN_BOTTOM | Панель управления привязывается к нижней стороне рабочей области фрейма |
| CBRS_ALIGN_LEFT | Панель управления привязывается к левой стороне рабочей области фрейма |
| CBRS_ALIGN_RIGHT | Панель управления привязывается к правой стороне рабочей области фрейма |
| CBRS_ALIGN_ANY | Панель управления привязывается к какой-либо стороне рабочей области фрейма |
| CBRS_BORDER_TOP | Рамка очерчивает верхний край панели управления, когда она видима |
| CBRS_BORDER_BOTTOM | Рамка очерчивает нижний край панели управления, когда она видима |
| CBRS_ BORDER _LEFT | Рамка очерчивает левый край панели управления, когда она видима |
| CBRS_BORDER _RIGHT | Рамка очерчивает правый край панели управления, когда она видима |
| CBRS_FLOAT_MULTI | В единственном окне мини-фрейма могут располагаться несколько панелей 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 пиксела |
| CCS_NOHILITE | В верхней части элемента управления не надо рисовать разделительную линию шириной 1 пиксел |
| CCS_NOMOVEY | В ответ на сообщение WM_SIZE элемент управления может изменять свои размеры и перемещаться только горизонтально, его вертикальная координата при этом остается постоянной. Не используйте этот стиль совместно со стилем CCS_NOSIZE |
| 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 |

- "Новая", появившаяся в Visual C++ 6.0 и использованная при создании "стандартной" панели инструментов мастером AppWizard:

```
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)
```

В качестве аргументов в наряду с теми же параметрами, что и у предыдущей функции, передаются также два дополнительных: первый — *rcBorders* — задает прямоугольник, определяющий положение и размер панели инструментов (если используются нулевые значения, то размер устанавливается автоматически), второй — *dwCtrlStyle* — позволяет дополнительно задать стили кнопок элемента управления Toolbar, на котором базируется панель инструментов. Этот параметр может иметь следующие значения:

| | |
|---|---|
| TBSTYLE_BUTTON | Стандартная кнопка |
| TBSTYLE_SEP | Разделитель между группами кнопок, представляющий собой маленький промежуток. "Кнопка" с таким стилем не может взаимодействовать с пользователем |
| TBSTYLE_CHECK | Кнопка с фиксацией. Эта кнопка "залипает" при нажатии ее пользователем. 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. Она остается нажатой до тех пор, пока не нажата другая кнопка из этой же группы (TBSTYLE_GROUP \| TBSTYLE_CHECK) |
| TBSTYLE_DROPDOWN | Не документирован #if (_WIN32_IE >= 0x0300) |
| TBSTYLE_AUTOSIZE | Не документирован // automatically calculate the cx of the button#if (_WIN32_IE >= 0x0400) |
| TBSTYLE_NOPREFIX | Не документирован // if this button should not have accel prefix(_WIN32_IE >= 0x0400) |
| TBSTYLE_TOOLTIPS | Панель инструментов может создавать и управлять элементом управления ToolTip (всплывающая подсказка). При использовании этого стиля необходимо самому обрабатывать извещения от этого элемента управления. |
| TBSTYLE_WRAPABLE | Панель инструментов может располагаться в несколько линий. Кнопки переносятся на следующую строку, если им не хватило места в текущей |
| TBSTYLE_ALTDRAG | Не документирован |
| TBSTYLE_FLAT | Не документирован #if (_WIN32_IE >= 0x0300) |
| TBSTYLE_LIST | Не документирован |
| TBSTYLE_CUSTOMERASE | Не документирован |
| TBSTYLE_REGISTERDROP | Не документирован #if (_WIN32_IE >= 0x0400) |
| TBSTYLE_TRANSPARENT | Не документирован #if (_WIN32_IE >= 0x0400) |
| TBSTYLE_EX_DRAWDDARROWS | Не документирован #if (_WIN32_IE >= 0x0400) |

3. Теперь осталось только загрузить из ресурсов образы кнопок. Для этого в классе *CToolBar* предусмотрены две возможности: через вызов функции *LoadToolBar* и путем загрузки образов кнопок при помощи функции *LoadBitmap* с привязкой их к соответствующим командам вызовом функции *SetButtons*. Рассмотрим эти функции более подробно.

**Примечание**

Visual C++, начиная с версии 4.x, поддерживает ресурс типа TOOLBAR, где задаются не только вид кнопок, но также осуществляется автоматическая привязка к ним команд. Поэтому функция *LoadBitmap* приводится скорее для справки, чем для реального использования.

```
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 *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, и 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 определяет каждой кнопке индекс образа. Индекс образа задает его позицию внутри битового образа массива.

```
BOOL CToolBar::LoadToolBar (LPCTSTR lpszResourceName)
```

и

```
BOOL CToolBar::LoadToolBar (UINT nIDResource)
```

Загружает изображение панели инструментов и ресурса, заданного либо именем — *lpszResourceName*, либо идентификатором — *nIDResource*. При этом сопоставление кнопок с образами и командами осуществляется автоматически.

Пример использования этих функций вы найдете в приведенном фрагменте кода. Теперь посмотрим, как к изображениям на кнопках можно добавить текстовые строки:

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
...
  // Создаем панель инструментов рисования
...
  // Добавляем к изображениям на кнопках текстовые строки

  // Прежде всего 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 осчастливлены 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, помятуя, что размер изображений
  // остался без изменений
  // константы 7 — требования Microsoft
  // Точнее, Microsoft требует значений 6 и 7,
  // но внешний вид при этом оставляет желать лучшего
  CRect buttonRect;
  m_wndPaneBar.GetItemRect(0, &buttonRect);
  m_wndPaneBar.SetSizes(CSize(buttonRect.Width(),
                              buttonRect.Height()),
                              CSize(imageRect.Width() — 7,
                                    imageRect.Height() — 7));
...
  return 0;
}
```

Рассмотрим функции, использованные в этом фрагменте.

```
void CToolBar::SetSizes (
  SIZE sizeButton,
  SIZE sizeImage)
```

Устанавливает размер (в пикселах) кнопок панели инструментов, задаваемый параметром *sizeButton*, при этом параметр *sizeImage* должен содержать размер образов, зафиксированный в соответствующем битовом массиве. Размеры кнопок задаются, исходя из размеров образов, плюс 7 пикселов 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.

```
virtual void CToolBar::GetItemRect (
    int nIndex,
    LPRECT lpRect)
```

Заполняет структуру *lpRect* координатами кнопки или разделителя, задаваемых 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.

```
BOOL CToolBar::SetButtonText (
    int nIndex,
    LPCTSRT lpszText)
```

Позволяет сопоставить кнопку (задается идентификатором *nIndex*) панели инструментов и текстовую строку (параметр *lpszText*), которая появится на ней.

После того как панель инструментов создана, необходимо ее где-то расположить. Рассмотрим фрагмент кода:

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
...
  // Создаем панель инструментов рисования
...
  // Заголовок для будущего окна,
  // если панель будет в плавающем режиме.
  // Для этой задачи используется функция класса CWnd
  m_wndPaneBar.SetWindowText(_T("Инструменты"));
  EnableDocking(CBRS_ALIGN_ANY);
  // Сразу настраиваем количество столбцов -
  // эта вспомогательная функция приведена ниже
  SetColumns(m_nPaneCol);
  m_wndPaneBar.EnableDocking(CBRS_ALIGN_ANY);
  DockControlBar(&m_wndPaneBar, AFX_IDW_DOCKBAR_RIGHT);
  // Размещаем панель инструментов в задаваемом месте экрана
  CPoint pt(::GetSystemMetrics(SM_CXSCREEN) - 300,
            ::GetSystemMetrics(SM_CYSCREEN) / 3);
  FloatControlBar(&m_wndPaneBar, pt);
...

  return 0;
}
// Разбивка панели на столбцы - находим
// конечные кнопки каждой строки
void CMainFrame::SetColumns(UINT nColumns)
{
  // Прежде всего определяем число кнопок панели инструментов
  int nCount = m_wndPaneBar.GetToolBarCtrl().GetButtonCount();
  for (int i = 0; i < nCount; i++)
  {
    // Мы собираемся изменять стиль,
    // поэтому запоминаем тот, который был установлен
    UINT nStyle = m_wndPaneBar.GetButtonStyle(i);
```

```
    BOOL bWrap = (((i + 1) % nColumns) == 0);
    if (bWrap)
      nStyle |= TBBS_WRAPPED;
    else
      nStyle &= ~TBBS_WRAPPED;
    // Устанавливаем новый стиль кнопок
    m_wndPaneBar.SetButtonStyle(i, nStyle);
  }
  // Сообщаем, что панель инструментов требует перерисовки
  m_wndPaneBar.Invalidate();
  // Перепозиционируем панель инструментов
  m_wndPaneBar.GetParentFrame()->RecalcLayout();
}
```

Как видите, код небольшой и достаточно простой. Поэтому мне осталось привести только некоторые пояснения по использованным в нем функциям. Начну с метода, который класс *CToolBar* унаследовал от своего базового класса *CControlBar*:

`void CControlBar::`**`EnableDocking`** `(DWORD dwStyle)`

Разрешает привязать панель управления к одной из сторон родительского фрейма, которая должна быть доступна для привязки. Если сторона, передаваемая в качестве параметра *dwStyle*, и сторона, разрешенная для привязки, у фрейма не совпадают, то панель управления не может быть привязана к родительскому фрейму. В качестве этого параметра можно использовать следующие значения: CBRS_ALIGN_TOP, CBRS_ALIGN_RIGHT, CBRS_ALIGN_LEFT, CBRS_ALIGN_BOTTOM, CBRS_ALIGN_ANY и CBRS_ALIGN_MULTI.

Поскольку возможность быть привязанной к родительскому фрейму зависит не только от "желания" самой панели управления, но и от ее родителя — чаще всего фрейма главного окна, здесь же рассмотрим функции класса *CFrameWnd*, которые отвечают за работу с панелями управления.

`void CFrameWnd::`**`EnableDocking`** `(DWORD dwDockStyle)`

Разрешает привязку панели управления к стороне(ам) фрейма, задаваемой(ых) в параметре *dwDockStyle* с возможными значениями CBRS_ALIGN_TOP, CBRS_ALIGN_BOTTOM, CBRS_ALIGN_RIGHT, CBRS_ALIGN_LEFT, CBRS_ALIGN_ANY. По умолчанию панели управления привязываются к сторонам фрейма в следующем порядке: верхняя, нижняя, левая и правая.

```
void CFrameWnd::DockControlBar (
  CControlBar *pBar,
  UINT nDockBarID = 0,
  LPCRECT lpRect = NULL)
```

Привязывает панель управления *pBar* к одной из сторон текущего фрейма, которая задается параметром *nDockBarID* и может быть нулем или комбинацией следующих значений:

| | |
|---|---|
| AFX_IDW_DOCKBAR_TOP | Привязка к верхней стороне фрейма |
| AFX_IDW_DOCKBAR_BOTTOM | Привязка к нижней стороне фрейма |
| AFX_IDW_DOCKBAR_LEFT | Привязка к левой стороне фрейма |
| AFX_IDW_DOCKBAR_RIGHT | Привязка к правой стороне фрейма |

Если параметр *nDockBarID* равен нулю, то сторона может быть любой из числа предварительно определенных при помощи двух функций: *CControlBar::EnableDocking* и *CFrameWnd::EnableDocking*. Параметр *lpRect* определяет в координатах экрана, где вне рабочей области фрейма будет привязано окно, содержащее плавающую панель элементов управления.

```
CFrameWnd* CFrameWnd::FloatControlBar (
  CControlBar *pBar,
  CPoint point,
  DWORD dwStyle = CBRS_ALIGN_TOP)
```

Переводит панель управления *pBar* в плавающее (floating) состояние. Обычно она вызывается при запуске приложения для восстановления предыдущих установок. Кроме того, эта функция вызывается фреймом, когда пользователь после перемещения панели выполняет операцию отпускания (drop), не достигнув места, доступного для привязывания. Параметр *point* задает координаты точки, в которой будет находиться левый верхний угол панели 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 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 управления или когда изменяется размер фрейма. Для того чтобы иметь возможность управлять отображением и поведением панелей управления после изменения компоновки фрейма, необходимо переопределить эту функцию в своем классе. Ее реализация, определенная по умолчанию, вызывает функцию *CWnd::RepositionBars*, чтобы перепозиционировать все панели управления во фрейме.

Еще две функции, которые мы использовали для изменения стиля панели инструментов, выглядят следующим образом.

```
UINT CToolBar::GetButtonStyle (int nIndex)
```

Возвращает стиль кнопки или разделителя, заданных на панели инструментов номером позиции *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 |
| TBBS_WRAP | Кнопка, находящаяся в этом состоянии, сообщает о том, что все следующие кнопки будут отображаться на новой строке. Этот стиль не документирован |

Как уже говорилось, класс *CToolBar* реализует основные функциональные возможности для всех панелей управления. Более специфические и интересные возможности реализованы в его производных классах. Но прежде чем переходить к их рассмотрению, отметим еще один важный момент: в том случае, если кнопка панели управления не имеет обработчиков для UPDATE_COMMAND_UI или COMMAND, то она автоматически блокируется фреймом и выводится серым цветом.

Последняя функция класса *CToolBar*, которую мы рассмотрим, позволяет получить ссылку на объект *CToolBarCtrl*:

```
CToolBar& CToolBar::GetToolBarCtrl ()
```

Позволяет воспользоваться функциональными возможностями общего элемента управления Toolbar, появившегося в Win32. С помощью этого элемента управления можно, в частности, осуществлять более тонкую настройку панели инструментов.

## Класс *CToolBarCtrl*

Для использования преимуществ общего элемента управления 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* (рис. 11.6), вызвав функцию этого класса *CToolBar::CToolBarCtrl*. Чаще всего используют именно второй способ.

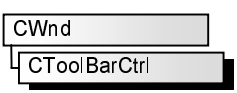



**Рис. 11.6.** Место класса *CToolbarCtrl* в иерархии библиотеки MFC

Любая кнопка панели инструментов имеет ассоциированную с ней структуру TBBUTTON, которая содержит полную информацию об этой кнопке. Определение структуры находится в файле <commctrl.h>:

```
typedef struct _TBBUTTON {
  int iBitmap;
  int idCommand;
  BYTE fsState;
  BYTE fsStyle;
#ifdef _WIN32
  BYTE bReserved[2];
#endif
  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 не видима |
| 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 данного флага позволяет создавать многострочные панели инструментов. Этот флаг можно указывать только вместе с флагом TBSTATE_ENABLED |
| TBSTATE_ELLIPSES | Не документирован. Может использоваться, если определено _WIN32_IE >= 0x0300 |
| TBSTATE_MARKED | Не документирован. Может использоваться, если определено _WIN32_IE >= 0x0400 |

В поле *fsStyle* записывается стиль кнопки. Можно использовать комбинацию стилей, приведенных при описании метода *CToolBar::CreateEx*. Поле *dwData* предназначено для хранения дополнительных данных, связанных 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 занести ее номер во внутреннем списке элемента управления. В противном случае оно должно быть обнулено. Поле *bReserved* зарезервировано для внутреннего использования.

Полную информацию о любой кнопке панели инструментов можно получить при помощи функции *GetButton*.

Достаточно большая группа функций позволяет установить новое или узнать текущее состояние определенной кнопки данного элемента управления. Их действие настолько очевидно, что мы ограничимся только их перечислением: *IsButtonEnabled, IsButtonChecked, IsButtonPressed, IsButtonHidden, IsButtonIndeterminate, GetState, SetState, EnableButton, CheckButton, PressButton, HideButton, Indeterminate*.

Мы использовали доступ к объекту класса *CToolBarCtrl* для определения числа кнопок панели инструментов:

```
int CToolBarCtrl::GetButtonCount ()
```

Функция позволяет узнать общее число кнопок панели инструментов.

Теперь необходимо добавить код, который будет отображать панель инструментов на экране. Для этого создадим элемент меню, как мы это делали в главе 8. Но сначала нужно решить, кто будет отвечать за отображение панели инструментов. Можно возложить эту задачу на дочернее окно, а можно на фрейм главного окна. Для ответа на этот вопрос вспомним, что мы решили каждому компоненту программы поручить свою задачу. Поэтому включим соответствующий пункт в меню, ассоциированное именно с фреймом главного окна.

1. Для добавления кода обработчика команды меню выведем на экран окно мастера ClassWizard (рис. 11.7), выполнив команду View | ClassWizard...

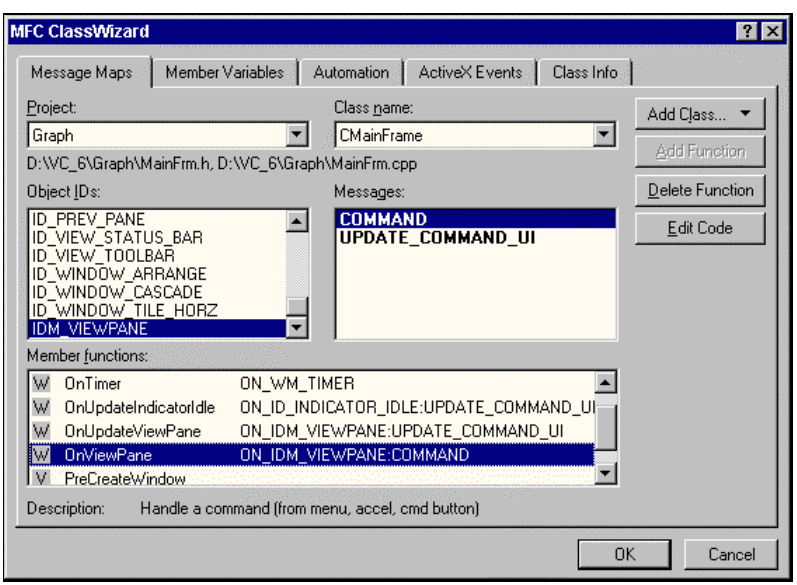



**Рис. 11.7.** Окно мастера ClassWizard

2. В комбинированном списке Class name выберите класс *CMainFrame*, с которым мы будем связывать созданную панель инструментов рисования.
3. В списке Object ID мастера ClassWizard выделите идентификатор созданного элемента меню IDM_VIEWPANE, при этом изменится содержимое списка Messages. Выделите в этом списке COMMAND и нажмите кнопку Add Function.

Мастер автоматически создаст для вас пустой каркас обработчика сообщения, в который мы добавим код, позволяющий отображать и скрывать панель инструментов:

```
void CMainFrame::OnViewPaneBar()
{
  // Проверяем "видимость" панели инструментов
  BOOL bVisible = ((m_wndPaneBar.GetStyle() & WS_VISIBLE) != 0);
  // Меняем видимость на экране
  ShowControlBar(&m_wndPaneBar, !bVisible, FALSE);
  // Перепозиционируем панель инструментов
  RecalcLayout();

// Здесь в будущем мы добавим некоторый код
...
}
```

Единственная неизвестная нам функция выглядит следующим образом:

```
void CFrameWnd::ShowControlBar (
  CControlBar *pBar,
  BOOL bShow,
  BOOL bDelay)
```

Параметр *bShow* определяет видимость панели на экране: TRUE — показать, FALSE — скрыть панель управления *pBar*. Значение параметра *bDelay*, равное TRUE, определяет, что перед выводом на экран нужна задержка.

Остальной код не должен вызвать дополнительных вопросов.

Теперь нужно отметить состояние, для чего необходимо определить обработчик сообщения:

```
void CMainFrame::OnUpdateViewPaneBar(CCmdUI* pCmdUI)
{
  // Проверяем "видимость" панели инструментов
  BOOL bVisible = ((m_wndPaneBar.GetStyle() & WS_VISIBLE) != 0);
  // Устанавливаем или сбрасываем маркер у пункта меню
  pCmdUI->SetCheck(bVisible);
}
```

Ну вот, теперь можно поиграть с созданной панелью инструментов — отображать ее и прятать, переводить в плавающее состояние и привязывать

к любому краю главного фрейма. Мы не можем только получить доступа к ее кнопкам, т. к. они заблокированы. Чтобы изменить такое положение, необходимо добавить код, выполняемый при нажатии на кнопку. К сожалению, ClassWizard нам помочь не может, поскольку вставлять обработчик для каждой кнопки слишком расточительно. Придется реализовывать это вручную. Добавим в карту сообщений класса *CMainFrame* следующие строки:

```
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
  //{{AFX_MSG_MAP(CMainFrame)
...
    // Этот фрагмент нас не интересует,
    // поскольку добавляется мастером ClassWizard
  //}}AFX_MSG_MAP
  ON_COMMAND_RANGE(ID_PANEERASE, ID_PANEOVAL, OnPane)
  ON_UPDATE_COMMAND_UI_RANGE(
                          ID_PANEERASE, ID_PANEOVAL, OnUpdatePane)
END_MESSAGE_MAP()
```

Добавим строки также в класс *CMainFrame*:

```
class CMainFrame : public CFrameWnd
{
...
// Generated message map functions
protected:
...
  //{{AFX_MSG(CMainFrame)
  ...
    // Этот фрагмент нас не интересует,
    // поскольку добавляется мастером ClassWizard
  //}}AFX_MSG
  afx_msg void OnPane(UINT nID);
  afx_msg void OnUpdatePane(CCmdUI* pCmdUI);
  DECLARE_MESSAGE_MAP()
};
```

Теперь осталось только реализовать, собственно, обработчик команд, ассоциированных с каждой кнопкой панели инструментов:

```
void CMainFrame::OnPane(UINT nID)
{
  // nID — идентификатор команды, ассоциированной с кнопкой
  m_nIndex = nID — ID_PANEERASE;  // Порядковый номер кнопки
  // Запоминаем идентификатор нажатой кнопки
  m_wndChild.m_nToolNum = nID;
```

```
  // Сбрасываем все кнопки
  for(int i = ID_PANEERASE; i <= ID_PANEOVAL; i++)
    m_wndPaneBar.GetToolBarCtrl().CheckButton(i, FALSE);
  // Определяем нажатую кнопку
  BOOL bCheck = m_wndPaneBar.GetToolBarCtrl().IsButtonChecked(nID);
  // "Нажимаем" ее
  m_wndPaneBar.GetToolBarCtrl().CheckButton(nID, !bCheck);
...
}
void CMainFrame::OnUpdatePane(CCmdUI* pCmdUI)
{
...
  // Нажимаем или отпускаем кнопку
  pCmdUI->SetCheck((UINT)(ID_PANEERASE + nIndex) == pCmdUI->m_nID);
}
```

Что еще сказать по поводу работы с кнопками панели инструментов? Вроде бы все понятно и так. Поэтому коснусь только изменения формы курсора мыши при нажатии на некоторую кнопку, поскольку мы планируем разработать графический редактор.

Прежде всего создим необходимые курсоры. Рассмотрим этот процесс на одном примере:

1. Раскройте вкладку ResourceView и создайте новый пустой курсор.
2. Нарисуйте внешний вид этого курсора (рис. 11.8).

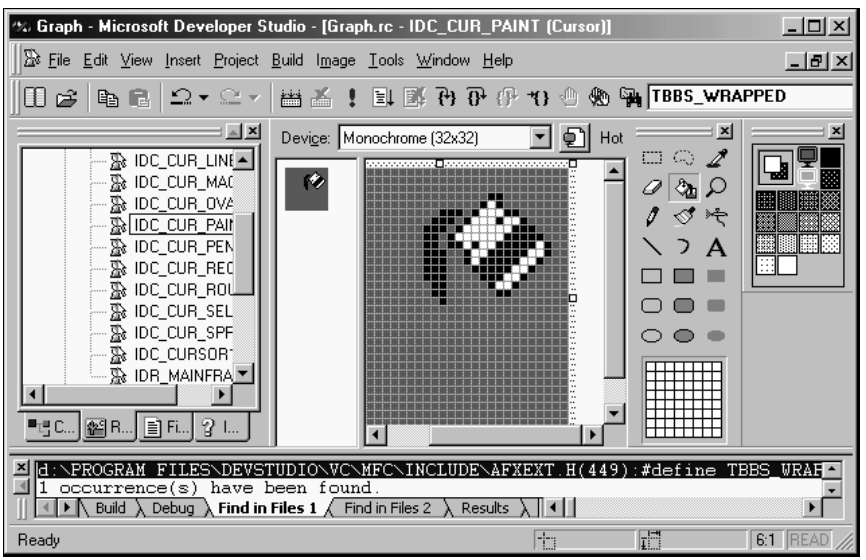



**Рис. 11.8.** Создаем курсор

3. Нажмите кнопку Hot, а затем на ту точку образа курсора, которая будет определять его нулевые, т. е. точку (0, 0), координаты.
4. Раскройте окно свойств курсора и в поле ID введите содержательный идентификатор, например, IDC_CUR_PAINT (рис. 11.9).
5. Аналогичную операцию проделайте для всех остальных курсоров.

**Примечание**

Идентификаторы курсоров необходимо располагать в том же порядке, что и кнопки панели инструментов. Это не требование, а всего лишь пожелание. Если это условие не выполнить, то в приводимые фрагменты кода приложения необходимо будет внести определенные изменения.

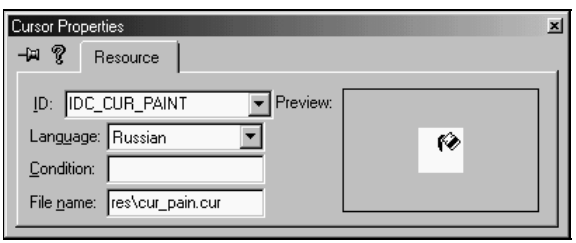



**Рис. 11.9.** Свойства курсоров

Теперь, когда необходимые курсоры созданы, внесем некоторые добавления в рассмотренный ранее код. Но начнем с описания функции API.

```
DWORD::SetClassLong (
  HWND hWnd,
  int nIndex,
  LONG dwNewLong)
```

Устанавливает в структуре WNDCLASS, описывающей оконный класс для *hWnd*, новое значение *dwNewLong* в поле, определяемое 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 класса окна; размер памяти, уже ассоциированной с классом, при этом не изменяется |
| GCL_CBWNDEXTRA | Устанавливает размер (в байтах) дополнительной памяти, предназначенной для использования отдельно каждым окном, создаваемым на базе класса окна; размер памяти, уже ассоциированной с окном, при этом не изменяется |
| GCL_HBRBACKGROUND | Устанавливает дескриптор кисти для фона, ассоциированного классом окна |
| GCL_HCURSOR | Устанавливает дескриптор курсора, ассоциированного классом окна |

| | |
|---|---|
| GCL_HICON | Устанавливает дескриптор пиктограммы, ассоциированной с классом окна; данное значение используется только в Windows 95/98 и не определено в Windows NT |
| GCL_HMODULE | Устанавливает дескриптор модуля, который зарегистрировал класс окна |
| GCL_MENUNAME | Устанавливает адрес строки меню, которая идентифицирует ресурс меню, ассоциированного с классом окна |
| 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 или нуль — при ошибке. Дополнительную информацию об ошибке можно получить, вызвав функцию API *GetLastError*.

Обратную задачу — получение информации из структуры класса окна — выполняет функция

```
DWORD ::GetClassLong (
  HWND hWnd,
  int nIndex).
```

**Примечание**

Получить значение дескриптора *hWnd* для требуемого окна можно с помощью функции *CWnd::GetSafeWnd*.

Мне осталось только привести код:

```
static UINT BASED_CODE cursors[] =
{
  IDC_CUR_ERASE,
  IDC_CUR_PEN,
  IDC_CUR_SELECT,
  IDC_CUR_BRUSH,
  IDC_CUR_SPRAY,
  IDC_CUR_PAINT,
  IDC_CUR_LINE,
  IDC_CUR_EYEDROP,
  IDC_CUR_MAG,
  IDC_CUR_RECT,
  IDC_CUR_ROUND,
  IDC_CUR_OVAL
};
...
```

```
void CMainFrame::OnViewPaneBar()
{
  // Проверяем "видимость" панели инструментов
  BOOL bVisible = ((m_wndPaneBar.GetStyle() & WS_VISIBLE) != 0);
...
  // Проверяем "видимость" панели инструментов и устанавливаем либо
  // текущий для выбранной кнопки курсор, либо "стандартную стрелку"
  if(!bVisible)
    ::SetClassLong(m_wndChild.GetSafeHwnd(),
                   GCL_HCURSOR,
                  (LONG)AfxGetApp()->LoadCursor(cursors[m_nIndex]));
  else
    ::SetClassLong(m_wndChild.GetSafeHwnd(),
                   GCL_HCURSOR,
                  (LONG)AfxGetApp()->LoadStandardCursor(IDC_ARROW));
}
void CMainFrame::OnPane(UINT nID)
{
  // nID — идентификатор команды, ассоциированной с кнопкой
  m_nIndex = nID — ID_PANEERASE;  // Порядковый номер кнопки
...
  // Устанавливаем соответствующий кнопке курсор
  ::SetClassLong(m_wndChild.GetSafeHwnd(),
                 GCL_HCURSOR,
                 (LONG)AfxGetApp()->LoadCursor(cursors[m_nIndex]));
}
```

**Примечание**

В окончательном варианте приложения в основную панель инструментов управления я добавил раскрывающийся список (элемент Combobox). Но эту часть мы рассмотрим, когда познакомимся с элементами управления.

Чтобы закончить с панелями управления, рассмотрим, каким образом можно работать со строкой состояния. Для нее библиотека MFC также содержит два класса: *CStatusBar* и *CStatusBarCtrl*.

# Работа со строкой состояния

Строка состояния (statusbar) — это панель управления, отображающая полосу, которую можно разделить на несколько областей для раздельного вывода в них текста или графической информации. Наиболее часто области используются как строка сообщений (например, для вывода расширенной информации об элементах меню или кнопках панели инструментов) и для отображения индикаторов состояния (например, индикаторов CAPS LOCK, NUM

LOCK и INSERT). В качестве стандартного шрифта для строки состояния используется шрифт "MS Sans Serif" высотой 10 пунктов. Изменение отображения строки состояния осуществляется автоматически благодаря механизму ON_UPDATE_COMMAND_UI: во время цикла простоя (idle time) она 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 разделение этого окна на несколько областей невозможно и выводить в него можно 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 первая область — "эластичная": ее длина определяется длиной области, не используемой всеми другими областями, которые к тому же выравнены 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.

## Класс *CStatusBar*

Класс *CStatusBar* отвечает в библиотеке MFC за реализацию строки состояния, для создания которой, как обычно, необходимы все те же два шага: создать объект класса и создать окно Windows. Рассмотрим их:

1. В нашем примере объект создается в классе *CMainFrame*:

   `CStatusBar::CStatusBar ()`

   Конструктор объекта, который определяет шрифт строки состояния и устанавливает его характеристики в значения, используемые по умолчанию.

```
class CMainFrame : public CFrameWnd
{
// Никто другой, кроме самого фрейма,
// не должен иметь доступа к строке состояния
protected:
  CStatusBar  m_wndStatusBar;
...
};
```

2. Теперь необходимо создать массив идентификаторов областей строки состояния:

```
// Для строки состояния необходимо определить
// массив идентификаторов ее областей
```

```
static UINT indicators[] =
{
  ID_SEPARATOR,
  ID_INDICATOR_NUM,
  ID_INDICATOR_IDLE,
  ID_INDICATOR_TIME
};
```

3. Само окно (строки состояния) создается главным фреймом в ответ на сообщение WM_CREATE:

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
...
  if(!m_wndStatusBar.Create(this) ||
     !m_wndStatusBar.SetIndicators(indicators,
                                   sizeof(indicators)/sizeof(UINT)))
  {
    TRACE0("Ошибка при создании строки состояния\n");
    return -1;
  }
...
}
```

Для создания строки состояния используется функция:

```
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 Windows *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 SBARS_SIZEGRIP. Обратите внимание на последний стиль, при котором правый нижний угол строки состояния принимает вид, показанный на рис. 11.10.

По умолчанию строка состояния располагается в нижней части родительского окна и имеет стиль SBARS_SIZEGRIP. Однако при желании вы можете расположить его и в верхней части, использовав стиль CBRS_TOP. Очевидно, что этот стиль нельзя комбинировать со стилем SBARS_SIZEGRIP, т. к., во-первых, такой элемент управления будет выглядеть очень странно, а во-вторых, он не будет работать так, как ожидает пользователь.

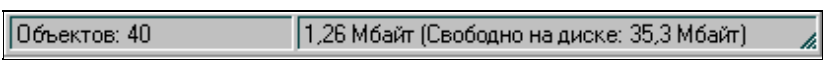



**Рис. 11.10.** Строка состояния, созданная со стилем SBARS_SIZEGRIP

Помимо функции *Create*, в Visual C++ 6.0 реализована еще одна, которая аналогична такой же функции класса *CToolBar*:

```
BOOL CStatusBar::CreateEx(
  CWnd* pParentWnd,
  DWORD dwCtrlStyle = 0,
  DWORD dwStyle = WS_CHILD | WS_VISIBLE | CBRS_BOTTOM,
  UINT nID = AFX_IDW_STATUS_BAR).
```

После того, как объект Windows "Строка состояния" создан, необходимо разбить ее на области. Используем для этого функцию

```
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.

Массив идентификаторов мы уже определили, но, кроме этого, нужны еще соответствующие строковые ресурсы (с целью экономии места привожу фрагмент из файла ресурсов, однако задавать их проще с помощью редактора ресурсов, выбрав создание String Table):

```
STRINGTABLE DISCARDABLE
BEGIN
  ID_INDICATOR_TIME       "00:00:00"
  ID_INDICATOR_IDLE       "•"
END
```

Естественно, существуют и предопределенные идентификаторы. Вот они:

| | |
|---|---|
| ID_INDICATOR_EXT | Расширенный индикатор выбора |
| ID_INDICATOR_CAPS | Индикатор Caps Lock |
| ID_INDICATOR_NUM | Индикатор Num Lock |
| ID_INDICATOR_SCRL | Индикатор Scroll Lock |
| ID_INDICATOR_OVR | Индикатор режима вставки |
| ID_INDICATOR_REC | Индикатор режима записи |
| ID_INDICATOR_KANA | Индикатор Kana Lock |
| ID_SEPARATOR | Разделитель |

Поскольку мы решили отображать в строке состояния текущее время и индикатор времени простоя, то необходимо каким-нибудь способом передать строке состояния эту информацию. Сделать это можно с помощью функции

```
BOOL CStatusBar::SetPaneText (
  int nIndex,
  LPCTSTR lpszNewText,
  BOOL bUpdate = TRUE)
```

Записывает текст, на который указывает *lpszNewText*, в область строки состояния, заданную номером *nIndex*. Если параметр *bUpdate* равен TRUE, то после записи текста область помечается как поврежденная для ее немедленного обновления.

Осталось только получить индекс по известному идентификатору, что можно сделать с помощью метода

```
int CStatusBar::CommandToIndex( UINT nIDFind)
```

Индекс первого идентификатора равен 0.

Последняя функция класса *CStatusBar*, которую мы рассмотрим, служит для получения ссылки на объект *CStatusBarCtrl*:

```
CStatusBarCtrl& CStatusBar::GetStatusBarCtrl ().
```

Это позволит вам воспользоваться функциональными возможностями общего элемента управления Statusbar, включающим, в частности, настройку строки состояния.

В заключение осталось сказать, каким образом 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 командами класса *CCmdUI*, отвечающего за обработку, являются *Enable* и *SetText*. Первая из них блокирует или разблокирует соответствующую область, при этом у заблокированных областей выключается отображение текста. Вторая функция позволяет изменить текст, который выводится в текущей области. Следует иметь в виду, что при выводе текста не происходит автоматического изменения размеров области.

```
// Каждый раз меняем символ в области строки состояния.
// Эта функция при отсутствии каких-либо действий
// вызывается примерно раз в секунду, а при манипуляциях мышью -
// практически постоянно
void CMainFrame::OnUpdateIndicatorIdle(CCmdUI* pCmdUI)
{
  // Обратите внимание, что переменная bIdle — статическая
  static BOOL bIdle = FALSE;
  // Выводим текст
  pCmdUI->SetText(bIdle ? " " : "\x95");
  bIdle = !bIdle;
}
```

**Примечание**

Идентификаторы строковых ресурсов не отображаются в окне мастера ClassWizard. Поэтому все необходимые действия следует производить вручную. Как это сделать, я уже показывал.

Практический пример вывода информации в строку состояния мы рассмотрим после знакомства с таймером. А мы на этом заканчиваем рассмотрение наиболее часто используемых панелей управления — панели инструментов и строки состояния — и переходим к достаточно полезному устройству — таймеру.

# Работа с таймером

Таймер в Windows является устройством, которое периодически извещает приложение об истечении заданного интервала времени. Происходит это путем посылки сообщения WM_TIMER. Таймер присоединяется к приложению при помощи специальной функции основного оконного класса *CWnd*:

```
UINT CWnd::SetTimer(
  UINT nIDEvent,
  UINT nElapse,
  void (CALLBACK EXPORT* lpfnTimer)(HWND, UINT, UINT, DWORD))
```

Первый параметр определяет идентификатор таймера, которым может быть любое целое положительное число. Второй параметр определяет 32-разрядное целое без знака, задающее интервал в миллисекундах, т. е. значение 7000 задает генерацию сообщений WM_TIMER один раз в 7 секунд. И, наконец, последний параметр позволяет определить функцию, которая будет получать сообщения таймера из вашего приложения.

Установить таймер лучше всего в обработчике сообщения WM_CREATE:

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
...
  // Создаем таймер для смены содержимого области строки состояния
  // Таймер будет срабатывать каждую секунду
  SetTimer(0, 1000, NULL);
  return 0;
}
```

Остановить поток сообщений WM_TIMER можно в любой момент, вызвав функцию

```
BOOL CWnd::KillTimer( UINT nIDEvent)
```

Единственным параметром является идентификатор таймера, который необходимо "отключить".

В любом случае перед завершением работы приложения необходимо уничтожить все активные таймеры. И если вы не сделали этого раньше, то используйте сообщение WM_DESTROY, предварительно создав соответствующий обработчик:

```
void CMainFrame::OnDestroy()
{
  // Пусть сначала библиотека сделает все необходимое
  CMDIFrameWnd::OnDestroy();
  // За собой всегда необходимо убирать — удаляем таймер
  KillTimer(0);
}
```

Теперь приведу некоторые пояснения. Начнем с первого параметра функции *SetTimer*. Как вы уже поняли, можно определить несколько таймеров, каждый из которых будет "срабатывать" в "свое" время. Так, например, если необходимо определить два таймера, то это можно сделать следующим образом:

```
// Первый таймер срабатывает один раз в секунду
SetTimer(1, 1000, NULL);
// Второй таймер срабатывает один раз в минуту
SetTimer(2, 60000, NULL);
```

Система Windows не хранит в очереди сообщений несколько сообщений WM_TIMER, а объединяет все такие сообщения в одно. Для того чтобы их разделить, требуется соответствующим образом организовать обработчик сообщения. Создать его очень просто с помощью уже знакомого мастера ClassWizard, в поле Messages которого необходимо выделить сообщение WM_TIMER и нажать кнопку Add Function (рис. 11.11).

Код, помещаемый в этот обработчик, может быть примерно таким:

```
void CMainFrame::OnTimer(UINT nIDEvent)
{
...
  switch(nIDEvent)
  {
    case 1: ...     // Обработка сообщения один раз в секунду
            break;
    case 2: ...     // Обработка сообщения один раз в минуту
            break;
  }
...
}
```

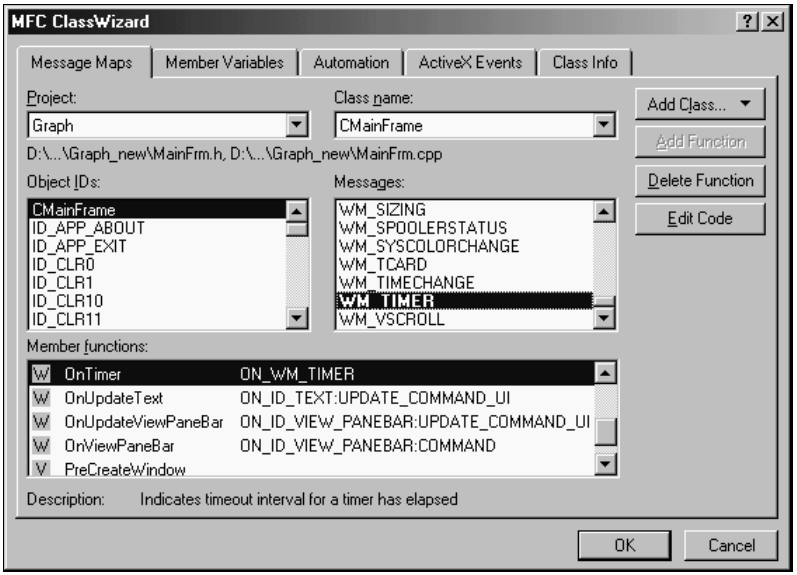



**Рис. 11.11.** Определяем обработчик системного сообщения WM_TIMER

**Примечание**

Если необходимо установить новый интервал времени для существующего таймера, то необходимо сначала уничтожить его, а потом определить снова.

Теперь то, что касается второго параметра. Таймер в Windows имеет ту же разрешающую способность — 54,925 миллисекунды, что и встроенный таймер PC. Поэтому существуют следующие ограничения:

- Сообщения WM_TIMER не могут посылаться чаще, чем 18,2 раза в секунду.
- Временной интервал, который вы задаете, всегда округляется до ближайшего меньшего целого числа, кратного частоте срабатывания таймера. Так, например, интервал в 1000 миллисекунд, разделенный на 54,925 миллисекунды, равен 18.207 срабатываний, которые округляются до 18, что фактически приводит к интервалу 989, а не 1000 миллисекунд.

Учитывайте эти обстоятельства при работе с таймером.

При том способе, который использовали мы, сообщения WM_TIMER посылаются в обычную оконную процедуру, где его можно обработать. Третий параметр функции *SetTimer* позволяет переадресовать такие сообщения в специальную функцию, которая называется функцией "обратного вызова" и вызывается Windows. Она должна иметь определенные параметры:

```
void CALLBACK EXPORT TimerProc(
  HWND hWnd,     // Дескриптор окна, для которого вызывается SetTimer
```

```
  UINT nMsg,    // Сообщение — WM_TIMER
  UINT nIDEvent // Идентификатор таймера
  DWORD dwTime  // Системное время
);
```

На этом мы закончим рассмотрение таймера — полученных сведений достаточно, чтобы отобразить в строке состояния текущее время.

```
void CMainFrame::OnTimer(UINT nIDEvent)
{
  // Отображаем текущее время в строке состояния
  m_wndStatusBar.SetPaneText(
  m_wndStatusBar.CommandToIndex(ID_INDICATOR_TIME), // Получаем индекс
  CTime::GetCurrentTime().Format("%H:%M:%S"));      // Получаем текущее
                                                    // время с помощью
                                                    // класса CTime
  // Передаем дальнейшую обработку в библиотеку
  CMDIFrameWnd::OnTimer(nIDEvent);
}
```

Итак, что мы имеем на данный момент? Прежде всего дочернее окно, в которое можно организовать вывод. Затем панель инструментов рисования, позволяющая выбрать необходимый инструмент, что сигнализируется нажатой кнопкой и изменением формы курсора (рис. 11.12).

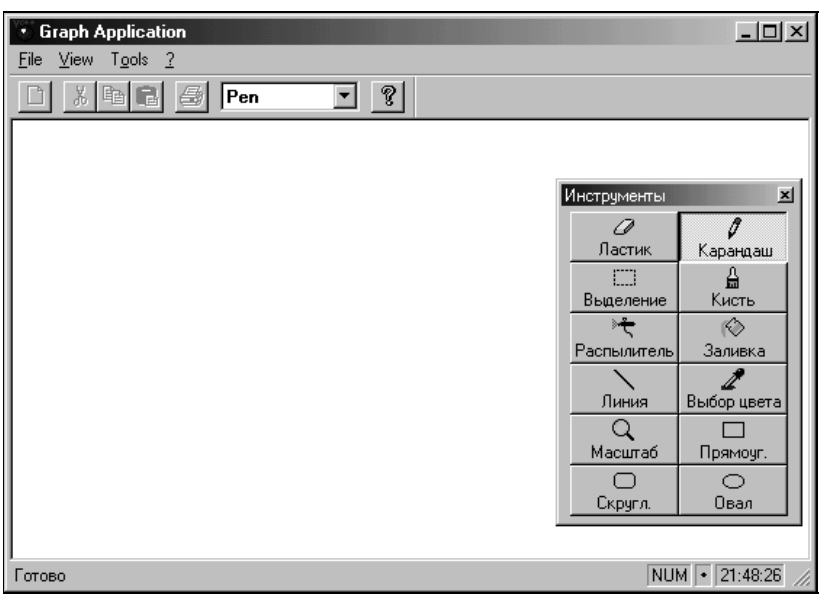



**Рис. 11.12.** Мы ввели панели инструментов и строку состояния

Осталось воспользоваться созданными элементами и научиться рисовать различные фигуры. Для этого нам понадобится организовать графический вывод и научиться работать с мышью. Определенный навык по рисованию в Windows мы уже приобрели, поэтому познакомимся с еще одним устройством — мышью.

# Рисование с помощью мыши

Прежде всего поставим задачу: в зависимости от нажатой кнопки на панели инструментов рисования вычертить соответствующую фигуру в дочернем окне. Всю подготовительную работу для этого мы уже проделали. Поэтому сейчас рассмотрим основные шаги, необходимые для динамического рисования:

1. Когда пользователь щелкает левой кнопкой мыши, приложение должно запомнить координаты *x* и *y* из сообщения WM_LBUTTONDOWN.
2. Когда пользователь двигает мышь по экрану, не отпуская левую кнопку, приложение рисует соответствующую фигуру всякий раз, когда получает сообщение WM_MOUSEMOVE. Причем перед тем, как изобразить новую фигуру, необходимо стереть предыдущую. Размеры новой фигуры вычисляются посредством комбинирования запомненных координат, полученных в сообщении WM_LBUTTONDOWN, с текущими, передающимися в сообщении WM_ MOUSEMOVE.
3. Когда пользователь отпускает левую кнопку мыши, генерируя тем самым сообщение WM_LBUTTONUP, приложение должно сформировать окончательное изображение соответствующей фигуры.

Очевидно, что описанный процесс довольно сильно упрощен, однако главное пока — это прочувствовать основные шаги. Еще раз внимательно прочитайте все необходимые этапы и переходите к деталям. Начнем с кода обработчика сообщения WM_LBUTTONDOWN.

**Примечание**

Надеюсь, что нет необходимости пояснять, где именно мы собираемся обрабатывать сообщения от мыши.

```
void CChildWnd::OnLButtonDown(UINT nFlags, CPoint point)
{
  // Создаем контекст устройства для клиентской области окна
  CClientDC dc(this);
  // Определяем вспомогательные переменные
  COLORREF color(RGB(255, 0, 0));
  CRect lpRect(point.x, point.y, point.x + 12, point.y + 12);
  CBrush brNew(m_color), *oldBrush;
```

```
// Устанавливаем текущий режим рисования
int oldMode = dc.SetROP2(R2_COPYPEN);
// Создаем новый карандаш
CPen  newPen(PS_SOLID, 1, RGB(0, 0, 0)),//DOT
// и выбираем его в контекст устройства,
*oldPen = dc.SelectObject(&newPen);
// а также выбираем кисть
oldBrush = dc.SelectObject(&brNew);
// Стираем старое изображение
dc.Rectangle(m_rcSel);

// В зависимости от выбранного инструмента запоминаем координаты
// нажатия кнопки мыши в соответствующей переменной,
// если это необходимо
switch(m_nToolNum)
{
  case ID_PANEEYEDROP:
    clr = dc.GetPixel(point.x, point.y);
    break;
  case ID_PANEERASE:
    dc.FillSolidRect(&lpRect, clr);
    break;
  case ID_PANESELECT:
    m_rcSel.left = m_rcSel.right  = point.x;
    m_rcSel.top  = m_rcSel.bottom = point.y;
    break;
  case ID_PANEOVAL:
  case ID_PANERECT:
  case ID_PANELINE:
    m_rcSel = CRect(0, 0, 0, 0);
    m_rcCur.left = m_rcCur.right  = point.x;
    m_rcCur.top  = m_rcCur.bottom = point.y;
    break;
  case ID_PANEPEN:
    m_pnDot.x = point.x;
    m_pnDot.y = point.y;
    dc.MoveTo(point.x, point.y);
    dc.LineTo(point.x, point.y);
    break;
  case ID_PANEPAINT:
    dc.SelectObject(&brNew);
    dc.ExtFloodFill(point.x, point.y,
                    dc.GetPixel(point.x, point.y),
                    FLOODFILLSURFACE);
    break;
}
```

```
  // Восстанавливаем старые параметры в контексте устройства
  dc.SelectObject(oldPen);
  dc.SelectObject(oldBrush);
  dc.SetROP2(oldMode);
  // Обязательно (!) организуем "захват" мыши
  SetCapture();
  // Устанавливаем переключатель начала рисования
  m_drawing = TRUE;
  // Передаем управление библиотеке для корректной работы
  CWnd::OnLButtonDown(nFlags, point);
}
```

Основным здесь является запоминание места события и "захват" мыши. С координатами все просто — они передаются в обработчик в качестве параметра *point*. Обращение к функции *CWnd::SetCapture* необходимо, поскольку в процессе рисования фигуры пользователь может вывести мышь за пределы окна, что в данном случае недопустимо.

После того как нажата левая кнопка мыши, приложение начинает улавливать все направляемые сообщения WM_MOUSEMOVE:

```
void CChildWnd::OnMouseMove(UINT nFlags, CPoint point)
{
  // Проверяем, что нажата левая кнопка мыши
  if(m_drawing == TRUE)
  {
    // Получаем контекст устройства
    CClientDC dc(this);
    // Устанавливаем текущий режим рисования
    int oldMode = dc.SetROP2(R2_NOTXORPEN);
    // Также определяем текущие переменные
    CPen  newPen(PS_SOLID, 1, RGB(0, 0, 0)),
         *oldPen = dc.SelectObject(&newPen);
    CPen pen(PS_DOT, 1, RGB(0, 0, 0));
    CRect lpRect(point.x, point.y, point.x + 12, point.y + 12);
    // Выбираем в контекст устройства предопределенную кисть
    CBrush *oldBrush = (CBrush*)dc.SelectStockObject(NULL_BRUSH);
    // Определяем выбранный инструмент рисования
    switch(m_nToolNum)
    {
      case ID_PANEERASE:
        dc.FillSolidRect(&lpRect, clr);
        break;
      case ID_PANEOVAL:
        // Стираем старое изображение
        dc.Ellipse(m_rcCur);
```

```
      // Фиксируем новые координаты
      m_rcCur.right  = point.x;
      m_rcCur.bottom = point.y;
      // Рисуем эллипс
      dc.Ellipse(m_rcCur);
      break;
    case ID_PANELINE:
      // Стираем старое изображение
      dc.MoveTo(m_rcCur.left, m_rcCur.top);
      dc.LineTo(m_rcCur.right, m_rcCur.bottom);
      m_rcCur.right  = point.x;
      m_rcCur.bottom = point.y;
      // Рисуем новую линию
      dc.MoveTo(m_rcCur.left, m_rcCur.top);
      dc.LineTo(m_rcCur.right, m_rcCur.bottom);
      break;
    case ID_PANESELECT:
      // В режиме выбора прямоугольник будем рисовать пунктирной
      // линией, поэтому переопределяем текущий карандаш
      dc.SelectObject(&pen);
      // Стираем старое изображение
      dc.Rectangle(m_rcSel);
      m_rcSel.right  = point.x;
      m_rcSel.bottom = point.y;
      // Рисуем новое изображение
      dc.Rectangle(m_rcSel);
      break;
    case ID_PANERECT:
      // Стираем старое изображение
      dc.Rectangle(m_rcCur);
      m_rcCur.right  = point.x;
      m_rcCur.bottom = point.y;
      // Рисуем новый прямоугольник
      dc.Rectangle(m_rcCur);
      break;
    case ID_PANEPEN:
      dc.SetROP2(R2_COPYPEN);
      dc.MoveTo(m_pnDot.x, m_pnDot.y);
      m_pnDot.x = point.x;
      m_pnDot.y = point.y;
      dc.LineTo(point.x, point.y);
      break;
  }
  // Восстанавливаем карандаш, кисть и режим рисования
  dc.SelectObject(oldPen);
  dc.SelectObject(oldBrush);
```

```
    dc.SetROP2(oldMode);
  }
  // Передаем управление библиотеке для корректной работы
  CWnd::OnMouseMove(nFlags, point);
}
```

В первой строке этого обработчика используется один из возможных способов проверки, нажата ли левая кнопка мыши. Если кнопка не нажата — сообщение игнорируется. В противном случае обработчик получает контекст устройства, устанавливает параметры карандаша, кисти и режима рисования, определяет выбранный инструмент, стирает предыдущее изображение, запоминает текущие размеры фигуры, рисует ее и освобождает контекст устройства.

Новым здесь является установка текущего режима рисования R2_NOTXORPEN с помощью функции *CDC::SetROP2*, которую мы рассмотрели в главе 9.

Используемая при этом логическая операция выбрана потому, что в этом случае, рисуя старую фигуру непосредственно по исходному изображению, мы тем самым эффективно стираем ее. Затем фигура рисуется еще раз, но уже с новыми размерами. Весь этот процесс повторяется снова и снова (с невероятной скоростью), пока пользователь не отпустит левую кнопку мыши:

```
void CChildWnd::OnLButtonUp(UINT nFlags, CPoint point)
{
  CClientDC dc(this);
  int oldMode = dc.SetROP2(R2_COPYPEN);
  CPen  newPen(PS_SOLID, 1, RGB(0, 0, 0)),
       *oldPen = dc.SelectObject(&newPen);
  // Выбираем в контекст устройства предопределенную кисть
  CBrush *oldBrush = (CBrush*)dc.SelectStockObject(NULL_BRUSH);
  // Определяем выбранный инструмент рисования
  switch(m_nToolNum)
  {
    case ID_PANEOVAL:
      // Стираем старое изображение
      dc.Ellipse(m_rcCur);
      // Фиксируем новые координаты
      m_rcCur.right  = point.x;
      m_rcCur.bottom = point.y;
      // Рисуем эллипс
      dc.Ellipse(m_rcCur);
      break;
    case ID_PANEPEN:
      // Здесь нам ничего делать не надо, т. к. в момент отпускания
      // левой кнопки мыши новая точка не появляется
      break;
```

```
    case ID_PANELINE:
      // Стираем старое изображение
      dc.MoveTo(m_rcCur.left, m_rcCur.top);
      dc.LineTo(m_rcCur.right, m_rcCur.bottom);
      // Фиксируем новые координаты
      m_rcCur.right  = point.x;
      m_rcCur.bottom = point.y;
      // Рисуем линию
      dc.MoveTo(m_rcCur.left, m_rcCur.top);
      dc.LineTo(m_rcCur.right, m_rcCur.bottom);
      break;
    case ID_PANERECT:
      // Аналогичная последовательность действий для прямоугольника
      dc.Rectangle(m_rcCur);
      m_rcCur.right  = point.x;
      m_rcCur.bottom = point.y;
      // Рисуем пунктирный прямоугольник
      dc.SelectObject(&newPen);
      dc.Rectangle(m_rcCur);
      break;
  }
  // Восстанавливаем карандаш, кисть и режим рисования
  dc.SelectObject(oldPen);
  dc.SelectObject(oldBrush);
  dc.SetROP2(oldMode);
  // Сбрасываем флаг рисования
  m_drawing = FALSE;
  // Освобождаем мышь
  ReleaseCapture();

  // Передаем управление библиотеке для корректной работы
  CWnd::OnLButtonUp(nFlags, point);
}
```

Этот код проверяет выбранный инструмент рисования и рисует окончательное изображение, после чего сбрасывает флажок, говоря о том, что пользователь решил прекратить рисование, и освобождает "захваченную" мышь.

Как видите, если рассматривать процесс последовательно, то он не кажется сложным. Поэтому просто еще раз повторю все шаги, чтобы вы четко запомнили их:

- запомнить, где произошло событие WM_LBUTTONDOWN;
- постоянно рисовать фигуру при получении сообщения WM_MOUSEMOVE;
- начертить окончательную фигуру, получив сообщение WM_LBUTTONUP.

Очевидно, что рассмотренный пример далеко не закончен, фактически это только небольшая заготовка для дальнейшей работы, и я предлагаю вам в качестве упражнения довести ее до конца самостоятельно.

# ГЛАВА 12



# В глубине меню

Чем хуже готовят повара,
тем вежливей должны быть официанты.
*Михаил Генин*

Что такое меню, как оно создается и как с ним работать? Начальные сведения уже были приведены в главе 4, здесь же мы постараемся осветить эти вопросы более подробно. Однако, прежде чем переходить к изучению соответствующих 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 под полосой заголовка в верхней части перекрывающегося или всплывающего окна и представляет собой словесные или символические опции, дающие возможность манипулировать особенностями программы. Такое меню называется *Главным меню приложения (*верхний уровень иерархии). Отдельные элементы меню располагаются в *полосе меню* (menu bar) — рис. 12.1 и 12.2.

Элементы меню могут иметь различные типы и содержать ряд модификаторов (рис. 12.3):

- командная клавиша — вместо выбора пункта меню команду можно вызвать с помощью этой клавиши;
- 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 командной клавиши (комбинации клавиш), куда входит эта буква. Такие буквы называются мнемоническими.

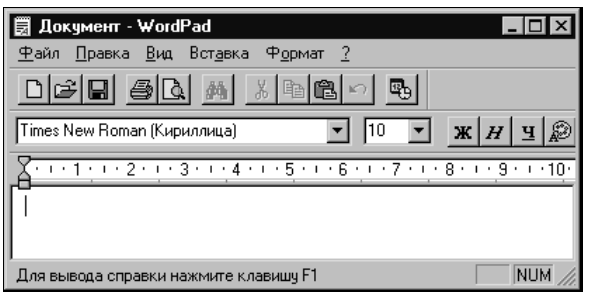



**Рис. 12.1.** Заголовок и Главное меню приложения WordPad

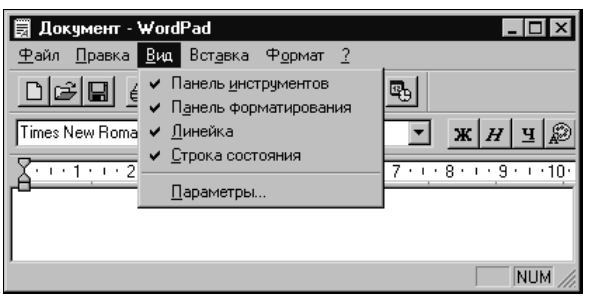



**Рис. 12.2.** Раскрытое меню пункта "Вид"

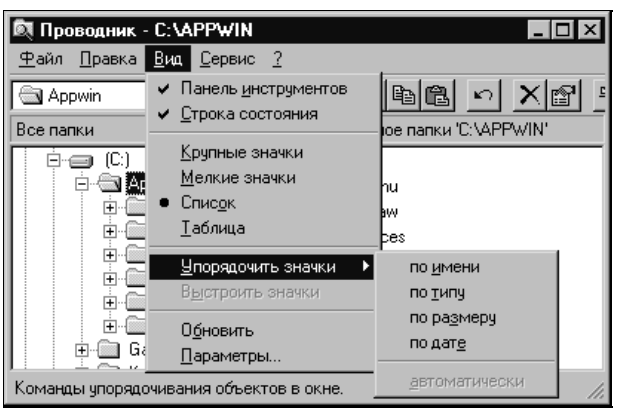



**Рис. 12.3.** Команда, имеющая подменю

Каждое окно, имеющее заголовок, может иметь *системное* меню, которое вызывается щелчком левой кнопки мыши на пиктограмме, расположенной в левой части заголовка окна (рис. 12.4), и представляет собой набор стандартных команд. В Windows 95 системное меню появляется, если щелкнуть правой кнопкой мыши в полосе заголовка или на пиктограмме свернутого окна.




**Рис. 12.4.** Системное меню

Система Windows предоставляет специальный тип меню, которое можно создавать 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, пиктограммой. Контекстное меню в некоторых случаях удобнее других, т. к. его содержимое зависит от того, для какого объекта оно было создано. Обычно оно выводится на экран щелчком правой кнопки мыши.




**Рис. 12.5.** Контекстное меню рабочей области окна приложения WordPad

Теперь, перечислив основные типы используемых в Windows меню, рассмотрим те 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 для них адекватные альтернати-

вы, реализованные в виде класса *CMenu* (рис. 12.6), полностью вобравшего в себя функции API.

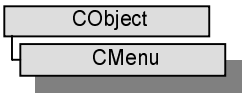



**Рис. 12.6.** Место класса *CMenu* в иерархии библиотеки MFC

Класс *CMenu* 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 меню. Следует совершенно четко представлять, что при создании объекта не происходит автоматического создания самого меню. 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 и 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 *nID*.

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 ()
```

Удаляет меню перед завершением работы приложения и возвращает системе занимаемые им ресурсы.

Для изменения состава меню, созданного функцией *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)
BOOL CMenu::AppendMenu (
    UINT nFlags,
    UINT nIDNewItem = 0,
    const CBitmap *pBmp)
```

Функции 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 доступен (MF_ENABLED) или блокирован (MF_DISABLED), но изображается обычным (черным) цветом |
| 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
- 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-битное 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)
```

Функции 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 *nPosition*:

| nFlags | nPosition |
|---|---|
| MF_BYCOMMAND | Определяет идентификатор элемента меню, перед которым вставляется новый элемент |

*(окончание)*

| nFlags | nPosition |
|---|---|
| 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)
```

Функции удаляют элемент меню, задаваемый параметром *nPosition*, значение которого интерпретируется так же, как в функции *InsertMenu*. Отличие 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 (задается параметром *nEnable*): доступное (MF_ENABLED), запрещенное

(MF_DISABLED) или заблокированное (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 отметку с помощью следующих 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 сбрасывая метку у 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 битовый массив:

```
BOOL CMenu::SetMenuItemBitmaps (
    UINT nPosition,
    UINT nFlags,
    const CBitmap *pBmpUnchecked,
    const CBitmap *pBmpChecked)
```

Функция 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 либо *pBmpChecked*, либо *pBmpUnchecked* равен NULL, то для соответствующего атрибута рядом с элементом ничего не отображается, а если оба параметра равны NULL, то используются стандартные галочка и кружок. При использовании функции следует иметь в виду, что при разрушении меню не происходит автоматического удаления битовых массивов — эта задача возлагается на приложение.

После внесения в меню всех изменений необходимо вызвать функцию

```
void CWnd::DrawMenuBar ()
```

Производит перерисовку полосы 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 выбором 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 задания расположения меню 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 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 *lpRect* 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: *MeasureItem* и *DrawItem*. Но они заслуживают отдельного разговора и мы к ним вернемся в разделе "Самоотображение элементов меню" данной главы.

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:

1. На основе шаблона, задаваемого в файле ресурса.
2. Динамическое создание при помощи специальных функций.
3. На основе шаблона в оперативной памяти.

Рассмотрим каждый из этих способов более подробно.

# Создание меню на основе шаблона

Этот подход мы уже рассматривали и поэтому не будем останавливаться на нем более подробно, а приведем только описание атрибутов, определяющих внешний вид и поведение строки меню (рис. 12.7, табл. 12.1).

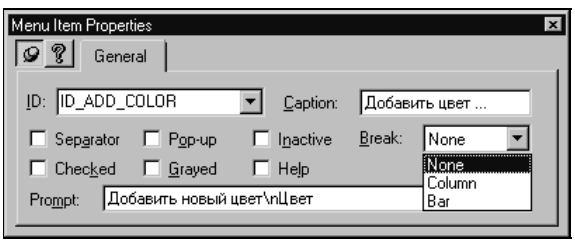



**Рис. 12.7.** Атрибуты меню в окне свойств, открытом в режиме настройки параметров меню

***Таблица 12.1.*** *Описание атрибутов, определяющих внешний вид 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 |
| 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 |

***Таблица 12.1*** *(окончание)*

| Атрибут | Описание |
|---|---|
| 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, необходимо подключить меню к окну приложения. Этот вопрос также достаточно подробно рассматривался, 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,          // идентификатор меню
                    RUNTIME_CLASS(CNoteDoc),
                    RUNTIME_CLASS(CNoteFrame),
                    RUNTIME_CLASS(CNoteView));
```

**Примечание**

Мы обязательно рассмотрим архитектуру "документ/представление" более подробно.

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. В табл. 12.2 приведены этапы работы с меню и возникающие при этом сообщения.

***Таблица 12.2.*** *Этапы работы с меню и возникающие при этом сообщения*

| Этап работы | Сообщение | Обработчик 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 роль перечисленных сообщений.

**Примечание**

Действие приведенных обработчиков справедливы 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 определенным приложением окна, связанного с командной клавишей |
| SC_HSCROLL | Прокрутка по горизонтали |
| SC_KEYMENU | Выбор элемента меню при помощи командной клавиши |
| SC_MAXIMIZE или SC_ZOOM | Разворачивание окна |
| SC_MINIMIZE или SC_ICON | Сворачивание окна |

| | |
|---|---|
| SC_MOUSEMOVE | Выбор элемента меню 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 |

**Примечание**

Кроме перечисленных, можно использовать и свои идентификаторы, как будет показано в рассматриваемом ниже примере.

Второй параметр *lParam* используется только при следующих значениях первого параметра *nID*:

| nID | lParam |
|---|---|
| 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) система 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)
```

Параметр *pMenu* 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, система Windows, прежде чем отобразить всплывающее меню, посылает владельцу сообщение WM_INITMENUPOPUP. Библиотека MFC вызывает обработчик, который дает приложению возможность изменить всплывающее меню перед его отображением:

```
void CWnd::OnInitMenuPopup (
    CMenu pPopupMenu,
```

```
    UINT nIndex,
    BOOL bSubMenu)
```

Параметр *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)
```

Параметр *nItemID* — идентификатор выделенного элемента или индекс всплывающего меню. Параметр *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 связывается всплывающее меню |
| MF_SEPARATOR | Элемент является разделителем |
| MF_SYSMENU | Элемент принадлежит системному меню |

Последний параметр *hSubMenu* используется в двух случаях, зависящих от состояния параметра *nFlags*:

| nFlags | hSubMenu |
|---|---|
| MF_SYSMENU | Идентифицирует меню, ассоциированное с сообщением |
| MF_POPUP | Идентифицирует дескриптор меню верхнего уровня |

Многие приложения используют сообщение WM_MENUSELECT 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 продемонстрировано в главе 11.

С учетом новых сведений покажем, как можно организовать "переключение" количества столбцов панели инструментов рисования в нашем приложении Graph:

1. Добавить в уже имеющуюся полосу меню элемент Tools, а в качестве его подпунктов — "2 column" и "3 column" (рис. 12.8).

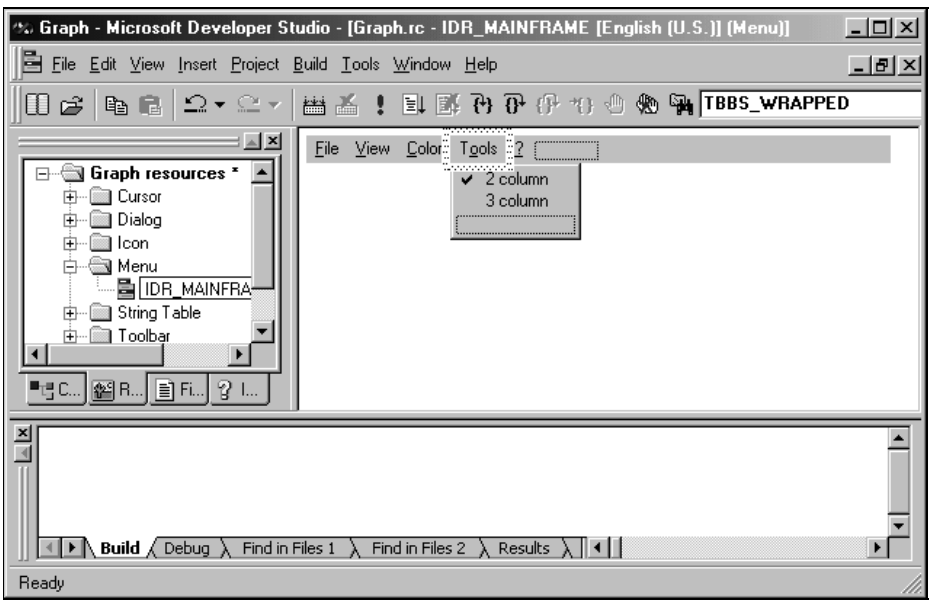



**Рис. 12.8.** Добавляем новые элементы меню

2. Внести приведенный ниже код в карту сообщений класса *CMainFrame*:

```
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
  //{{AFX_MSG_MAP(CMainFrame)
  ...
  //}}AFX_MSG_MAP
  ON_COMMAND_RANGE(ID_TOOLS_2COLUMN, ID_TOOLS_3COLUMN,
                   OnToolsColumn)
  ON_UPDATE_COMMAND_UI_RANGE(ID_TOOLS_2COLUMN, ID_TOOLS_3COLUMN,
                             OnUpdateToolsColumn)
END_MESSAGE_MAP()
```

3. В описание класса *CMainFrame* добавить определение обработчиков:

```
//{{AFX_MSG(CMainFrame)
...
//}}AFX_MSG
afx_msg void OnToolsColumn(UINT nID);
afx_msg void OnUpdateToolsColumn(CCmdUI* pCmdUI);
```

4. И, наконец, написать код самих обработчиков:

```
void CMainFrame::OnToolsColumn(UINT nID)
{
  // Запоминаем выбранное число столбцов
  m_nPaneCol = (nID == ID_TOOLS_2COLUMN) ? 2 : 3;
  // Устанавливаем столбцы на панели инструментов рисования
  SetColumns(m_nPaneCol);
}

void CMainFrame::OnUpdateToolsColumn(CCmdUI* pCmdUI)
{
  // Определяем выбранный пункт меню
  UINT nCol = (pCmdUI->m_nID == ID_TOOLS_2COLUMN) ? 2 : 3;
  // Устанавливаем маркер состояния — галочку у выбранного
  // пункта меню и сбрасываем у остальных
  if(m_nPaneCol == nCol)
    pCmdUI->SetCheck(TRUE);
  else
    pCmdUI->SetCheck(FALSE);
}
```

Как видите, все довольно просто и, на мой взгляд, нет надобности больше задерживаться на этом вопросе: ведь при использовании шаблонов подавляющую часть работы выполняет операционная система, а все действия, необходимые для работы с меню, представляют относительную сложность только при построении самого первого меню.

Поэтому мы переходим к практическим примерам. Как вы помните, мы еще не закончили наше приложение Graph. Здесь мы добавим в него возможность выбора цвета как для рисования линий, так и для заливки. Но сделаем это, как и договаривались, не общепринятым способом, а таким, который позволит проиллюстрировать максимум возможностей работы с меню.

## Добавление элемента в системное меню

Итак, мы создаем контекстное меню, появляющееся при нажатии правой кнопки мыши в клиентской области окна и позволяющее выбрать цвет за-

ливки (да и текущий цвет линий). Одновременно в системное меню добавим пункт, разрешающий работу с этим контекстным меню (рис. 12.9).

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
  ...
  // Получаем доступ к системному меню
  CMenu* pSysMenu = GetSystemMenu(FALSE);
  // Определяем текст, которым будет представлен
  // созданный элемент системного меню
  CString strColorMenu("Разрешить смену текущего цвета");
  // Добавляем в него разделитель и новый элемент
  // "Разблокировать меню Цвет"
  pSysMenu->AppendMenu(MF_SEPARATOR);
  pSysMenu->AppendMenu(MF_BYCOMMAND | MF_STRING,
                       IDM_COLOR, strColorMenu);
  ...
}
```

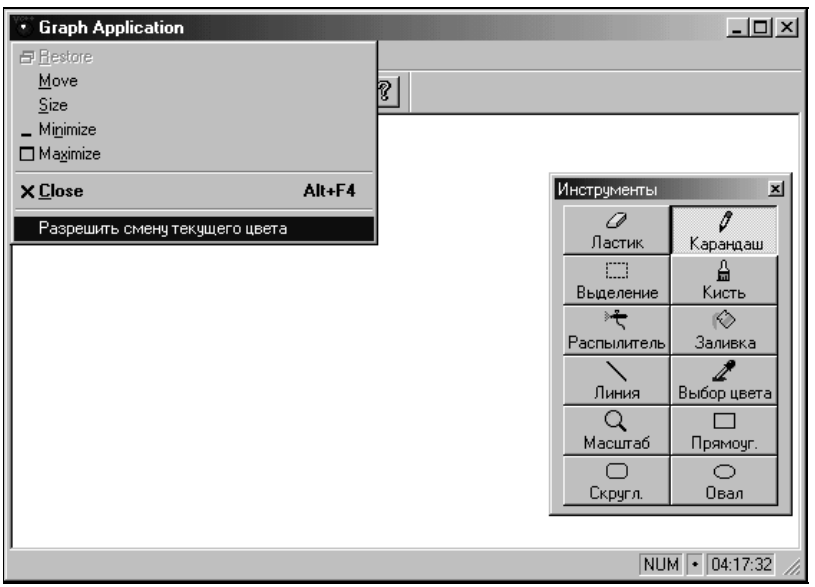



**Рис. 12.9.** Добавляем новый пункт в системное меню

Давайте коротко разберем наши действия. Прежде всего необходимо получить доступ к системному меню, что мы и делаем с помощью функции

`CMenu* CWnd::GetSystemMenu(BOOL bRevert)`

Позволяет получить идентификатор копии системного меню, так что с ним можно проводить любые допустимые манипуляции. При добавлении своего элемен-

та в системное меню необходимо учитывать, что его идентификаторы имеют номера, начинающиеся с 0xF000. Поэтому собственный идентификатор должен иметь номер, меньший этого значения. Кроме того, в качестве последней цифры шестнадцатеричного представления номера команды рекомендуется использовать 0. Параметр *bRevert* задает режим работы функции: если он равен FALSE, можно выполнять модификацию системного меню. Значение TRUE "возвращает" системное меню в состояние, приписанное ему по умолчанию.

Следующая строка кода

```
CString strColorMenu("Разрешить смену текущего цвета");
```

задает текст, который должен появиться в системном меню. Хотя его длина не оговаривается, все же не надо делать его слишком длинным. Но в любом случае система Windows сама "подгонит" размеры этого меню под максимальный текст.

Нам осталось только добавить те элементы меню, которые нам необходимы. Это, безусловно, разделитель и собственно текст нашего меню. Выполняется это очень просто с помощью функции *CMenu::AppendMenu*.

Здесь необходимы некоторые пояснения. Как вы помните, при создании меню с помощью редактора ресурсов мы обязательно должны были присвоить каждому его элементу идентификатор, который является ничем иным, как некоторым целым числом. Так вот, редактор ресурсов сам назначает числовой эквивалент введенному идентификатору. Совсем по-другому обстоит дело в данном случае. При динамическом добавлении элемента меню мы *самостоятельно* должны сопоставить идентификатор с некоторым цифровым эквивалентом. Сделать это можно, непосредственно редактируя файл <resource.h>, а можно с помощью IDE. Именно такой подход мы сейчас и рассмотрим:

1. Выберите пункт меню View | Resource symbols и на экране появится одноименный диалог (рис. 12.10), в котором вы найдете все имеющиеся идентификаторы.

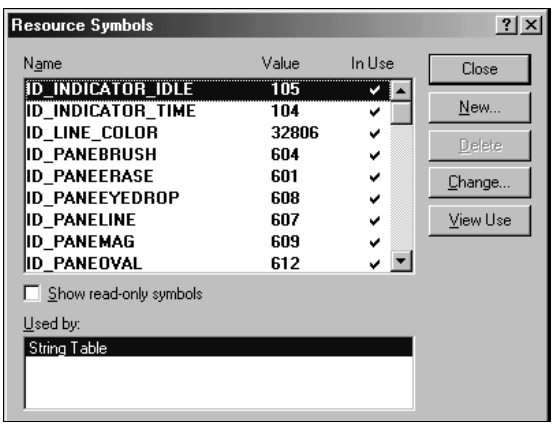



**Рис. 12.10.** Здесь можно получить информацию по имеющимся идентификаторам

2. Нажмите кнопку New, чтобы создать новый идентификатор. На экране появится диалог New Symbol.
3. В поле Name введите обозначение идентификатора, а в поле Value — его цифровое значение (рис. 12.11).
4. Нажмите последовательно кнопки OK и Close в соответствующих диалогах — новый идентификатор создан.

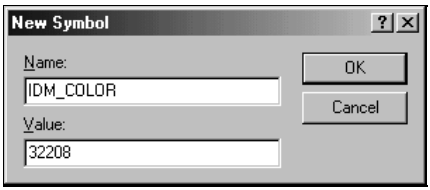



**Рис. 12.11.** Создаем новый идентификатор

Вот, собственно, и все. Теперь можно смело вставлять новый элемент в системное меню и создавать для него соответствующий обработчик.

**Примечание**

Некоторые из представленных в блоке диалога Resource Symbols идентификаторы не снабжены галочкой в столбце In Use. И в поле Used by для них выводится загадочная надпись "not used". Естественно, это не означает, что данный идентификатор не нужен. Просто ClassWizard о нем ничего не знает.

Обработка команд системного меню несколько отличается от стандартной, и об этом необходимо сказать несколько слов. Я уже отмечал, что когда пользователь выбирает элемент системного меню, вызывается обработчик сообщения WM_SYSCOMMAND. Следовательно, наша задача — реализовать этот обработчик. Очевидно, что делать это мы будем с помощью мастера ClassWizard, который создаст необходимые компоненты карты сообщений и заготовку обработчика:

```
class CMainFrame : public CFrameWnd
{
...
public:
  BOOL m_bEnable;          // индикатор включения контекстного меню
protected:
...
  //{{AFX_MSG(CMainFrame)
...
  afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
  //}}AFX_MSG
  ...
};
...
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
```

```
//{{AFX_MSG_MAP(CMainFrame)
...
  ON_WM_SYSCOMMAND()
//}}AFX_MSG_MAP
...
END_MESSAGE_MAP()
...
void CMainFrame::OnSysCommand(UINT nID, LPARAM lParam)
{
  // Прежде всего отсекаем команды несистемного меню
  if((nID & 0xFFF0) == IDM_COLOR)
  {
    // Получаем доступ к системному меню
    CMenu *pSysMenu = GetSystemMenu(FALSE);
    // Определяем текущее состояние элемента
    UINT nState = pSysMenu->GetMenuState(IDM_COLOR, MF_BYCOMMAND);
    // Если элемент отмечен, ...
    if(nState == MF_CHECKED)
    {
      // сбрасываем маркер, ...
      pSysMenu->CheckMenuItem(IDM_COLOR,
                              MF_BYCOMMAND | MF_UNCHECKED);
      // разрешаем работу с контекстным меню
      m_bEnable = FALSE;
    }
    else // и наоборот
    {
      pSysMenu->CheckMenuItem(IDM_COLOR, MF_BYCOMMAND | MF_CHECKED);
      // запрещаем работу с контекстным меню
      m_bEnable = TRUE;
    }
  }
  else
  {
    // Если команда "не наша", необходимо передать ее
    // библиотеке для корректной обработки
    CWnd::OnSysCommand(nID, lParam);
  }
}
```

В приведенном фрагменте обратите внимание на строчку

```
BOOL m_bEnable;          // индикатор включения контекстного меню
```

которая включена для того, чтобы как-то оправдать задачу создания новой команды системного меню. Значение TRUE этой переменной разрешит нам создать и работать с контекстным меню, а FALSE, соответственно, запретит.

## Создание контекстного меню

Для решения поставленной задачи нам осталось выбрать команду, по которой будет создаваться контекстное меню, и реализовать ее обработчик. Обычно контекстные меню выводятся на экран при нажатии на правую кнопку мыши. Мы не будем отступать от этого правила.

Создайте обработчик сообщения WM_RBUTTONDOWN. Не забудьте, что помимо самой функции необходимо определить компоненты карты сообщений.

```
void CChildWnd::OnRButtonDown(UINT nFlags, CPoint point)
{
  // Создавать и отображать контекстное меню будем только
  // в том случае, если "получили разрешение" от системного меню
  if(((CMainFrame *)GetParentFrame())->m_bEnable)
  {
    // Поскольку мы создаем меню, нам нужен соответствующий объект
    CColorMenu menu;
    // Создаем контекстное меню
    menu.CreatePopupMenu();
    for(int i = 0; i < 16; i++)
    {
      // Добавляем в него элементы
      menu.AppendMenu(MF_OWNERDRAW, ID_CLR0 + i, "");
    }
    // Для расположения контекстного меню относительно экрана
    // необходимо преобразовать положение курсора мыши
    // в экранные координаты
    ClientToScreen(&point);
    // Выводим контекстное меню на экран
    menu.TrackPopupMenu(TPM_LEFTALIGN | TPM_RIGHTBUTTON,
                        point.x, point.y, this);
  }
  // Как всегда, не мешаем нормальной работе
  CWnd::OnRButtonDown(nFlags, point);
}
```

Первое, что бросается в глаза — это появление нового класса *CColorMenu*, о котором мы ничего не говорили. Да и не могли ничего сказать, потому что только приступаем к его созданию. До сих пор мы пользовались объектами класса *CMenu*, и этого было вполне достаточно. Зачем же нам понадобился еще один класс? Все очень просто — мы хотим взять на себя задачу отрисовки элементов меню. Этим мы займемся несколько позднее, а пока будем считать, что работаем с объектом класса *CMenu*.

Итак, после того как объект соответствующего класса создан, можно создавать собственно контекстное меню:

```
menu.CreatePopupMenu();
```

Однако пока в нем нет ни одного элемента. Чтобы исправить такое положение, добавим их :

```
for(int i = 0; i < 16; i++)
  menu.AppendMenu(MF_OWNERDRAW, ID_CLR0 + i, "");
```

Следующим шагом является преобразование положения курсора мыши в экранные координаты

```
ClientToScreen(&point);
```

чтобы место, где мы щелкнули правой кнопкой мыши, соответствовало левому верхнему углу нашего контекстного меню. Если мы этого не сделаем, все будет работать по-прежнему, но выглядеть не очень аккуратно.

**Примечание**

Эта точка может соответствовать и левому нижнему углу контекстного меню, но такое перестроение автоматически выполняет сама система Windows.

После того как мы все подготовили, осталось только вывести созданное меню на экран. Что и делает, как вы помните, функция

```
menu.TrackPopupMenu(TPM_LEFTALIGN | TPM_RIGHTBUTTON,
                    point.x, point.y, this);
```

Теперь выбор любого элемента нашего меню приведет к тому, что будет вызван соответствующий обработчик, который, правда, мы еще не создали. Проделайте это самостоятельно либо найдите его в полном тексте приложения Graph, который имеется на сопроводительной дискете. А мы будем двигаться дальше.

## Самоотображение элементов меню

Для того чтобы создать меню, внешний вид которого отличается от стандартного (например, кроме текста мы хотим вывести там еще и пиктограмму), нам необходимо перехватить и обработать два сообщения: WM_MEASUREITEM и WM_DRAWITEM. Библиотека MFC уже подготовила все для того, чтобы этот процесс не занял много времени, и реализовала в классе *CMenu* (и не только в нем) следующие две функции:

```
virtual void CMenu::MeasureItem (LPMEASUREITEMSTRUCT lpMeasureItemStruct)
```

Вызывается библиотекой MFC при создании меню, имеющего 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 при создании меню, имеющего 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.

Таким образом, прежде всего нам необходимо создать свой класс, производный от *CMenu*. Рассмотрим один из возможных вариантов:

1. Раскроем окно мастера ClassWizard и выполним команду New раскрывающейся кнопки Add Class.
2. В появившемся диалоге введем имя создаваемого класса в поле Name.
3. К сожалению, ClassWizard не позволяет создавать класс на основе CMenu. Поэтому поступаем следующим образом: в раскрывающемся списке Base class выбираем какой-нибудь класс, например, *CButton* (рис. 12.12). Нажимаем кнопку OK, затем еще раз OK — новый класс создан. Теперь нам надо внести некоторые изменения.

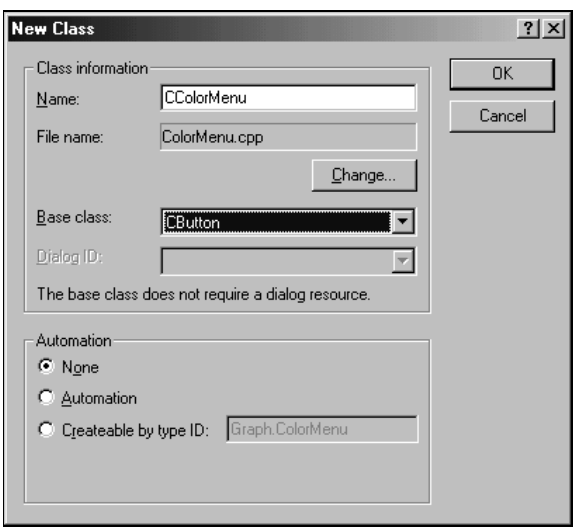



**Рис. 12.12.** Воспользоваться ClassWizard можно даже в том случае, если в списке базовых классов нет того, который нужен

4. В окне Workspace на вкладке ClassView дважды щелкните на имени CColorMenu, что приведет к открытию файла, где этот класс определен. Замените имя CButton на CMenu — сделать это надо в одном единственном месте.
5. Раскройте вкладку FileView в окне Workspace и дважды щелкните по имени файла <colormenu.cpp>. В строке

```
BEGIN_MESSAGE_MAP(CColorMenu, CButton)
```

замените *CButton* на *CMenu*. До этой строки можно "добраться" и другим способом, я вам показываю только один из возможных вариантов.

6. Сохраните сделанные изменения. В результате мы получили собственный класс *CColorMenu*, производный от *CMenu*. Кстати, в данном конкретном случае можете удалить из определения и реализации класса (обязательно в обоих местах) все компоненты, кроме конструктора, — они нам не понадобятся.

После того как класс, базирующийся на *CMenu*, создан, нам необходимо переопределить две функции, которые отвечают за обработку сообщений WM_MEASUREITEM и WM_DRAWITEM. Воспользуйтесь для этого мастером ClassWizard. В результате вы должны получить следующий код:

```
#if !defined(
  AFX_COLORMENU_H__0C5A82D5_B46D_11D3_BAB9_00600864785A__INCLUDED_)
#define AFX_COLORMENU_H__0C5A82D5_B46D_11D3_BAB9_00600864785A__INCLUDED_
#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000
// ColorMenu.h : header file
//
/////////////////////////////////////////////////////////////////////////
CColorMenu window
class CColorMenu : public CMenu
{
// Construction
public:
  CColorMenu();
// Implementation
public:
  virtual void MeasureItem(LPMEASUREITEMSTRUCT lpMeasureItemStruct);
  virtual void DrawItem(LPDRAWITEMSTRUCT lpDrawItemStruct);
};
#endif // !defined(
  AFX_COLORMENU_H__0C5A82D5_B46D_11D3_BAB9_00600864785A__INCLUDED_)
/////////////////////////////////////////////////////////////////////////
// ColorMenu.cpp : implementation file
//
#include "stdafx.h"
#include "Graph.h"
#include "ColorMenu.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
```

```
static char THIS_FILE[] = __FILE__;
#endif
////////////////////////////////////////////////////////////////////
CColorMenu
CColorMenu::CColorMenu()
{
}
////////////////////////////////////////////////////////////////////
CColorMenu message handlers
void CColorMenu::MeasureItem(
                    LPMEASUREITEMSTRUCT lpMeasureItemStruct)
{
}
void CColorMenu::DrawItem(LPDRAWITEMSTRUCT lpDrawItemStruct)
{
}
////////////////////////////////////////////////////////////////////
CColorMenu message handlers
```

Нам осталось только наполнить содержанием созданные для нас оболочки функций *DrawItem* и *MeasureItem*. Однако сначала рассмотрим использованные в них структуры *DRAWITEMSTRUCT* и *MEASUREITEMSTRUCT*.

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. Может принимать следующие значения:

| | |
|---|---|
| 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 отри-

цательное 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, когда 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*.

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 равно 255). *itemData* — назначение этого поля аналогично одноименному полю структуры *DRAWITEMSTRUCT*.

Теперь задаем размеры элементов меню. Для этой задачи предназначена функция *MeasureItem*:

```
void CColorMenu::MeasureItem(
                    LPMEASUREITEMSTRUCT lpMeasureItemStruct)
{
  lpMeasureItemStruct->itemWidth  = 130;
  lpMeasureItemStruct->itemHeight = 20;
}
```

Как вы понимаете, размеры вы можете задавать произвольные. Главное — сделать это именно здесь.

Нам осталось только нарисовать в элементах меню то, ради чего, собственно, все это и затевалось. Используем функцию:

```
void CColorMenu::DrawItem(LPDRAWITEMSTRUCT lpDrawItemStruct)
{
  // Определяем прямоугольник,
  // который будем заполнять соответствующим цветом
  CRect rect(lpDrawItemStruct->rcItem.left,
             lpDrawItemStruct->rcItem.top,
             lpDrawItemStruct->rcItem.right - 110,
             lpDrawItemStruct->rcItem.bottom);
  // При обработке сообщения WM_DRAWITEM для нас уже создан
  // контекст устройства, и нам остается им воспользоваться
  CDC *pDC = CDC::FromHandle(lpDrawItemStruct->hDC);
  // Дальнейшие действия выполняем только в случае,
  // если перерисовки требует весь элемент меню
  if(lpDrawItemStruct->itemAction & ODA_DRAWENTIRE)
  {
    // По идентификатору элемента меню находим его цвет
    COLORREF cr = colors[lpDrawItemStruct->itemID - ID_CLR0];
    // Заполняем прямоугольник выбранным цветом
    pDC->FillSolidRect(&rect, cr);
    // и отделяем его от последующего текста
    pDC->MoveTo(rect.right + 1, 0);
    pDC->LineTo(rect.right + 1, rect.bottom);
    // Выводимый текст не должен создавать "пятна"
    pDC->SetBkMode(TRANSPARENT);
    // Выводим текстовое обозначение цвета
    pDC->TextOut(rect.right + 10, rect.top + 4,
                 clrString[lpDrawItemStruct->itemID - ID_CLR0]);
  }
}
```

Помимо этого не забудьте определить необходимые идентификаторы и тексты:

```
// MainFrm.cpp : implementation of the CMainFrame class
//
```

```
// Определяем массив цветов
COLORREF colors[] =
{
  RGB(  0,   0,   0),
  RGB(255, 255, 255),
  RGB(128, 128, 128),
  RGB(192, 192, 192),
  RGB(128,   0,   0),
  RGB(255,   0,   0),
  RGB(128, 128,   0),
  RGB(255, 255,   0),
  RGB(  0, 128,   0),
  RGB(  0, 255,   0),
  RGB(  0, 128, 128),
  RGB(  0, 255, 255),
  RGB(  0,   0, 128),
  RGB(  0,   0, 255),
  RGB(128,   0, 128),
  RGB(255,   0, 255)
};
// ColorMenu.cpp : implementation file
//
// Мы должны иметь доступ к массиву цветов
extern COLORREF colors[];
// Определяем массив названий цветов
CString clrString[] =
{
  "Черный",
  "Белый",
  "Темно-серый",
  "Серый",
  "Темно-красный",
  "Красный",
  "Темно-желтый",
  "Желтый",
  "Темно-зеленый",
  "Зеленый",
  "Темно-голубой",
  "Голубой",
  "Темно-синий",
  "Синий",
  "Темно-малиновый",
  "Малиновый"
};
```

```
// Resource.h
//
...
// Для каждого цвета необходим свой идентификатор
#define ID_CLR0                         32788
#define ID_CLR1                         32789
#define ID_CLR2                         32790
#define ID_CLR3                         32791
#define ID_CLR4                         32792
#define ID_CLR5                         32793
#define ID_CLR6                         32794
#define ID_CLR7                         32795
#define ID_CLR8                         32796
#define ID_CLR9                         32797
#define ID_CLR10                        32798
#define ID_CLR11                        32799
#define ID_CLR12                        32800
#define ID_CLR13                        32801
#define ID_CLR14                        32802
#define ID_CLR15                        32803
...
```

Думаю, что нет никакой необходимости давать еще какие-либо пояснения в дополнение к имеющимся комментариям. То, что получилось в результате наших действий, представлено на рис. 12.13.

## Создание собственных маркеров состояния

Последний пример, который мы рассмотрим в связи с меню, заключается в реализации собственных маркеров, выводимых вместо "галочки".

Первое, что нам надо — это создать рисунки, которые будут показывать отмеченное и неотмеченное состояния. Например, такие, как представленные на рис. 12.14. Я уже не буду объяснять, как это сделать.

Следующий необходимый шаг — написать код, позволяющий использовать созданные маркеры. Как обычно, сначала определим необходимые переменные, что мы и делаем в описании класса *CMainFrame*:

```
class CMainFrame : public CFrameWnd
{
...
// Attributes
public:
  CBitmap m_bmpCheck;   // для отмеченного состояния
```

```
  CBitmap m_bmpUnCheck; // для неотмеченного состояния
...
}
```

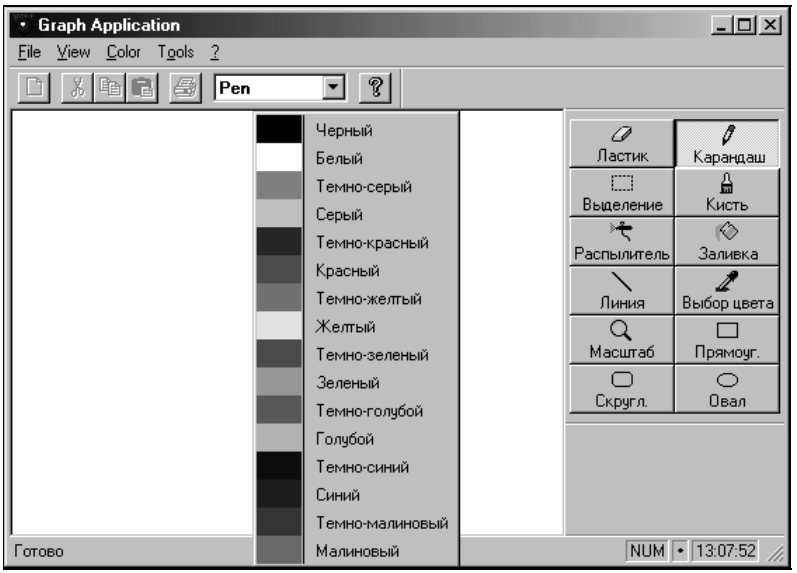



**Рис. 12.13.** Результат наших действий по рисованию элементов меню

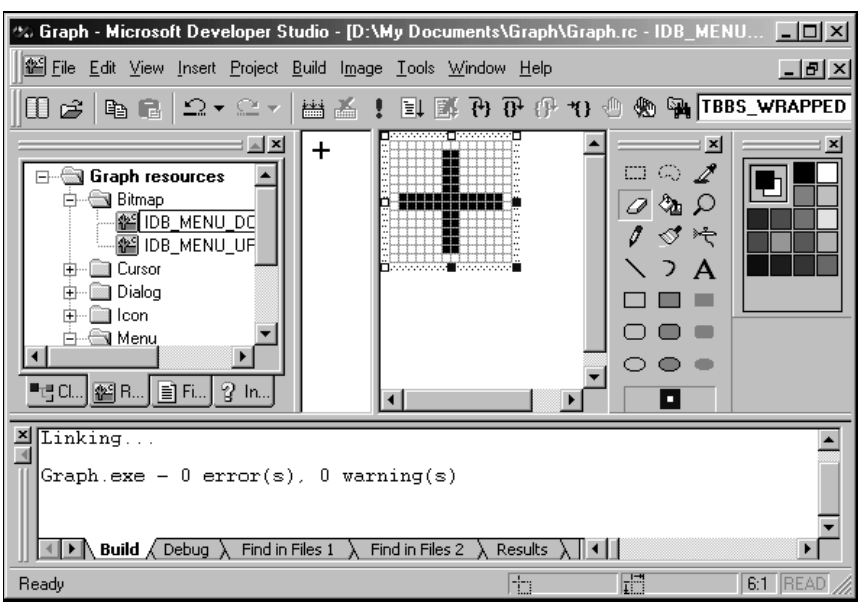



**Рис. 12.14.** Так будет выглядеть "галочка" у отмеченного элемента меню

Затем добавляем код в обработчик события WM_CREATE:

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
...
  // Загружаем битовые массивы с новыми маркерами
  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);
  // Получаем доступ к главному меню
  CMenu *pViewMenu = NULL;
  CMenu *pTopMenu = AfxGetMainWnd()->GetMenu();
  int iPos;
  // Ищем элемент меню "2 column" по идентификатору ID_TOOLS_2COLUMN
  for (iPos = pTopMenu->GetMenuItemCount()-1; iPos >= 0; iPos--)
  {
    // Поиск выполняем по позиции элемента
    CMenu* pMenu = pTopMenu->GetSubMenu(iPos);
    if (pMenu && pMenu->GetMenuItemID(0) == ID_TOOLS_2COLUMN)
    {
      // Если соответствующий идентификатор найден,
      // прекращаем поиск, запомнив указатель на этот элемент
      pViewMenu = pMenu;
      break;
    }
  }
  // Мы обязательно должны найти элемент меню
  ASSERT(pViewMenu != NULL);
  // Устанавливаем новые маркеры для отмеченного и
  // неотмеченного состояний,
  // которые теперь будут появляться вместо "галочек"
  pViewMenu->SetMenuItemBitmaps(ID_TOOLS_2COLUMN, MF_BYCOMMAND,
                                &m_bmpUnCheck, &m_bmpCheck);
  pViewMenu->SetMenuItemBitmaps(ID_TOOLS_3COLUMN, MF_BYCOMMAND,
                                &m_bmpUnCheck, &m_bmpCheck);
...
}
```

И вот что у нас получилось в результате (рис. 12.15).

Конечно, рисунки маркеров оставляют желать лучшего, но вы можете сами нарисовать что-то более красивое. А как "подключить" их к меню, я вам рассказал.

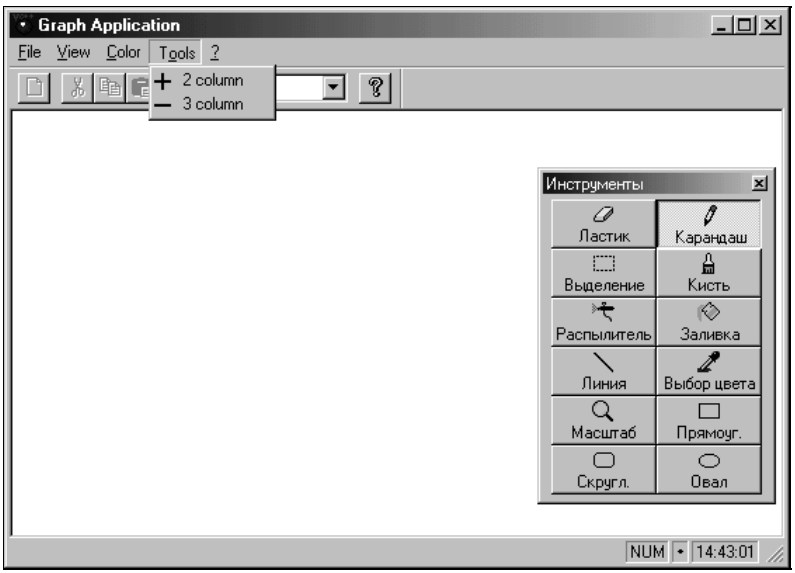



**Рис. 12.15.** Так выглядят новые маркеры отмеченного и неотмеченного состояний

# ГЛАВА 13



# Создание многодокументных приложений

Слишком много таланта иметь нельзя,
но слишком много талантов — можно.
*Мария Эбкар-Эшенбах*

Ну вот мы и подошли к следующей важной теме. Я надеюсь, что нет необходимости подробно представлять *многодокументный интерфейс* (MDI, Multiple Document Interface), который позволяет в одном приложении работать с несколькими окнами. MDI описан в руководстве по разработке интерфейса пользователя System Application Architecture /Common User Access/ Advanced Interface Design Guide (SAA/CUA), созданном фирмой IBM. В системе Windows интерфейс MDI используется, начиная с версии 3.0, и его типичным представителем является хорошо всем знакомый текстовый процессор Microsoft Word.

Как и все приложения Windows, MDI-приложение имеет одно главное окно, которое управляет своими окнами "MDI child".

**Примечание**

Для названия окна документа я буду придерживаться терминологии фирмы Microsoft, которая для дочерних окон MDI-приложений использует названия "MDI child frame", "MDI child window" или просто "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. 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 пиктограмма 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 располагаются кнопки изменения размера и закрытия окна.

Одним словом, можно создать аналогичное приложение с использованием обычных окон, но для этого придется затратить немало усилий. Достаточно серьезной работой будет и создание полноценного MDI-приложения с помощью функций библиотеки SDK (Software Development Kit). Чтобы в этом убедиться, достаточно посмотреть исходные тексты приложения MULTIPAD, поставляемые в составе примеров вместе 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 рассмотрим 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* — это базовый класс для главного окна любого многодокументного приложения (рис. 13.1). Если быть более точными, то в данном случае следует говорить не об окне, а о фрейме, поскольку помимо него создается еще и специальное окно, называемое окном MDICLIENT, которое управляет рабочей областью фрейма и является, собственно, родительским для окон "MDI child".

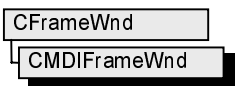



**Рис. 13.1.** Место класса *CMDIFrameWnd* в иерархии библиотеки MFC

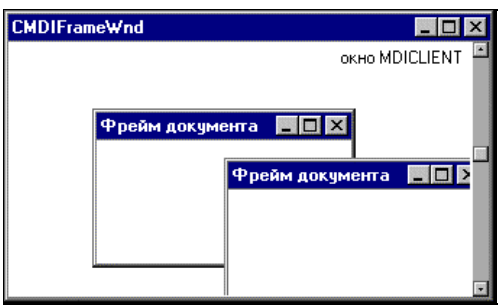



**Рис. 13.2.** Типы окон интерфейса MDI

Уточним некоторые моменты, связанные с процессом создания MDI-окна. Как обычно, сначала создается объект нашего оконного класса. 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, в частности *OnCreateClient*. Однако и это скрыто в функциях библиотеки, если при создании SDI-окна вызывается функция класса *CFrameWnd*, то при создании MDI-окна вызывается одноименная функция класса *CMDIFrameWnd*. В первом случае она отвечает за создание (если необходимо) присоединяемого к окну дочернего окна, а во втором — за создание специального окна MDICLIENT, которое, как мы уже говорили, полностью управляет рабочей областью главного окна MDI-приложения и отвечает за работу с окнами "MDI child". Все перечисленные окна показаны на рис. 13.2.

**Примечание**

Разница, безусловно, есть, но явно она заметна только в случае построения приложения на основе архитектуры "документ/представление". В других же случаях она скрыта в недрах библиотеки.

Прежде чем переходить к примерам, давайте рассмотрим функциональные возможности, которые класс *CMDIFrameWnd* предоставляет в наше распоряжение.

`void CMDIFrameWnd::`**`MDIActivate`** `(CWnd *pWndActivate)`

Параметр *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. Такое же сообщение посылается, если пользователь изменил фокус дочернего окна при помощи мыши или клавиатуры. 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 также 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.

`void CMDIFrameWnd::`**`MDITile`** `()`

и

`void CMDIFrameWnd::`**`MDITile`**`(int nType)`

Располагают все окна "MDI child" без перекрытия. Первая версия функции располагает окна по вертикали, а вторая в качестве параметра *nType* может принимать 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 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 (Окно) основного меню типа ID_WINDOW_NEW и ID_WINDOW_TILE_HORZ. Если раскрывающееся меню не содержит идентификаторов стандартных команд, то эту функцию следует переопределить.

```
CMenu* CMDIFrameWnd::MDISetMenu (
    CMenu *pFrameMenu,
    CMenu *pWindowMenu)
```

Заменяет соответствующие текущие меню на переданные в качестве параметров, если *pFrameMenu* и/или *pWindowMenu* не равны NULL. 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 для управления окнами "MDI child" вы используете фрейм, то эту функцию вызывать не следует.

Как видите, всю основную работу по поддержанию работы MDI-приложений выполняет система Windows совместно с библиотекой MFC.

Прежде чем двигаться дальше, давайте также 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 несколько упрощенно, позаимствовали свои свойства как у дочерних, так и у перекрывающихся окон (рис. 13.3).

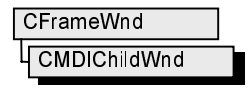



**Рис. 13.3.** Место класса *CMDIChildWnd* в иерархии библиотеки MFC

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".

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. Активное текущее окно "MDI child" может определять заголовок своего родительского окна, если для него установить бит стиля FWS_ADDTOTITLE. Описываемая 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 сброшен 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, управляющее объектом класса *CMDIChildWnd*, необходимо вызвать функцию *CWnd::GetParent*.

Теперь, когда мы познакомились с обоими классами *CMDIFrameWnd* и *CMDIChildWnd*, позволяющими создавать приложения на базе интерфейса MDI, пришло время рассмотреть конкретный пример, который поможет закрепить полученные сведения и познакомиться с дополнительными возможностями библиотеки MFC.

# Пример MDI-приложения

В этом разделе мы рассмотрим пример (рис. 13.4), на котором я продемонстрирую основные принципы построения MDI-приложений. В дальнейшем он может послужить основой для разработки многодокументных приложений любой степени сложности.

Но прежде всего, как и в любом другом случае, необходимо создать проект. Для наших целей воспользуемся мастером MyWizard4, который реализует для нас каркас приложения. Начнем с класса-приложения:

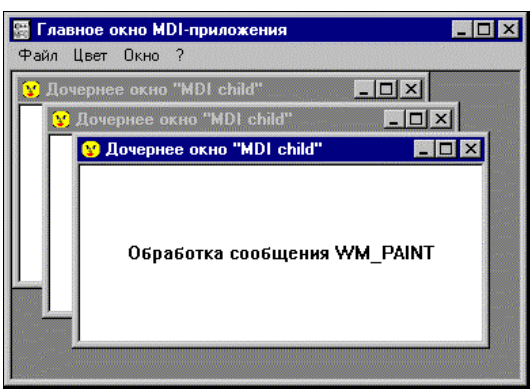



**Рис. 13.4.** Главное и дочерние окна MDI-приложения

```
BOOL CMDIApp::InitInstance()
{
  // Заголовок будем брать из ресурсов
  CString csCaption;
  csCaption.LoadString(IDR_MAINFRAME);
  // Ищем копию нашего приложения
  CWnd *pWnd = CWnd::FindWindow(NULL, csCaption);
  // Если таковая есть, то ...
  if(pWnd != NULL){
    // раскрываем его, если оно было свернуто
    if(pWnd->IsIconic())
      pWnd->ShowWindow(SW_RESTORE);
  // или выводим на передний план
  pWnd->SetForegroundWindow();
  return FALSE;
 }
 // Создаем объект главного MDI-фрейма
 CMainFrame* pMainFrame = new CMainFrame;
 // и сам фрейм
 if (!pMainFrame->LoadFrame(IDR_MAINFRAME))
  return FALSE;
  m_pMainWnd = pMainFrame;
  pMainFrame->ShowWindow(m_nCmdShow);
  pMainFrame->UpdateWindow();
  return TRUE;
}
```

Первое, что сразу бросается в глаза при рассмотрении приведенного фрагмента — это практически полная идентичность интерфейса и реализации нашего класса "приложение" (*CMDIApp*) и класса *CFirstApp*, описанного выше (см. главу 3). А если не считать появления специального обработчика, 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 компонуете составные части так, как этого требует конкретная задача.

## Поиск запущенного экземпляра приложения

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 замечанием, что в объекте этого класса будет храниться символьная строка с именем окна, загруженная из файла ресурсов.

Как создать такой ресурс, вы к этому моменту должны уже хорошо представлять, поэтому ограничусь только его "внешним видом" (рис. 13.5).

Но вернемся к фрагменту кода. Мы вставили его 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. В отличие от ранних версий Windows, для проверки того, не было ли приложение уже загружено, мы не можем воспользоваться параметром *hInstance*, передаваемом в командной строке, поскольку 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.

**Примечание**

Функцию *FindWindow* нельзя использовать для поиска дочерних окон.

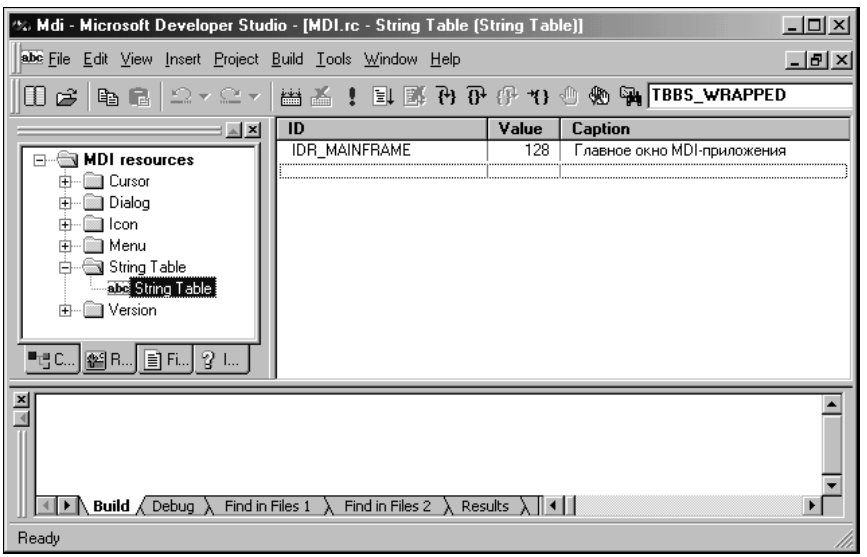



**Рис. 13.5.** Для поиска экземпляра приложения нам понадобится строковый ресурс

```
static CWnd* CWnd::FindWindow (
    LPCTSTR lpszClassName,
    LPCTSTR lpszWindowName)
```

Через параметр *lpszClassName* передается указатель на символьную строку, определяющую имя класса искомого окна. Если этот параметр равен NULL, как в нашем случае, то поиск будет осуществляться во всех классах окон. 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 библиотеке MFC, то параметром *lpszClassName* пользоваться просто невозможно, т. к. имя класса генерируется каждый раз новое и имеет примерно такой вид: "Afx:400000:8:1526:0:13F7" (при следующем запуске, даже второй копии приложения, 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 полноты картины приведем еще одну функцию (не использованную в нашем приложении), 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 при этом не разрешает изменять стили окна.

Возвращаясь к нашему примеру, отметим, что если работающая копия приложения обнаружена, то нам остается только завершить работу текущего экземпляра. Если же работающая копия не найдена, то переопределенная функция *InitInstance* приступает к инициализации приложения. Этот процесс уже был подробно описан ранее, и мы не будем к нему возвращаться.

Займемся теперь используемыми классами окон. Прежде всего обратим внимание на базовый класс нашего главного окна:

```
class CMainFrame : public CMDIFrameWnd
{
  ...
};
```

Как видите, он образован из класса, специально спроектированного для окон MDI. При этом действия по созданию главного окна MDI-приложения осуществляются 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. Строгих правил, определяющих место, где следует регистрировать оконный класс, нет. 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));
  return 0;
}
```

Обратите внимание на то, что нам не надо явно создавать специальное окно MDICLIENT, т. к. оно автоматически формируется функциями библиотеки. Вы, конечно, можете вмешаться в этот процесс, переопределив функцию класса *CMDIFrameWnd — CreateClient*. Однако еще раз подчеркну — 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, связанные с меню многодокументного интерфейса.

## Изменение полосы меню

Здесь для демонстрации всех команд меню мы рассмотрим не рисунок с внешним видом меню, а фрагменты из файла ресурсов, которые получаются после того, как мы создали меню рассмотренным выше способом.

**Примечание**

Использование графического редактора ресурсов для создания меню значительно проще, чем внесение изменений непосредственно в файл ресурсов. Однако это не означает, что такой подход является обязательным.

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
```

**Примечание**

Команду меню "О программе ..." я оставил, поскольку она у нас есть. Но вопросы, которые с ней связаны, мы рассмотрим в следующей главе, когда будем знакомиться с блоками диалога.

Прежде всего мы 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 — целое окно (например, раскрывающийся перечень компонентов, который появляется при открытии меню File (Файл) во многих приложениях) и MENUITEM — единичный элемент меню, после объявления которого в кавычках записана символьная строка, изображаемая в строке меню. Объявляется также константа, которая ассоциирована с данным пунктом меню. (Обратите внимание, что *каждый* пункт меню ассоциируется с некоторой константой.) Как вы знаете из предыдущей главы, когда поль-

зователь выбирает конкретный пункт меню, соответствующая константа и пункт меню упаковываются в параметры сообщения WM_COMMAND и посылаются в соответствующую оконную процедуру. 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 создаем обработчик сообщения *OnText* и связываем 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;
}
```

Все перечисленные до этого момента действия вам уже хорошо знакомы. Поэтому переходим к более интересному для нас вопросу — подключению специального меню для дочернего окна.

Прежде всего необходимо подготовить само меню. Вот как оно выглядит у нас:

```
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 пиктограммы",     ID_WINDOW_ARRANGE
    END
    POPUP "?"
    BEGIN
        MENUITEM "&О программе ...",            ID_APP_ABOUT
    END
END
```

Для того чтобы подключить меню окна "MDI child" к основному окну MDI-приложения, мы определили компонент *m_menu* класса *CTextWnd*:

```
class CTextWnd : public CMDIChildWnd
{
...
protected:
  static CMenu m_menu;  // Меню для всех окон "MDI child"
                        // Если необходимо для каждого окна
                        // использовать свое меню, ключевое слово
                        // static надо убрать
...
};
```

Теперь остается только присвоить идентификатор меню окна "MDI child" специальному компоненту класса:

```
BOOL CTextWnd::PreCreateWindow(CREATESTRUCT& cs)
{
...
  // Проверяем, не было ли меню уже загружено
  if (m_menu.m_hMenu == NULL)
    m_menu.LoadMenu(IDR_MDITEXT);
```

```
  // Сообщаем, что теперь должно выводиться именно это меню
  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)
...
  // Макросов много, а обработчик будет только один
  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 именно на него задачу идентификации этих команд.

**Примечание**

Другой способ использования одного обработчика для нескольких команд, базирующийся на макросе ON_COMMAND_RANGE, был рассмотрен ранее в главе 5. Предложенный способ удобен тем, что позволяет переложить вспомогательную работу на ClassWizard.

Вот как выглядит его код:

```
// Для значений цветов создаем специальный массив
COLORREF colorArray[] =
{
  RGB (0, 0, 0),       // черный
  RGB (255, 255, 255), // белый
  RGB (255, 0, 0),     // красный
```

```
  RGB (0, 0, 255),      // синий
  RGB (0, 255, 0)       // и зеленый цвета
};
...
void CTextWnd::OnColor()
{
  // Получаем идентификатор команды
  nIDColor = LOWORD(GetCurrentMessage()->wParam);
  // По идентификатору находим значение цвета
  clrText = colorArray[nIDColor - IDM_BLACK];
  // Даем команду на перерисовку окна
  Invalidate();
}
```

**Примечание**

Здесь необходимо соблюдение обязательного условия: идентификаторы команд цветов должны строго следовать друг за другом. Иначе вы получите непредсказуемые результаты. Сбой в порядке следования идентификаторов происходит в том случае, если произошло переключение с формирования меню на создание какого-либо другого ресурса. В этом случае необходимо открыть файл <resource.h> и вручную присвоить последовательные числовые значения требуемым идентификаторам.

Использованная в этом коде функция *CWnd::GetCurrentMessage* возвращает указатель на структуру *MSG*, в слове *wParam* которой содержится идентификатор выбранного пункта меню (идентификатор команды). Вот фрагмент из файла <resource.h>:

```
...
#define IDM_BLACK                       101
#define IDM_WHITE                       102
#define IDM_RED                         103
#define IDM_BLUE                        104
#define IDM_GREEN                       105
...
```

Как видите, используемые в данном фрагменте идентификаторы команд идут один за другим. Это не случайно и позволяет нам использовать их для определения индекса в массиве цветов как (*nIDColor — IDM_BLACK*). Попробуйте изменить числовое значение у какого-либо идентификатора — программа сразу же перестанет адекватно реагировать на эту команду.

Нам осталось рассмотреть принцип действия специальной команды обновления и связанный с ней класс *CCmdUI*. Поставим задачу отметить "галочкой" тот элемент меню, который является выбранным на текущий момент времени. Реализовать это очень просто — воспользуемся функцией *CCmdUI::SetCheck* в обработчике *OnUpdateColor*. Поскольку этот обработчик

каждый раз вызывается для всех элементов раскрывающегося меню, то перед перерисовкой необходимо обеспечить, чтобы "галочка" была установлена только у того элемента, который был перед этим выбран (`pCmdUI->m_nID == nIDColor`), а у остальных — убрана:

```
void CTextWnd::OnUpdateColor(CCmdUI* pCmdUI)
{
  // Поскольку текущим может быть только один цвет,
  // необходимо его отслеживать
  pCmdUI->SetCheck(pCmdUI->m_nID == nIDColor);
}
```

Вот, собственно, и все, что я хотел рассказать о работе с меню в приложениях многооконного интерфейса.

Непосредственный процесс создания окна вам уже хорошо знаком по однодокументным приложениям. Поэтому обратим внимание только на следующую строку из функции *OnText*:

```
_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.

## Стандарт 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 стоит 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, оказывается недостаточно для представления всех букв 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 65000 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), японского (кана) 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 упрощения задачи по 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 как

```
typedef wchar_t TCHAR;
```

а если не определен, то как

```
typedef char TCHAR;
```

При использовании этого типа данных строка символов объявляется как

```
TCHAR szString[128];
```

Кроме того, для корректной работы со строками необходимо использовать макрос _TEXT (или _T), определенный как

```
#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;
- во избежание каких-либо проблем в строковой арифметике используйте выражение *(sizeof(szBuffer) / (sizeof(TCHAR))* вместо *sizeof(szBuffer))*.

После этого небольшого отступления можно возвращаться к нашему примеру.

Последней в обработчике *OnColor* вызывается функция *CWnd::Invalidate*. Мне представляется важным познакомиться с ней поближе.

```
void CWnd::Invalidate (BOOL bErase = TRUE)
```

Функция 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 не пуста, 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 только какую-либо ее часть, в классе *CWnd* реализованы специальные функции *InvalidateRect* и *InvalidateRgn*. Первая из них "помечает" прямоугольную область, а вторая — произвольную.

## Еще раз о сообщении WM_PAINT

Сообщение, которое мы здесь рассматриваем, является одним из наиболее важных, если не самым важным, 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 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()
{
  // Без определения контекста устройства
  // мы ничего не сможем нарисовать
  CPaintDC dc(this);
  // Выбираем в контекст устройства текущий цвет
  dc.SetTextColor(clrText);
  // Цвет фона запрашиваем у системы
  dc.SetBkColor(::GetSysColor(COLOR_WINDOW));
  // Определяем прямоугольник, в котором будем рисовать
  CRect rect;
  GetClientRect(rect);
  // В данном случае "рисование" будет заключаться
  // в выводе строки текста
```

```
  // Отображаем текст в центре окна
  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 была посвящена отдельная большая глава. 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, вызывает автоматически 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, чтобы избежать истощения ресурсов. Для этого в деструкторе автоматически вызывается функция *CWnd::EndPaint*. Между этими двумя процедурами располагается ядро обработчика сообщения WM_PAINT.

Наш обработчик демонстрирует лишь один из основных способов вывода текста в Windows-программах. Он основан на применении функции *DrawText*, которую мы уже рассматривали. Для форматирования мы выбрали центрирование по вертикали и горизонтали, а также вывод текста одной строкой. Функция *DrawTex* 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*.

В нашем обработчике осталась еще одна функция. И поскольку она не имеет отношения к контексту устройств, опишу ее подробно:

`void CWnd::`**`GetClientRect`** `(LPRECT lpRect)`

> Функция 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;
```

Определяет координаты левого верхнего (*left* — *x*-координата, *top* — *y*-координата) и правого нижнего (*right* — *x*-координата, *bottom* — *y*-координата) углов некоторого прямоугольника.

Поскольку Windows большей частью работает с прямоугольными областями, в библиотеке MFC определен специальный класс *CRect*, который опирается на только что рассмотренную структуру *RECT* и, кроме того, включает в себя специальные функции для манипулирования объектами 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*.

И последнее, что хотелось бы сказать по поводу сообщения WM_PAINT. Проделайте следующий эксперимент. Добавьте в программу обработчик нажатия левой кнопки мыши (как это сделать, вы уже отлично знаете):

```
void CTextWnd::OnLButtonDown(UINT nFlags, CPoint point)
{
  // И здесь нам нужен контекст устройства
  CClientDC dc(this);
  TCHAR strText[] = "Обработчик сообщения WM_LBUTTONDOWN";
  // Задаем цвета текста и фона
  dc.SetTextColor(clrText);
  dc.SetBkColor(::GetSysColor(COLOR_WINDOW));
  // Выводим собственно текст
  dc.TextOut(10, 10, strText, strlen(strText));
  // Не забываем передать управление библиотеке,
  // и в этом нам хорошо помогает ClassWizard
  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*.

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 по его заголовку, то увидите примерно то, что изображено на рис. 13.6.

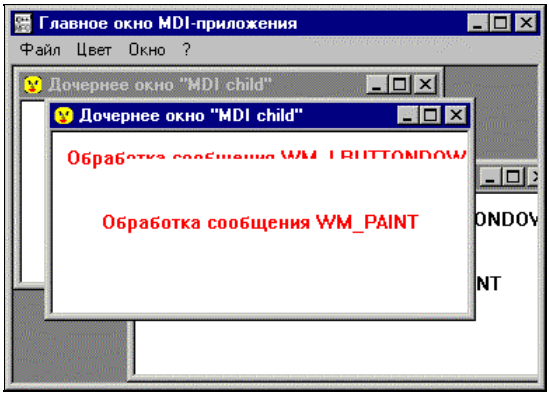



**Рис. 13.6.** Пример вывода текста в рабочую область окна с использованием и без использования обработчика сообщения WM_PAINT

Если вы отчетливо понимаете, что произошло, то вы действительно вникли в суть механизмов, реализованных 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.

В заключение осталось сказать 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 не забывайте вызывать соответствующий обработчик базового класса.



# Часть V

## Интерактивное взаимодействие с приложением

**Глава 14.** Модальные и немодальные блоки диалога

**Глава 15.** Наборы свойств

**Глава 16.** Эти разнообразные элементы управления

# ГЛАВА 14



# Модальные и немодальные блоки диалога

Импровизировать может лишь тот,
кто знает роль назубок.
*Джоди Фостер*

*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*.

Блоки диалога бывают двух типов — *модальные* (modal) и *немодальные* (modeless). Разница между ними заключается в способе управления потоками сообщений. Модальный блок диалога блокирует все остальные окна приложения так, что пользователь не может с ними ничего сделать, пока не закроет блок. Другими словами, модальный блок отсекает 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 для них уже давно сложился стандартный и широко используемый набор элементов управления.

**Примечание**

Наряду с уже привычными элементами управления Windows: кнопками (PUSHBUTTON), флажками (CHECK BOX), переключателями (RADIO BUTTON), списками (LIST BOX), элементами редактирования (EDIT), комбинированными списками (COMBO BOX), полосами прокрутки (SCROLLBAR) и статическими элементами (STATIC) в Win32 добавлено достаточно большое число новых, которые будут подробно рассмотрены ниже.

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 Graph"
FONT 8, "MS Sans Serif"
BEGIN
  ICON  IDR_MAINFRAME,IDC_STATIC,7,7,21,20
  CTEXT "Приложение Graph Версия 1.0",
    IDC_STATIC,51,17,119,8, SS_NOPREFIX
  CTEXT  "Copyright © 2000",IDC_STATIC,127,34,71,8
  DEFPUSHBUTTON   "OK",IDOK,178,7,32,14,WS_GROUP
  LTEXT  "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).

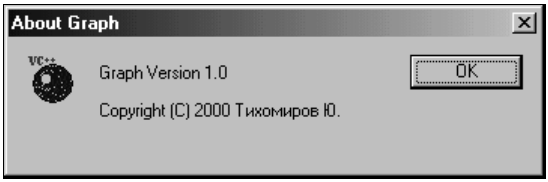



**Рис. 14.1.** Эскиз блока диалога

Но какой бы способ вы не выбрали, помните, что блок диалога — это всего-навсего еще один ресурс. Внимательно посмотрев на приведенный текст, вы заметите, что первая строчка содержит идентификатор ресурса блока диалога (IDD_ABOUTBOX), за которым следует объявление типа ресурса (DIALOG) и размеров блока диалога. Следующая строчка 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 определять 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 не будет и полосы заголовка.

Описание блока диалога заканчивается несколькими строчками, заключенными между BEGIN и END. Они определяют пиктограмму, 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 пиктограмму.

Строчка, описывающая кнопку, начинается с оператора, определяющего тип элемента управления (DEFPUSHBUTTON) и имеет 7 параметров. Первый — текст, ассоциирующийся с элементом управления, затем идут идентификатор, прямоугольник размещения и стиль. Сектор размещения задается четырьмя числами: координатами левого верхнего угла, шириной и высотой.

Поскольку и пиктограмма (ICON), и статические элементы (LTEXT и CTEXT) являются окнами, то неудивительно, что они имеют такие 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, являющийся общим для различных элементов управления и имеющий 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 (пиктограммами, 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 подробнее.

# Класс *CDialog*

Основное назначение класса *CDialog* состоит в предоставлении интерфейса программирования для управления блоками диалога.

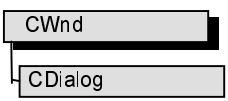



**Рис. 14.2.** Место класса *CDialog* в иерархии библиотеки MFC

Использование данного класса 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 — создание 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)
```

Первый параметр определяет шаблон: *lpszTemplateName* — указатель на символьную строку, содержащую имя шаблона, *nIDTemplate* — номер идентификатора ресурса. Параметр *pParentWnd* указывает на объект родителя или вла-

дельца, которому принадлежит объект блока диалога. Если он равен NULL, то родительским окном является основное окно приложения.

Следующим шагом является создание 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 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 |
| 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 |
| DS_CENTERMOUSE | Курсор мыши располагается в центре блока диалога |
| 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 командные клавиши и т. д. |
| DS_FIXEDSYS | Блок диалога использует SYSTEM_FIXED_FONT вместо SYSTEM_FONT |
| 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 он обязательно должен иметь установленным 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 применяется при создании с помощью шаблона блока диалога других типов окон.)

В поле *cdit* задается число элементов блока диалога или, другими словами, — число структур *DLGITEMTEMPLATE*, которые описывают элементы управления блока диалога. Эта структура содержит следующие поля:

```
typedef struct {
  DWORD style;
  DWORD dwExtendedStyle;
  short x;
  short y;
  short cx;
  short cy;
  WORD id;
} DLGITEMTEMPLATE;
```

Все поля структуры *DLGITEMTEMPLATE* 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 атома, 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 этой функции инициализируют объект модального блока диалога, используя предварительно созданный в памяти шаблон, на который указывает либо *lpDialogTemplate* — указатель на структуру *DLGTEMPLATE*, либо *hDialogTemplate* — дескриптор глобальной памяти, где определены структура *DLGTEMPLATE* и данные для каждого элемента управления.

**Примечание**

Обратите особое внимание на то, что после выполнения функции *CDialog::InitModalIndirect* система Windows создает блок диалога, но не отображает его. Он появляется на экране только после непосредственного 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)
```

Функцию можно вызвать прямо внутри конструктора класса, производного от *CDialog*, а ее параметры аналогичны параметрам конструктора для модального блока диалога.

Второй способ создания немодального блока диалога опирается на шаблон, предварительно созданный в памяти. Этот способ по своей сути ничем не отличается от аналогичного способа 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. Если стиль не установлен, то 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 окончания работы с немодальным блоком диалога необходимо вызвать функцию *CWnd::DestroyWindow* для того, 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 него 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 система 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 фокус ввода явно устанавливается на какой-либо элемент управления, и ненулевое, когда фокус ввода устанавливается на первый элемент.

После окончания работы с модальным блоком диалога обязательно должна вызываться функция

`void CDialog::`**`EndDialog`** `(int nResult)`

Функция не закрывает блок диалога, 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 обработчика кнопки OK.

`virtual void CDialog::`**`OnOK`** `()`

Фиксирует данные блока диалога и обновляет соответствующие переменные приложения. Если нужны какие-либо дополнительные действия, то можно переопределить эту функцию для модального блока диалога. В случае немодального блока диалога ее переопределение обязательно. 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 (Отмена) в классе реализован свой обработчик

`virtual void CDialog::`**`OnCancel`** `()`

Завершает работу с модальным блоком диалога, вызывая функцию *EndDialog*, и возвращает IDCANCEL. Для немодального блока диалога эту функцию необходимо переопределить и вызвать внутри нее функцию *DestroyWindow*.

Последней переопределяемой функцией, реализованной в классе, является

`virtual void CDialog::`**`OnSetFont`** `(CFont *pFont)`

Вызывается в ответ на сообщение WM_SETFONT и устанавливает шрифт, определенный в редакторе блока диалога. Если шрифт нужно изменить, то функцию следует переопределить. Установленный шрифт используется по умолчанию всеми элементами управления блока диалога.

При обсуждении способов создания блока диалога я упомянул о том, что они имеют свои координаты, которые называются *координатами блока 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)
```

Получает на входе 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 ()
```

Функции 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*).

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 функцией

```
CWnd* CWnd::GetDlgItem (int nID)
```

или

```
void CWnd::GetDlgItem (
    int nID,
    HWND *phWnd),
```

которые по идентификатору элемента управления блока диалога возвращают либо указатель на соответствующий оконный объект, либо указатель на дескриптор окна этого объекта.

Теперь, после того как мы познакомились с возможностями класса *CDialog*, рассмотрим пример простейшего блока диалога.

# Создаем простейший блок диалога

Наш блок настолько прост, что даже нет необходимости создавать свой класс для его поддержки. Однако мы все-таки 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. 14.3. Сам блок диалога имеет вид, показанный на рис. 14.4. Чтобы не дублировать код, мы ограничимся только теми добавлениями, которые внесены в приложение MDI.

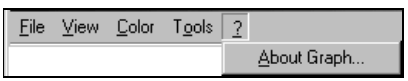



**Рис. 14.3.** Элемент "About Graph" меню "?"

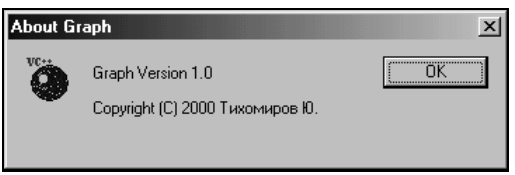



**Рис. 14.4.** Блок диалога

```
//////////////////////////////////////////////////////
// Файл CAppAbout.h
// Copyright (c) 2000 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 класса *CAppAbout*:

```
////////////////////////////////////////////////////////
// Файл CAppAbout.cpp
// Copyright (c) 2000 Тихомиров Ю.
////////////////////////////////////////////////////////
////////////////////////////////////////////////////////
// Блок диалога CAppAbout: реализация класса
// Конструктор класса
CAppAbout::CAppAbout(CWnd* pParent /*=NULL*/) :
  Dialog(CAppAbout::IDD, pParent) // этого можно и не делать
{
  Create(CAppAbout::IDD, pParent);
  //{{AFX_DATA_INIT(CAppAbout) // между этими операторными скобками
  //}}AFX_DATA_INIT            // код вставляет ClassWizard
}
// Для обмена данными с блоком диалога
void CAppAbout::DoDataExchange(CDataExchange* pDX)
{
  CDialog::DoDataExchange(pDX);
  //{{AFX_DATA_MAP(CAppAbout)  // см. предыдущий комментарий
  //}}AFX_DATA_MAP
}
// Карта сообщений блока диалога
BEGIN_MESSAGE_MAP(CAppAbout, CDialog)
  //{{AFX_MSG_MAP(CAppAbout)
  // No message handlers
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

**Примечание**

Чтобы не заниматься написанием кода самому, я воспользовался тем, который генерирует AppWizard. Единственная разница заключается в том, что код, связанный с классом *CAppAbout*, вынесен мной в отдельные файлы — AppWizard помещает его в файлы, связанные с классом приложения.

Изменения в классе *CMDIApp*:

```
////////////////////////////////////////////////////////
// Файл Graph.h
// Copyright (c) 2000 Тихомиров Ю.
////////////////////////////////////////////////////////
// Объект приложения CGraphApp: интерфейс класса

class CMDIApp : public CWinApp
{
...
// Implementation
  //{{AFX_MSG(CMDIApp)
  afx_msg void OnAppAbout();
  //}}AFX_MSG
DECLARE_MESSAGE_MAP()
};

////////////////////////////////////////////////////////
// Файл MDI.cpp
// Copyright (c) 2000 Тихомиров Ю.
////////////////////////////////////////////////////////
BEGIN_MESSAGE_MAP(CMDIApp, CWinApp)
  //{{AFX_MSG_MAP(CMDIApp)
  ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()
// Обработчик команды меню About
void CMDIApp::OnAppAbout()
{
  // Создаем объект класса
  CAppAbout aboutDlg;
  // Выводим на экран модальный блок диалога
  aboutDlg.DoModal();
}
```

В шаблон меню необходимо добавить следующий пункт:

```
...
POPUP "&?"
  BEGIN
    MENUITEM "About Graph ...",              ID_APP_ABOUT
  END
...
```

И, наконец, необходимо создать шаблон блока диалога, который должен иметь идентификатор — IDD_ABOUTBOX. Можете воспользоваться для этого одним из трех предлагаемых способов (на самом деле их больше):

- непосредственно вставить приведенный выше код описания блока диалога в файл ресурсов <graph.rc>;
- скопировать шаблон блока диалога из примера Graph в создаваемый вами;
- самостоятельно создать шаблон блока диалога с помощью графического редактора ресурсов.

**Примечание**

Последний способ не такой уж и сложный, как может показаться на первый взгляд, но зато самый эффективный.

Как видите, все достаточно просто. Блок диалога появляется на экране при выборе пользователем пункта "About Graph..." из меню справки. Заметим, что для вызова этого блока диалога такой путь является стандартным. После того как 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 изменить конструктор и обработчик команды ID_APP_ABOUT, как показано ниже:

```
CAppAbout::CAppAbout(CWnd* pParent)
{
...
 // Создаем объект Windows
 Create(CAppAbout::IDD, pParent);
...
}
void CMDIApp::OnAppAbout()
{
  // Создаем объект Windows
  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 наглядно иллюстрирует общий подход к созданию блоков диалога на основе шаблона, определенного в файле ресурсов. Теперь, опираясь на эти знания, рассмотрим пример приложения, который поможет нам изучить еще несколько новых интересных моментов:

- 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.

# Блок диалога в качестве главного окна приложения

Для реализации этой возможности нужно проделать всего три шага.

1. Определить в ресурсах шаблон блока диалога. Способы, которыми это можно сделать, мы подробно рассмотрели выше. На рис. 14.5 показан блок диалога для нашего приложения, созданный с помощью редактора ресурсов.

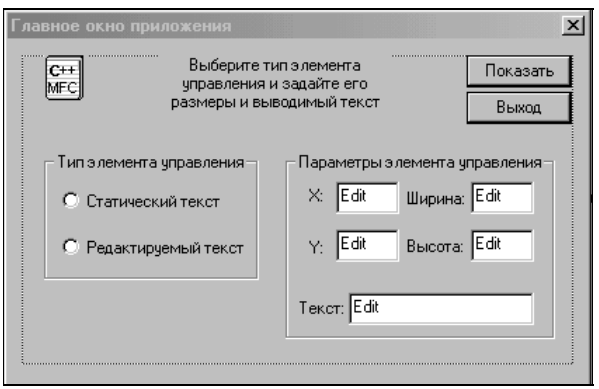



**Рис. 14.5.** Главное окно приложения является блоком диалога

2. Создать класс на основе класса *CDialog*

```
class CTemplateDlg : public CDialog
{
public:
  CTemplateDlg(CWnd* pParent = NULL);
...
};
```

3. Создать объект нового класса *CtemplateDlg*, что мы и делаем в переопределенной функции *InitInstance*:

```
BOOL CTemplateApp::InitInstance()
{
...
  // Создаем объект класса
  CTemplateDlg dlg;
  // Поскольку это главное окно приложения,
  // присваиваем указатель на него переменной m_pMainWnd
  m_pMainWnd = &dlg;
  // Выводим диалог на экран
  int nResponse = dlg.DoModal();
...
}
```

**Примечание**

Вы, наверное, обратили внимание, что я не говорю о необходимости создания класса приложения, т. к. это уже должно быть очевидно.

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 обмена данными с блоком диалога необходимо прежде всего определить в нашем классе соответствующие переменные, которые смогут принимать и передавать эти данные. Для этого воспользуемся услугами мастера ClassWizard:

1. Раскройте созданный шаблон блока диалога.
2. При нажатой клавише <Ctrl> дважды щелкните по тому элементу управления, с которым вы хотите сопоставить переменную.
3. На экране появится блок диалога Add Member Variable. В нем введите имя переменной m_x в поле Member variable name, в раскрывающемся списке Category можно оставить выбранное значение Value, а в раскрывающемся списке Variable type выберите UINT, поскольку координаты у нас представляются целым без знака (рис. 14.6). Нажмите кнопку OK, и переменная добавлена.

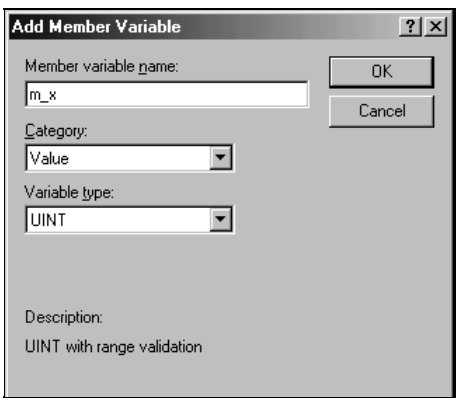



**Рис. 14.6.** Добавляем переменную, ассоциированную с элементом управления блока диалога

**Примечание**

Вполне возможна ситуация, когда вместо блока диалога Add Member Variable на экране появится блок диалога Adding a Class (рис. 14.7). Это произойдет в том случае, если, например, не создан класс для этого диалога, или вы успели заменить его идентификатор, о котором ClassWizard ничего не знает. В первом

случае нажмите кнопку OK и выполните необходимые действия для создания нового класса, а во втором — установите переключатель в положение Select an existing class и также выполните соответствующие инструкции.

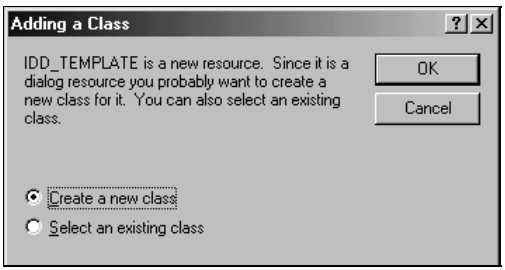



**Рис. 14.7.** Если не создан необходимый класс, то сделать это можно сейчас

4. Аналогичную процедуру повторите для всех остальных необходимых элементов управления блока диалога. В результате, когда вы откроете окно мастера ClassWizard и раскроете вкладку Member Variables, увидите все введенные переменные (рис. 14.8).

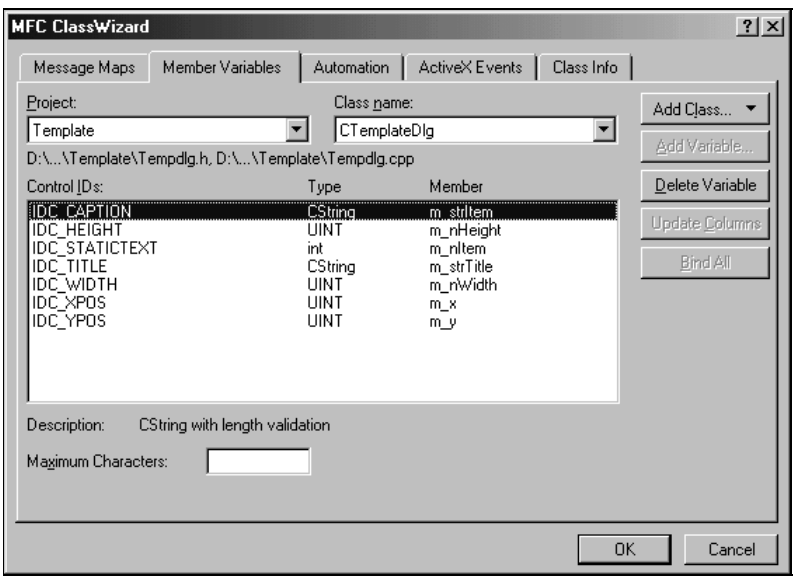



**Рис. 14.8.** Здесь можно посмотреть переменные, ассоциированные с элементами управления блока диалога

Вам осталось только посмотреть на код, который создал ClassWizard:

```
/////////////////////////////////////////////////////////////////////////
// Блок диалога CTemplateDlg
// Определение класса
class CTemplateDlg : public CDialog{
...
```

```
// Dialog Data
  // Специальный формат AFX комментариев для ClassWizard
  //{{AFX_DATA(CTemplateDlg)
    // Здесь ClassWizard располагает переменные
    // для обмена с блоком диалога
    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
...
// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CTemplateDlg)
protected:
  // Поддержка механизма обмена данными с блоком диалога — (DDX/DDV)
  virtual void DoDataExchange(CDataExchange* pDX);
//}}AFX_VIRTUAL
...
};
```

Мне представляется уместным привести список элементов управления и соответствующих им типов переменных:

| | |
|---|---|
| Элемент редактирования (EDITBOX) | CString, int, UINT, long, DWORD, float, double, short, BOOL |
| Обычный флажок (CHECKBOX) | BOOL |
| Флажок с тремя состояниями (CHECKBOX) | int |
| Первый в группе переключатель (RADIOBUTTON) | int |
| Несортированный 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:

```
// Реализация класса
CTemplateDlg::CTemplateDlg(CWnd* pParent)
      : CDialog(CTemplateDlg::IDD, pParent){
  ... // Здесь можно расположить инициализацию своих переменных
```

```
  // Специальный формат AFX комментариев для ClassWizard
  //{{AFX_DATA_INIT(CTemplateDlg)
    // Инициализация переменных для обмена с блоком диалога
    m_strTitle = _T("");
    m_x = 0;
    m_y = 0;
    m_nWidth = 0;
    m_nHeight = 0;
    m_nItem = -1;
    m_strItem = _T("");
  //}}AFX_DATA_INIT
... // а здесь другие необходимые действия
}
```

**Примечание**

Рассматриваемый ниже механизм справедлив для любого класса, производного от *CWnd*, а не только для блоков диалога. Однако поскольку чаще всего он применяется именно при работе с блоками диалога, я позволил себе использовать термин "обмен данными с блоком диалога".

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)`

Единственный 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. Данная функция 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. Основным назначением функции *CWnd::UpdateData* является проведение инициализации и обработки ошибок при обращении к функции

`virtual void CWnd::`**`DoDataExchange`** `(CDataExchange *pDX),`

которая, собственно, через ее параметр *pDX* (указатель на объект класса *CDataExchange*) и "занимается" обменом данных и проверкой их корректности. Эту функцию необходимо переопределить для каждого класса блока диалога, который должен обеспечивать взаимодействие с пользователем. Переопределенная функция имеет следующую стандартную форму, аналогичную приведенной в рассматриваемом приложении:

```
// Переопределенная функция,
// предназначенная для обмена данными с блоком диалога,
// которую также добавил ClassWizard
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
  }
```

Общая схема обмена данными наглядно представлена на рис. 14.9.

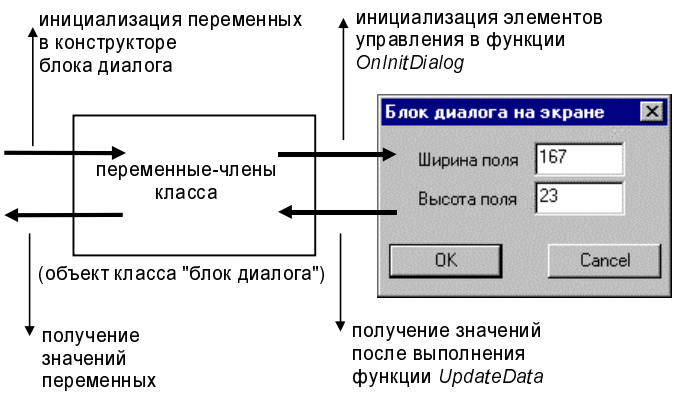



**Рис. 14.9.** Общая схема обмена данными в приложении

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`

Флаг указывает направление при операциях обмена данными (Dialog 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 (Dialog Data Validation, DDV) флаг всегда установлен в TRUE. Значение флага определяется параметром, передаваемым функцией *CWnd::UpdateData*.

После того как вы определите свою функцию *DoDataExchange* и не хотите или не можете воспользоваться услугами мастера ClassWizard, вам необходимо вставить в нее требуемые функции DDX, по одной на каждый элемент управления, участвующий в обмене. Библиотека MFC предоставляет доста-

точно 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 формат:

`DDX_XXX(pDX, nIDC, member)` —

для обмена данными

и

`DDV_XXX(pDX, member, ...)` —

для проверки корректности введенных данных

В этих функциях используются следующие параметры: XXX — имя соответствующей функции (например, *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.

Данный шаг, связанный с обменом данными в блоках 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 необходимо 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 "элемент редактирования"). После того как объекты созданы и проинициализированы, можно создавать соответствующие окна Windows. Эту задачу у нас выполняет функция *CMemory::Show*:

**Примечание**

Создание необходимых классов *CMemory* и *CItem* я не рассматриваю, поскольку с основой этого процесса мы уже познакомились.

```
void CMemory::Show()
{
  // При работе с шаблонами блоков диалога в памяти все
  // текстовые строки должны быть представлены в 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
  // и заголовка 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 память
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 Unicode строку
  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);
}
```

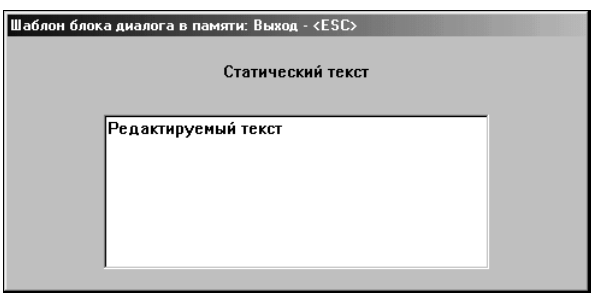



**Рис. 14.10.** Блок диалога, созданный функцией *CDialog::InitModalIndirect*

Функция *CMemory::Show* вызывается всякий раз, когда пользователь нажимает кнопку "Показать" в главном окне приложения. Параметры, вводимые

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. 14.10.

Понятно, что приведенный пример не претендует на изящество, однако свою роль в иллюстрации рассматриваемых положений он успешно выполняет.

# 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.

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 их создание. В настоящее время в библиотеку включены следующие классы: *CFileDialog*, *CColorDialog*, *CFontDialog*, *CFindReplaceDialog*, *CPrintDialog* и *CPageSetupDialog* (рис. 14.11).

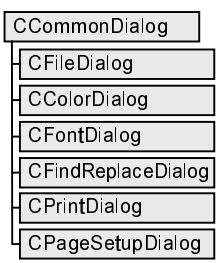



**Рис. 14.11.** Место стандартных блоков диалога в иерархии классов MFC

Работа с этими класса не представляет особых трудностей, и поэтому здесь я остановлюсь только на одном из них — классе *CColorDialog*, отвечающим за создание модального блока диалога для выбора и/или создания цветов, которые определены для вашего дисплея.

## Для выбора цветов не обязательно создавать новый класс

Класс *CColorDialog*, как и другие классы стандартных блоков диалога, продуман настолько хорошо, просто и удобно, что вряд ли у кого-нибудь возникнет желание создавать на его основе производный класс. Основным компонентом этого класса, позволяющим осуществлять настройку блока диалога, 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;
```

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* — 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. Поле *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. Поле *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. Поле *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*. Поле *lpfnHook* — указатель на функцию фильтра *CCHookProc*, которая может обрабатывать сообщения блока диалога. Это поле игнорируется, если не установлен флаг CC_ENABLEHOOK. Поле *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 макрос MAKEINTRESOURCE. Это поле игнорируется, если не установлен флаг CC_ENABLETEMPLATE. Поле *Flags* определяет множество битовых флагов, используемых 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 (Определить цвет), 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 (?) |

Из функций класса *CColorDialog* приведу описание наиболее часто используемых:

```
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 этой 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.

Приведу также пример настройки и работы со стандартным блоком диалога. Это будет последняя наша встреча с приложением Graph. И здесь мы добавим в него возможность выбирать цвет заливки не только из ограниченного набора контекстного меню, но и из всего диапазона поддерживаемых цветов. Я сознательно опускаю все вопросы, связанные с меню, картой сообщений и созданием соответствующего обработчика — в этом вы уже в состоянии разобраться самостоятельно. Ваше внимание я акцентирую только на содержимом обработчика. Но сначала фрагмент кода:

```
//////////////////////////////////////////////////////////////
// Файл ChildWnd.h
// Copyright (c) 2000 Тихомиров Ю.
//////////////////////////////////////////////////////////////
void CChildWnd::OnLineColor()
{
  // Прежде всего создаем объект класса
  CColorDialog clrDlg;
  // Размер структуры задавать обязательно, но это очень просто
  clrDlg.m_cc.lStructSize = sizeof(clrDlg.m_cc);
  // При открытии блока диалога мы хотим,
  // чтобы устанавливался текущий цвет линии
  clrDlg.m_cc.rgbResult = m_clrLine;
  // В качестве дополнительных цветов устанавливаем
  // описанные ранее 16 стандартных цветов
  clrDlg.m_cc.lpCustColors = colors;
  // Если не используются установки по умолчанию, не забывайте
  // устанавливать флаг CC_ENABLEHOOK для подключения необходимой
  // для обработки функции обратного вызова, даже если вы
  // не определяете для этого свою функцию
  clrDlg.m_cc.Flags = CC_ENABLEHOOK |
/*  Пока закомментируем эти флаги
                      CC_FULLOPEN    | // Выводим расширенный вариант
                                       // блока диалога
                      CC_RGBINIT;      // По умолчанию в блоке диалога
                                       // будет устанавливаться текущий
                                       // цвет линии
*/
  // Выводим блок диалога на экран ...
  if(clrDlg.DoModal() == IDOK)
  {
    // и в случае действительного выбора цвета — закрытие
    // блока диалога кнопкой OK
```

```
    // Изменяем текущий цвет линии,
    // запоминая его значение в специальной переменной
    m_clrLine = clrDlg.GetColor();
  }
}
```

Внешний вид рассматриваемого блока диалога представлен на рис. 14.12.

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. 14.13).

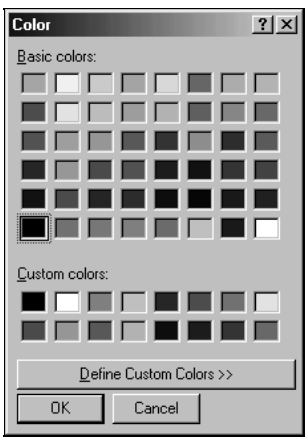



**Рис. 14.12.** Блок диалога для выбора цвета

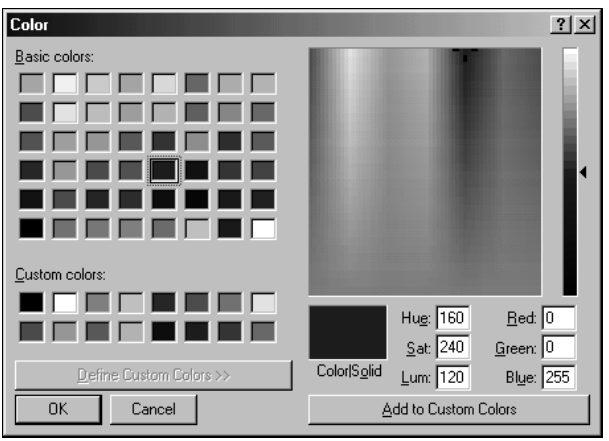



**Рис. 14.13.** Расширенный вариант блока диалога для выбора цвета

Как видите, все довольно просто, и мне осталось только добавить небольшой комментарий. А связан он со строкой

```
clrDlg.m_cc.lpCustColors = colors;
```

в которой в качестве дополнительных цветов (Custom colors) мы предлагаем использовать те, которые определили для работы с контекстным меню и которые уже присутствуют в базовой палитре. С одной стороны безразлично, с каких именно дополнительных цветов мы будем начинать работу с приложением. Но на самом деле мы поступили значительно хуже, просто отбросив эти шестнадцать дополнительных цветов. Вы уже догадались, в чем дело? Действительно, предположим, что мы "настроили" нужный нам цвет и записали его в качестве одного из дополнительных, после чего стали с ним работать. Но вот нам захотелось изменить цвет, и мы снова вызываем этот блок диалога. И что же? В качестве дополнительных цветов нам опять предлагаются все те же самые.

Вы согласны, что такое поведение может нас устроить только в простеньком, никому больше не нужном учебном примере? Тогда будем вносить необходимые изменения. И поможет нам функция *CColorDialog::GetSavedCustomColors*,

которая требует предварительного определения статического массива для дополнительных цветов. Сделаем ее также и глобальной:

```
...
static COLORREF *aCustom;
...
```

Теперь нам осталось только воспользоваться названной выше функцией и внести небольшое изменение в приведенный код. Все, что я внес нового, выделено полужирным текстом и, кроме того, убраны приводимые ранее комментарии:

```
void CChildWnd::OnLineColor()
{
  CColorDialog clrDlg;
  // Получаем доступ к массиву дополнительных цветов
  aCustom = clrDlg.GetSavedCustomColors();
  clrDlg.m_cc.lStructSize = sizeof(clrDlg.m_cc);
  clrDlg.m_cc.rgbResult = m_clrLine;
  clrDlg.m_cc.lpCustColors = aCustom;
  clrDlg.m_cc.Flags = CC_ENABLEHOOK | CC_FULLOPEN | CC_RGBINIT;
  if(clrDlg.DoModal() == IDOK)
  {
    m_clrLine = clrDlg.GetColor();
  }
}
```

Обратите внимание, всего две новые строчки, а результат совершенно другой — дополнительные цвета теперь не теряются (рис. 14.14).

На этом мы заканчиваем знакомство с блоками диалога и переходим к достаточно обширной теме, которой мы уже частично касались. Речь, как вы догадались, идет об элементах управления.

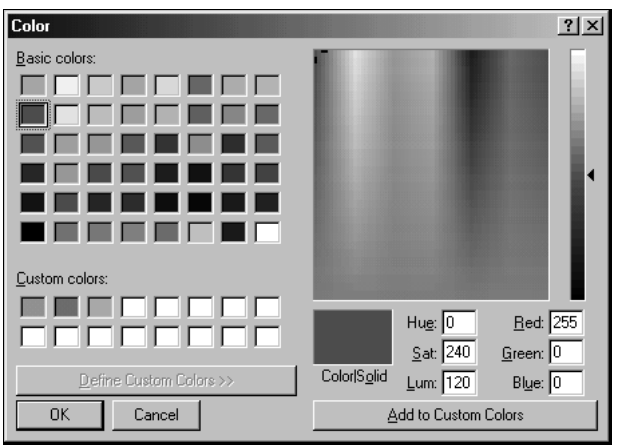



**Рис. 14.14.** Теперь можно пользоваться и дополнительными цветами без опасения их потерять



# ГЛАВА 15

# Наборы свойств

Когда мы пытаемся вытащить что-нибудь одно,
оказывается, что оно связано со всем остальным.
*Закон Муира*

*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, делает видимой соответствующую ей страницу свойств.

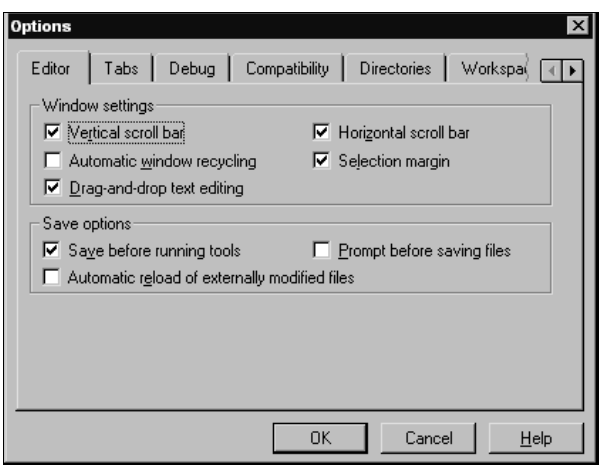



**Рис. 15.1.** Окно Options оболочки Visual C++ 6.0

В качестве примера набора свойств можно привести блок диалога просмотра и изменения параметров оболочки Visual C++ 6.0 (рис. 15.1).

Использовать набор свойств весьма удобно в тех случаях, когда имеются группы некоторых параметров, которые можно изменять. Он позволяет собрать 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. 15.2).




**Рис. 15.2.** 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, а *CPropertyPage* и *CPropertyPageEx* предназначены для создания включаемых в набор страниц свойств. Несколько особняком стоит класс *CTabCtrl*, используемый для создания вкладок и работы 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. Однако возможность доступа к объектам данного класса все же имеется, и его функции применяются в основном для создания вкладок, отличающихся от создаваемых по умолчанию.

**Примечание**

Классы *CPropertySheetEx* и *CPropertyPageEx* доступны только в Visual C++ 6.0.

Проще всего создать набор свойств, воспользовавшись средством добавления в проект различных программных компонентов. Для этого выберите команду Insert\Component, а затем вставьте в проект набор свойств с требуемыми характеристиками (рис. 15.3).

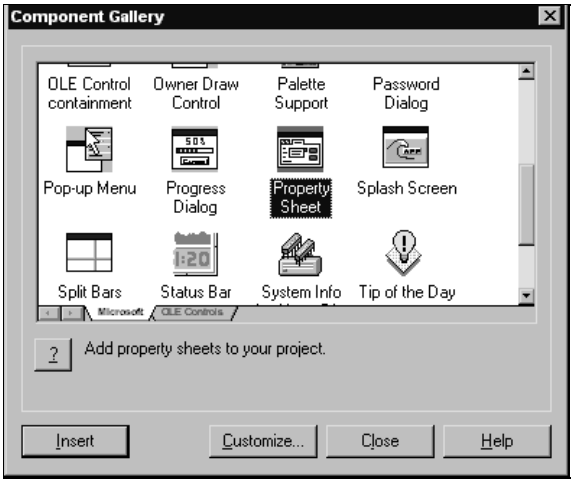



**Рис. 15.3.** Окно выбора добавляемых в проект программных компонентов

При этом требуется проделать следующее:

- определить, будет ли добавляемый компонент набором свойств или мастером;
- указать, поддерживается ли возможность предварительного просмотра произведенных изменений (по сути это добавление рисованного объекта и ничего больше);
- определить, будет ли набор свойств модальным или нет;
- задать класс из набора существующих в проекте классов, который будет иметь функцию создания набора свойств и возможность доступа к его элементам;
- задать количество страниц;
- заменить имена создаваемых классов (по желанию).

Если впоследствии понадобится добавить новые страницы свойств, то следует воспользоваться механизмом, изложенным ниже.

## Создание объекта "набор свойств"

Следующим шагом будет создание объекта класса *CPropertySheet* в тексте программы. Обычно такой объект создается в обработчике команды, осуществляющей вывод на экран набора свойств.

```
CPropertySheet propsheet;
CPropertySheet propsheet("Параметры", pParentWnd);
```

или

```
CPropertySheet propsheet("Параметры ", GetParent());
```

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)
```

Параметр *nIDCaption* — идентификатор заголовка для набора свойств; *pParentWnd* — указатель на родительское окно. Если его значение равно NULL, то родительским окном является главное окно приложения. Параметр *iSelectPage* — индекс страницы, выводимой первой при отображении набора свойств на экран. По умолчанию выводится первая страница набора свойств. Параметр *pszCaption* — указатель на строку, содержащую заголовок для набора свойств (не может быть равным NULL).

Если не нужно добавлять в набор свойств дополнительные элементы управления (например, окно предварительного просмотра) или выводить на экран немодальный набор свойств, то создавать класс, производный от класса *CpropertySheet*, вообще не требуется.

**Примечание**

Немодальный набор свойств по умолчанию не содержит кнопок, и в большинстве случаев в него потребуется добавить элементы управления для того, 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). Допускается добавлять и другие необходимые для работы элементы управления, например, дополнительное окно просмотра, в котором будут выводиться изображения, находящиеся в выбираемых пользователем файлах. Дополнительные элементы управления можно вводить 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.

Вставка дополнительных элементов управления в набор свойств обычно требует увеличения размеров его диалогового окна. Для этого после вызова функции базового класса *OnCreate* или *OnInitDialog* необходимо вызвать функцию *CWnd::GetWindowRect* для получения значений размеров 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 класса *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)
```

или

```
void CPropertySheet::Construct (
  LPCTSTR pszCaption,
  CWnd* pParentWnd = NULL,
  UINT iSelectPage = 0)
```

Параметры здесь те же, что и у конструктора класса *CPropertySheet*. Данную функцию можно вызывать, если ни один из конструкторов с параметрами класса *CPropertySheet* еще не вызывался. В случае создания массива объектов *CPropertySheet* вызов функции *Construct* обязателен для каждого элемента массива.

Ниже приведен пример, иллюстрирующий применение этой функции:

```
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:

- создание одного объекта каждого класса, производного от класса *CPropertyPage*, который ранее был создан с помощью ClassWizard;
- вызов функции *CPropertySheet::AddPage* для каждой страницы, добавляемой 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, для добавления страницы можно не ждать создания окна набора свойств, а вызвать функцию *AddPage* непосредственно из конструктора.

```
// 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

Создание набора свойств в виде модального блока диалога осуществляется вызовом функции *DoModal*:

```
virtual int CPropertySheet::DoModal ()
```

Выводит на экран модальный набор свойств. Функция возвращает значения IDOK или IDCANCEL, а в случае неуспешного завершения — 0. Следует иметь в виду, что если набор свойств создан как мастер, то возвращаются другие значения — ID_WIZFINISH или IDCANCEL.

После завершения работы функции окна и все страницы, связанные с набором свойств, удаляются. Сами объекты остаются существовать. Как правило, данные 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. Немодальный набор свойств формируется вызовом функции *CPropertySheet::Create*.

```
BOOL CPropertySheet::Create (
  CWnd* pParentWnd = NULL,
  DWORD dwStyle = (DWORD) -1,
  DWORD dwExStyle = 0)
```

Параметр *pParentWnd* указывает на родительское окно (если равен NULL, то родительским окном является главное окно приложения); *dwStyle* задает стили окна (по умолчанию — комбинация стилей WS_SYSMENU, WS_POPUP, WS_CAPTION, DS_MODALFRAME, DS_CONTEXT_HELP и WS_VISIBLE); а *dwExStyle* — расширенные стили окна (по умолчанию — стиль WS_EX_DLGMODALFRAME).

Функцию *CPropertySheet::Create* можно вызывать как из конструктора, так и после того, как объект набора свойств уже создан. Функция возвращает управление непосредственно после создания набора свойств. Для уничтожения набора свойств следует использовать 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. Од-

ним из методов обеспечения обмена является определение указателя из набора свойств на внешний объект. Далее следует определить функцию в производном от *CPropertySheet* классе, которая будет отслеживать изменения данных набора свойств всякий раз, когда фокус ввода передается от одного внешнего объекта к другому. Такая функция вызывается, естественно, не объектом набора свойств, а самим приложением и осуществляет сброс текущих данных каждой страницы для отображения свойств текущего внешнего объекта, для чего она должна иметь доступ к объектам *CPropertyPage*, определенным в классе *CPropertySheet*.

**Примечание**

Если вы, как и я, воспользовались услугами Galery Components, то всю эту работу за вас проделает мастер, и полученные в последних разделах сведения понадобятся только в случае, когда вы захотите расширить возможности созданного набора свойств.

## Обмен данными

Кроме вывода на экран набора свойств, следует обеспечить также 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 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, отсутствующую у обычных блоков диалога, а именно — позволяет пользователям изменять задаваемые на странице свойств параметры во внешнем объекте до того, как окно с набором свойств будет закрыто. Это осуществ-

ляется с помощью кнопки Apply, нажатие которой приводит к изменению у объекта *всех* параметров *всех* страниц (а не только текущей активной страницы) в наборе свойств. По умолчанию эта кнопка недоступна. Необходимо самостоятельно написать код программы, делающий ее доступной для использования в требуемый момент работы с набором свойств. Следует также написать код, реализующий выполнение необходимых действий при нажатии этой кнопки.

Если не требуется предоставлять пользователю описанную возможность, то в общем случае нет необходимости удалять саму кнопку. Ее можно просто оставить недоступной для того, чтобы сохранить общность с другими приложениями Windows. Чтобы объявить об изменении параметров в странице свойств 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. Функцию *SetModified* следует вызывать всякий раз, когда пользователь внес какие-либо изменения в параметры. Одним из способов обнаружения этого действия 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 становится недоступной, для чего нужно вызвать функцию *SetModified* со значением параметра, равным FALSE. Причем сделать это необходимо для каждой из страниц свойств, для которых функция *SetModified* вызывалась ранее со значением параметра, равным TRUE, иначе 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.

Обработчик команды Apply описывается следующим кодом:

```
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);
}
```

Запретить отображение кнопки Apply очень просто:

```
void [11]CMainFrame::OnProperties()
{
  // Создаем объект класса
  CMyPropertySheet propSheet;
  // Удаляем кнопку Apply
  propSheet.m_psh.dwFlags |= PSH_NOAPPLYNOW;
  // Выводим набор свойств на экран
  propSheet.DoModal();
}
```

Далее мы приводим очень краткий обзор классов *CPropertySheet* и *CPropertyPage*, используемых для создания набора свойств, а также новых — *CPropertySheetEx* и *CPropertyPageEx*, реализующих некоторые дополнительные возможности.

## 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)
```

Параметр *pPage* — указатель на добавляемую страницу (не может быть равным NULL). 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 вызывается после вывода набора свойств на экран, 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)
```

Параметр *pPage* в первой версии — указатель на удаляемую страницу (не может быть равен NULL), а во второй — 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 в классе *CPropertySheet* реализованы следующие функции: *GetActiveIndex*, *GetPage*, *GetPageIndex*, *GetPageCount*, *GetActivePage*, *SetActivePage*, *SetTitle*.

Следует отдельно отметить функцию, позволяющую получить доступ к вкладкам набора свойств:

`CTabCtrl* CPropertySheet::GetTabControl ()`

Возвращает указатель на объект класса *CTabCtrl*.

Вкладки входят в набор свойств неявно 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.

Для завершения работы набора свойств служит функция:

`void CPropertySheet::EndDialog (int nEndID)`

Параметр *nEndID* задает значение, которое возвращает как результат своей работы функция *DoModal*.

Функция *EndDialog* вызывается библиотекой, когда окно набора свойств закрывается нажатием кнопок OK, Cancel или 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.

## Изменение параметров отображения

В заключение описания класса *CPropertySheet* приведем пример изменения стилей и параметров отображения набора свойств. На рис. 15.4 представлен набор свойств с установленными по умолчанию параметрами вывода.

Для изменения некоторых флагов (параметров 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*.

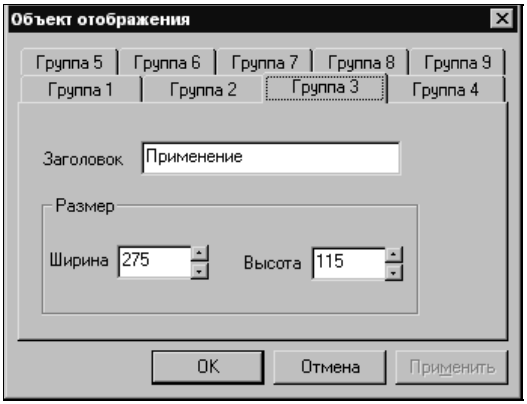



**Рис. 15.4.** Набор свойств со значениями параметров вывода, заданными по умолчанию

Вот как можно, например, удалить кнопку Apply из окна набора свойств:

```
CMyPropertySheet::CMyPropertySheet(CWnd* pWndParent)
       : PropertySheet(IDS_PROPSHT_CAPTION, pWndParent)
{
  // Удаляем кнопку Apply
  m_psh.dwFlags |= PSH_NOAPPLYNOW;
  ...
}
```

Как видите, для настройки набора свойств используется компонент класса *m_psh*:

```
PROPSHEETHEADER CPropertySheet::m_psh
```

В структуре *PROPSHEETHEADER* хранятся основные параметры набора свойств. Ее можно использовать для изменения этих параметров уже после создания набора, 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
```

Поле *dwSize* — размер структуры в байтах; *dwFlags* — массив флагов, указывающих, какие переменные данной структуры должны быть использованы, а какие проигнорированы. Может быть комбинацией следующих значений:

| | |
|---|---|
| PSH_DEFAULT | Использовать значения 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 и 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 его установленным, но потом могут быть 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. Флаг PSH_HEADER игнорируется, если одновременно не установлен флаг PSH_WIZARD97 |
| PSH_MODELESS | Сообщает функции *::PropertySheet*, что набор свойств необходимо создать как немодальный блок диалога (по умолчанию соз- |

| | |
|---|---|
| | дается модальный). Если установлен этот флаг, то функция *::PropertySheet* немедленно после создания диалога возвращает дескриптор набора свойств. (При использовании класса *CPropertySheet* манипуляции флагом осуществляются 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 | Устанавливает, что 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*.

Поле *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). Поле *pszIcon* предназначено 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). Поле *pszCaption* задает заголовок набора свойств, 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:"). Поле *nPages* — число элементов в массиве *phpage*. Поле *nStartPage* задает индекс, а поле *pStartPage* — имя страницы, которая появится на экране первой при создании набора свойств (параметр должен содержать идентификатор строкового ресурса или указатель на строку, содержащую имя). Поле *ppsp* — указатель на массив структур PROPSHEETPAGE, определяющих страницы набора свойств. Если в *dwFlags* не задано значение PSH_PROPSHEETPAGE, то данный параметр будет проигнорирован. Поле *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). Поле *pfnCallback* — указатель на функцию приложения, которая вызывается при инициализации набора свойств (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).

Для изменения способа отображения вкладок следует использовать функцию *OnInitDialog*:

```
BOOL CMyPropertySheet::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 более правильный способ (введенный в последней версии библиотеки MFC), при котором изменение стиля вывода вкладок осуществляется кодом библиотеки.

```
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.

```
BOOL CMyPropertySheet::OnInitDialog()
{
  // Устанавливаем стиль вывода вкладок в одну линию
  EnableStackedTabs(TRUE);
```

```
  BOOL bResult = CPropertySheet::OnInitDialog();
  return bResult;
}
```

Результат использования функции *EnableStackedTabs* приведен на рис. 15.5.

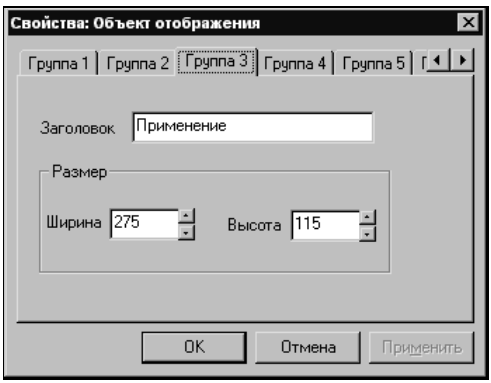



**Рис. 15.5.** Набор свойств после изменения некоторых значений параметров вывода

В последней версии библиотеки MFC добавлен новый класс набора свойств — *CPropertySheetEx*, который 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.

Вторым классом, используемым для создания включаемых в набор свойств страниц, является класс *CPropertyPage*.

# Настройка страниц набора свойств

Выше мы говорили, что объекты класса *CPropertyPage* представляют собой отдельные страницы, входящие в набор свойств. Прежде всего должен быть создан сам объект *CPropertySheet*, а затем необходимо создать объекты класса *CPropertyPage* для каждой вставляемой в набор свойств страницы и 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:

```
PROPSHEETPAGE CPropertyPage::m_psp
```

Эту структуру можно использовать для изменения параметров страницы уже после ее создания, но до того, как она выведена на экран.

## Создание страницы свойств

Класс *CPropertyPage* имеет три версии конструктора:

```
CPropertyPage::CPropertyPage (),
CPropertyPage::CPropertyPage (
  UINT nIDTemplate,
  UINT nIDCaption = 0)
```

и

```
CPropertyPage::CPropertyPage (
  LPCTSTR lpszTemplateName,
  UINT nIDCaption = 0)
```

Параметр *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*.

## 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. Параметр *bChanged* может иметь 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.

## Переопределяемые функции

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`** `()`

Принимаются *все* изменения для *всех* страниц набора свойств, набор свойств теряет фокус ввода, и функция возвращает значение TRUE. Чтобы данная функция могла быть вызвана, необходимо прежде вызвать функцию *SetModified* со значением параметра, равным TRUE, которая делает доступной кнопку Apply, если пользователь произвел изменения параметров в странице свойств. Реализация 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 произвести 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 *OnApply* при активной второй странице приводит к ее вызову 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 при этом необходимо дополнительно совершать какие-либо действия.

Как я уже неоднократно упоминал, в библиотеку MFC добавлены несколько новых классов, в число которых входит и *CPropertyPageEx*, расширяющий возможности работы со страницами свойств.

Этот новый класс библиотеки MFC предоставляет поддержку для расширенной структуры PROPSHEETPAGE, введенной в Windows 98 и Windows NT и содержащей дополнительные флаги и компоненты для работы с широкой областью заголовка, приспособленной для размещения как самого заголовка, так и подзаголовка.

# 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. В отличие от обычного набора свойств, мастер не имеет кнопок OK и Apply (Применить), а содержит кнопки Back (Назад), Next (Далее) (рис. 15.6) или Finish (Готово), Cancel (Отмена) и Help (Справка).

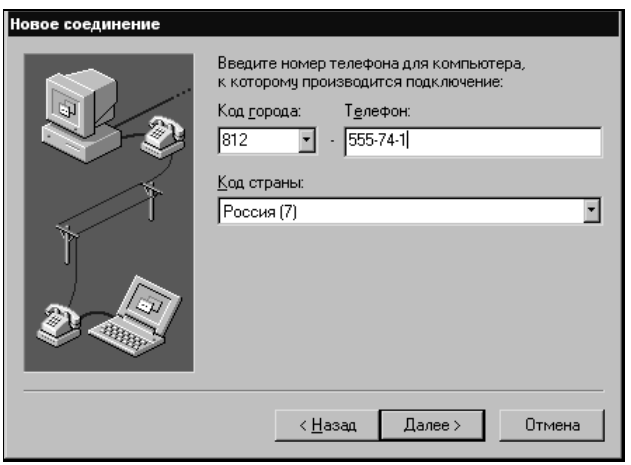



**Рис. 15.6.** Типичное окно мастера

## Создание мастеров

Для создания мастера нужно следовать шагам создания обычного набора свойств, но прежде, чем вызывать функцию *CPropertySheet::DoModal*, следует вызвать функцию *SetWizardMode*:

`void CPropertySheet::`**`SetWizardMode`** `()`

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 устанавливает флаг PSF_WIZARD.

Чтобы сделать доступными 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*.

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 кнопка Next. Обе 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. Если функция возвращает значение TRUE, то набор свойств не уничтожается. Данную функцию можно переопределить, если необходимо выполнить какие-либо действия при нажатии пользователем кнопки Finish. При этом функция должна вернуть значение TRUE для того, чтобы набор свойств не был уничтожен.

Вот, собственно, и все, что я хотел рассказать о наборах свойств и мастерах. С примерами использования и работы с наборами свойств мы познакомимся в следующей главе, где предметом изучения станут элементы управления.

# ГЛАВА 16



# Эти разнообразные элементы управления

Чем сложнее действие, тем проще персонажи.
*Карел Чапек*

Библиотека MFC предоставляет набор классов, которые облегчают создание стандартных элементов управления Windows: кнопок различного вида, статического текста, поля редактирования текста, полос прокрутки, списка и комбинированного списка. В табл. 16.1 приведены названия классов и элементы управления, создаваемые этими классами. Иерархия классов, связанных с элементами управления, представлена на рис. 16.1.

***Таблица 16.1.*** *Классы MFC для создания стандартных элементов управления*

| Класс | Элемент управления |
|---|---|
| CStatic | Статический текст или изображение |
| CButton | Командные кнопки, срабатывающие по нажатию, переключатели и флажки |
| CBitmapButton | Кнопки с заданными рисунками для всех возможных состояний |
| CListBox | Список |
| CCheckListBox | Список элементов, реализованных в виде флажков |
| CDragListBox | Список с возможностью переноса его элементов при помощи мыши |
| CComboBox | Комбинированный блок списка и текстового поля |
| CComboBoxEx | Комбинированный список с возможностью непосредственной работы со списками изображений |

***Таблица 16.1*** *(окончание)*

| Класс | Элемент управления |
|---|---|
| CEdit | Текстовое поле |
| CScrollBar | Полоса прокрутки |

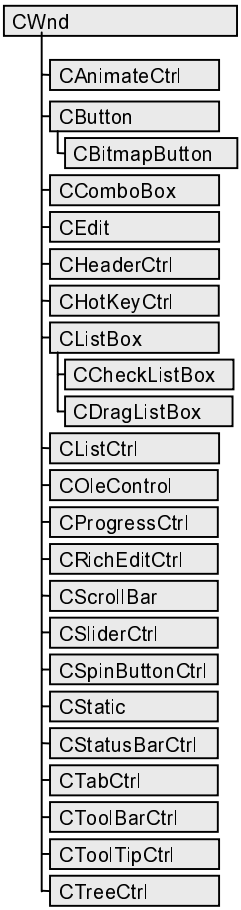



**Рис. 16.1.** Место элементов управления в иерархии классов MFC

# Создание элементов управления

Практически все элементы управления можно создать двумя способами:

- в редакторе ресурсов,
- непосредственно в коде приложения.

## Создание элементов управления в редакторе ресурсов

Элементы управления можно создавать во время создания шаблона блока диалога при помощи редактора ресурсов. При этом пользователь просто перетаскивает с помощью мыши необходимые элементы управления из панели с их изображениями в блок диалога, изменяя по своему желанию размеры, расположение и параметры выбранного элемента управления. Редактор ресурсов помещает описание нового элемента блока диалога в файл ресурсов. При реализации блока диалога вызовом функций *Create* или *DoModal* система Windows сама создает элементы управления и размещает их в этом блоке диалога, причем это верно для всех окон на базе шаблона блока диалога — модальных и немодальных блоков диалога и форматных представлений.

## Создание элементов управления в тексте приложения

Элементы управления можно задать непосредственно в исходном тексте приложения. Как и другие объекты классов библиотеки MFC, инкапсули-

рующие визуальные объекты Windows, элементы управления создаются в два этапа: сначала определяется объект данного конкретного класса, а затем вызывается функция *Create* для реализации соответствующего элемента управления Windows и связывания его с объектом класса. Обычно для блока диалога это выполняется в функции *OnInitDialog*, для обычного окна — в функции *OnCreate*, а для набора свойств — в одной из функций *OnSetActive* или *OnInitDialog* соответствующей страницы.

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* задается прямоугольник, определяющий размеры и расположение данного объекта, 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* вызывается функция *SetFocus* для передачи фокуса ввода новому полю редактирования.

Для реализации элементов управления можно в равной мере пользоваться как первым способом, используя редактор ресурсов (этот способ применяется чаще), так и вторым, описывая элементы управления непосредственно в тексте программы.

Единственное, что следует помнить — это то, что если элемент управления создается в блоке диалога, то он автоматически уничтожается, когда пользователь закрывает диалоговое окно, а если элемент управления создается программно внутри окна, то его необходимо уничтожать "вручную". Если он создан с помощью функции *new*, то, соответственно, для уничтожения необходимо вызывать функцию *delete*.

Теперь перейдем непосредственно к элементам 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, 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).

Из всех возможностей мы рассмотрим только одну, которая требует написания некоторого кода.

## Изменение цвета

Следующий пример иллюстрирует, как на странице набора свойств в ответ на сообщение WM_CTLCOLOR изменяется "кисть" для закрашивания фона статических элементов управления.

Для этого необходимо создать обработчик ON_WM_CTRLCOLOR. Проще всего воспользоваться мастером 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.

```
...
BOOL CMyPropertyPage2::OnInitDialog()
{
  CPropertyPage::OnInitDialog();
  m_brush.CreateHatchBrush(HS_CROSS, RGB(255, 255, 0));
  return TRUE;
}
```

```
// Обработчик сообщения WM_CTLCOLOR
HBRUSH CMyPropertyPage2::OnCtlColor(
                              CDC* pDC, CWnd* pWnd, UINT nCtlColor)
{
  // Получаем дескриптор текущей кисти страницы свойств
  HBRUSH hbr = CPropertyPage::OnCtlColor(pDC, pWnd, nCtlColor);
  // Заменяем "кисть" только для статических
  // элементов управления
  if (nCtlColor == CTLCOLOR_STATIC)
    hbr = m_brush;
  return hbr;
}
...
```

В результате страница свойств будет иметь следующий вид (рис. 16.2). Обратите также внимание, что на странице представлены два статических элемента управления. С первым мы только что поработали, а второй — некоторый рисунок, и для его отображения я не привел ни одного фрагмента кода. Вполне достаточно оказалось сделать все определения в редакторе ресурсов: сначала создать ресурс BITMAP, импортировав его из заранее подготовленного рисунка, а затем выбрать его в качестве свойства статического элемента управления (рис. 16.3).

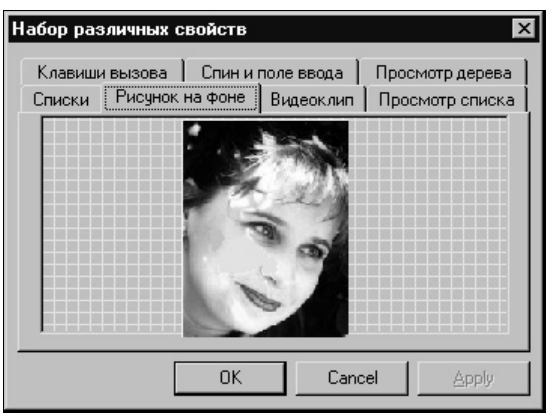



**Рис. 16.2.** Страница свойств со статическими элементами управления




**Рис. 16.3.** Статический элемент управления будет представлять собой рисунок

Мы не будем рассматривать элемент управления "кнопка", поскольку работа с ним элементарна, а пример реализации вы найдете и в этой главе, и в главе 14, где мы рассматривали работу с блоками диалога.

# Список

С помощью класса *CListBox* можно создать элемент управления Windows "список", представляющий собой прямоугольник с находящейся внутри последовательностью тестовых или пользовательских элементов, которые пользователь может пролистывать и выбирать. Список используется, например, для просмотра имен файлов на диске.

Бывают списки, позволяющие выбирать только одну запись (элемент) или одновременно несколько записей (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 всех них предусмотрены специальные макросы, добавляющие в карту сообщений окна функции обработки. Большинство уведомлений посылается родительскому окну тогда и только тогда, когда список является текущим элементом управления.

## Изменение параметров списка

Не все параметры списка можно задать при помощи редактора ресурсов или функции *Create*. Далее приведены описания функций 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 пикселах, 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 пикселах), в пределах которой список может прокручиваться. Если размер окна списка меньше этого значения, то строки в списке можно 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)
```

Параметр *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 пикселах. 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 пикселах. 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 пикселах) в списке, имеющем несколько столбцов (созданном со стилем LBS_MULTICOLUMN), где *cxWidth* задает ширину в пикселах всех столбцов.

## 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 и исключая таким образом необходимость реализации функций отображения элементов, что особенно важно для самоотображаемых элементов управления.

Начнем с наиболее часто используемой функции, которая добавляет строку текста в качестве элемента в список:

```
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 при создании списка этот стиль был задан, то после добавления строки 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.

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 ускоряет 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 используется обычное выделение памяти для записей, которым не хватило выделенного объема.

Как и любой список вообще, "список" как элемент управления Windows и вместе с ним класс *CListBox* требуют наличия, 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 выделяет найденную строку, причем если строка найдена, то она выделяется и, если нужно, список прокручивается так, 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.

Ктоме того, для поиска строк существует еще одна функция, но в отличие от предыдущих она осуществляет поиск строки в списке, *точно* совпадающей с заданной строкой:

```
int CListBox::FindStringExact (
    int nIndexStart,
    LPCTSTR lpszFind)
```

Назначение параметра *nIndexStart*, несмотря на отличающееся название, полностью аналогично назначению параметра *nStartAfter* предыдущих функций. Если список создан как самоотображаемый, но без стиля LBS_HASSTRINGS, то функция *FindStringExact* попытается согласовать 32-битовое значение, связанное с элементом, со значением параметра *lpszFind* (указатели в Win32 API 32-разрядные).

## 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 предназначены 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 MFC 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. Это значение может быть больше, чем размерность буфера (параметр *rgIndex*), заданная параметром *nMaxItems*. 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*, возвращающая 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.

На первый взгляд может показаться, что элемент управления "список" предназначен, в основном, для текстовых элементов или для визуализации в виде списка произвольных данных, как правило, временного использования. В отличие от чисто "программных" классов-контейнеров, в реализации описываемых списков основное внимание уделено визуализации и они обладают меньшими возможностями 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, наконец, последняя функция этой группы позволяет 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 со стилями 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. (Мы уже упоминали эту структуру, рассматривая работу с меню.)

Функция определения размеров элементов списка:

```
virtual void CListBox::MeasureItem (LPMEASUREITEMSTRUCT lpMeasureItem-
Struct)
```

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 система Windows могла определить размеры списка. Если список создан со стилем LBS_OWNERDRAWVARIABLE, то функция *MeasureItem* вызывается для каждой записи списка, если нет, то функция будет вызвана всего один раз.

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 задается 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 |

# 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*. В любом случае необходимо переопределение функций *CListBox::CompareItem*, *CCheckListBox::DrawItem* и *CCheckListBox::MeasureItem*.

## Функции для работы с расширенным списком

Следующая пара функций позволяет программно 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 и может принимать следующие значения: 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;
}
```

Кроме того, в классе *CCheckListBox*, естественно, переопределены и функции, отвечающие за самоотображение элементов списка: *DrawItem* и *MeasureItem*.

Следующий пример иллюстрирует создание самоотображаемого, содержащего флажки списка с использованием класса *CChecklistBox*. Здесь я привожу только те фрагменты, которые мы должны добавлять сами, поскольку все остальное сделает ClassWizard.

```
class CMyPropertyPage1 : public CPropertyPage
{
...
  // Для реализации самоотображаемого списка необходимо создать свой
  //  класс, чтобы переопределить соответствующие функции
  CColorBox m_CheckBox;
  // Данные страницы
  //{{AFX_DATA(CMyPropertyPage1)
    enum { IDD = IDD_PROPPAGE1 };
```

```
    CVListBox  m_List;
  //}}AFX_DATA
...
};
```

Как видите, необходимо создать два новых класса, чтобы "перехватить" отображение элементов списка:

```
class CColorBox : public CCheckListBox
{
// Construction
public:
  CColorBox();
  virtual ~CColorBox();
// Operations
public:
  // Эту функцию надо будет реализовать самостоятельно
  void AddColorItem(COLORREF color);
// Overrides
// ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CColorBox)
public:
  virtual int CompareItem(LPCOMPAREITEMSTRUCT lpCompareItemStruct);
  virtual void DrawItem(LPDRAWITEMSTRUCT lpDrawItemStruct);
  virtual void MeasureItem(LPMEASUREITEMSTRUCT lpMeasureItemStruct);
  //}}AFX_VIRTUAL
...
};
class CVListBox : public CListBox
{
...
// Overrides
// ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CVListBox)
public:
  virtual void DrawItem(LPDRAWITEMSTRUCT lpDrawItemStruct);
  virtual void MeasureItem(LPMEASUREITEMSTRUCT lpMeasureItemStruct);
  //}}AFX_VIRTUAL
...
};
```

После того как классы созданы, необходимо обеспечить их реализацией соответствующих функций:

```
// Минимальная величина обрамления пиктограммы в элементе списка
#define FRAME  2
...
```

```
void CVListBox::DrawItem(LPDRAWITEMSTRUCT lpDrawItemStruct)
{
  CDC dc;
  // Контекст устройства уже был создан, и его надо только подключить
  dc.Attach(lpDrawItemStruct->hDC);
  CRect rect = lpDrawItemStruct->rcItem;
  // Смещение для вывода текста (32 — размер пиктограммы)
  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 пиктограмму, центрируя ее 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 перегрузить, но, т. к.
// она вызывается только при добавлении элемента в список,
```

```
// а наша реализация настраивает высоту элемента при каждом
// отображении, реализация ее ничего не делает
void CVListBox::MeasureItem(LPMEASUREITEMSTRUCT lpMeasureItemStruct)
{
}
```

И добавки для класса *CColorBox*:

```
#define COLOR_ITEM_HEIGHT 20
...
void CColorBox::AddColorItem(COLORREF color)
{
  // Добавляем строку в список
  AddString((LPCTSTR) color);
}
// Переопределяем функцию для сравнения строк списка
int CColorBox::CompareItem(LPCOMPAREITEMSTRUCT lpCompareItemStruct)
{
  // return -1 = элемент 1 расположен перед элементом 2
  // return 0  = элемент 1 и элемент 2 не различаются
  // return 1  = элемент 1 расположен после элемента 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 (выбрано/не выбрано)
// рисуют 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)))
  {
    // Выделение рамки, 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))
  {
    // Удаление рамки, 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;
}
```

Нам осталось только создать два элемента управления типа *ListBox* на странице набора свойств и добавить для этой страницы соответствующий код:

```
BOOL CMyPropertyPage1::OnInitDialog()
{
  CPropertyPage::OnInitDialog();
  // 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 RGB-составляющей
  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));
  CRect rect;
  GetClientRect(&rect);
  rect.left = rect.right/2 - 24;
  rect.right -= 8;
  rect.top += 10;
  rect.bottom -= 10;
  m_List.MoveWindow(&rect);
  // Настраиваем начальные размеры для всех элементов списка
  // (в нашем примере, в принципе, это делать не обязательно)
  m_List.SetItemHeight(0, 40);
  // Добавляем три элемента в список
  m_List.AddString(_T("Короткий текст"));
  m_List.AddString(_T("Средний по длине текст\nв две строки"));
  m_List.AddString(_T("Длинная строка, для отображения которой "
                      "необходимо несколько строк"));
  return TRUE;
}
```

После запуска приложения на экран будет выведен набор свойств. Выбрав страницу "Списки", вы увидите список, в котором строки будут иметь различные цвета, и список, содержащий текст и пиктограмму (рис. 16.4).

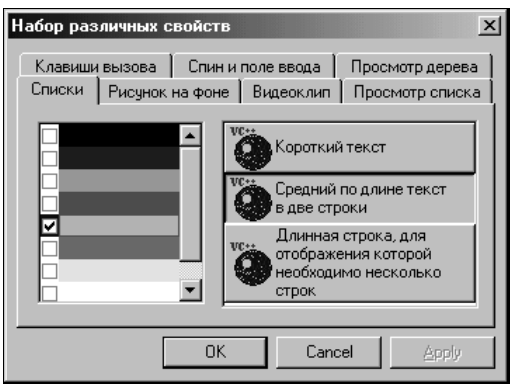



**Рис. 16.4.** Страница свойств с различными списками

**Примечание**

Обратите внимание на то, что связывание члена *CCheckBoxDlg::m_CheckBox* происходит без механизма обмена данными DDX (Dialog Data Exchange). Дело

в том, что мастер ClassWizard не поддерживает класс *CCheckListBox*. (В примере класс *CColorBox* изначально был создан на основе класса *CListBox*, затем вручную имя базового класса было заменено на *CCheckListBox*.)

# 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*, или страница набора свойств), то необходимо добавить в карту сообщений этого окна соответствующий обработчик.

Работа с этим элементов управления достаточно проста и мы не будем останавливаться на нем подробно.

# Просмотр видеоклипов

Класс *CAnimateCtrl* поддерживает функциональные возможности элемента управления Windows, который реализует просмотр видеоклипов (элемент управления "анимация").

Система 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 объектов Windows — MCIWnd.

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"

Фирма 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.

В библиотеке MFC для этого реализован 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 режиму просмотра видеоклипов.

```
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*, а для перемещения к определенному кадру, но без инициализации "проигрывания" — функцию *Seek*:

```
BOOL CAnimateCtrl::Seek (UINT nTo)
```

Единственный параметр *nTo* 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 MFC существует макрос ON_CONTROL, при помощи которого определяют элемент уведомления, тип уведомления и функцию его обработки.

Пример, созданный на основе страницы свойств, приводится после описания элемента управления "Индикатор".

# 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 реализуют индикацию двумя способами: добавляют элемент индикации в блок диалога либо используют его в специально созданных панелях диалога. И тот, и другой способ подразумевает использование шаблонов блока диалога, что исключает необходимость создания функции для определения окна индикатора. Библиотека 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* (при задании элемента в шаблоне блока диалога ее использование не требуется).

Реализация процесса и его взаимосвязи с индикацией полностью ложится на 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. Для задания интервала класс *CProgressCtrl* 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 нижнюю и верхнюю границы интервала, 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 действий 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 вместе с блоком 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.

# Пример использования просмотра видеоклипов и индикатора

Давайте рассмотрим страницу свойств, которая содержит два элемента управления для просмотра одного и того же видеоклипа, а также индикатор. "Проигрывание" видеоклипа осуществляется попеременно то одним элементом управления, то другим. Переключение осуществляется по завершению работы одного из элементов. Для индикатора создаются два таймера. Задача первого — имитация начала и окончания некоторой операции, второго — имитация собственно операции. Для наглядности в код "начала операции" добавлены функции начальной инициализации индикатора.

```
class CMyPropertyPage3 : public CPropertyPage
{
...
// Данные
  //{{AFX_DATA(CMyPropertyPage3)
    enum { IDD = IDD_PROPPAGE3 };
    CAnimateCtrl  m_Animate_2;
    CAnimateCtrl  m_Animate_1;
  //}}AFX_DATA
```

```
// Overrides
// ClassWizard generate virtual function overrides
  //{{AFX_VIRTUAL(CMyPropertyPage3)
public:
  virtual BOOL OnSetActive();
//  virtual BOOL OnKillActive();
protected:
  virtual void DoDataExchange(CDataExchange* pDX);
  //}}AFX_VIRTUAL
// Implementation
protected:
  // Generated message map functions
  //{{AFX_MSG(CMyPropertyPage3)
    afx_msg void OnTimer(UINT nIDEvent);
    afx_msg void AnimateStop_1();
    afx_msg void AnimateStop_2();
   //}}AFX_MSG
...
};
```

**Примечание**

Обратите внимание, что в данном случае инициализация страницы производится специальной функцией *OnSetActive* вместо использованной ранее функции родительского класса *OnInitDialog*. Такой подход на самом деле является более правильным.

```
// Функция обработки уведомления от первого элемента
// управления о завершении просмотра — начинаем просмотр
// во втором элементе управления
void CMyPropertyPage3::AnimateStop_1()
{
  m_Animate_2.Play(0, (UINT)-1, 1);
}
// Функция обработки уведомления от второго элемента
// управления о завершении просмотра — начинаем просмотр
// в первом элементе управления
void CMyPropertyPage3::AnimateStop_2()
{
  m_Animate_1.Play(0, (UINT)-1, 1);
}
// Инициализируем страницу свойств
BOOL CMyPropertyPage3::OnSetActive()
{
  // "Открываем" клип для первого элемента управления
  m_Animate_1.Open(IDR_VISUAL);
```

```
  // и немедленно начинаем его просмотр
  m_Animate_1.Play(0, (UINT)-1, 1);
  // "Открываем" клип для второго элемента управления
  m_Animate_2.Open(IDR_VISUAL);
  // Получаем указатель на объект, созданный в редакторе ресурсов
  CProgressCtrl *pProgress =
    (CProgressCtrl)GetDlgItem(IDC_PROGRESS);
  // Устанавливаем диапазон, начальную позицию и шаг
  pProgress->SetRange(0, 1000);
  pProgress->SetPos(0);
  pProgress->SetStep(10);
  SetTimer(ID_PULSE, 5000, NULL);
  SetTimer(ID_INPROG,  50, NULL);
  return CPropertyPage::OnSetActive();
}
void CMyPropertyPage3::OnTimer(UINT nIDEvent)
{
  static int nCheck = 1;
  CProgressCtrl *pProgress =
         (CProgressCtrl *)GetDlgItem(IDC_PROGRESS);
  if (nIDEvent == ID_PULSE)
  {
    if (nCheck = !nCheck)
    {
      pProgress->SetRange(0, 1000);
      pProgress->SetPos(0);
      pProgress->SetStep(10);
    }
  } else if (nIDEvent == ID_INPROG && nCheck)
      pProgress->StepIt();
  CPropertyPage::OnTimer(nIDEvent);
}
```

Результат работы программы показан на рис. 16.5.

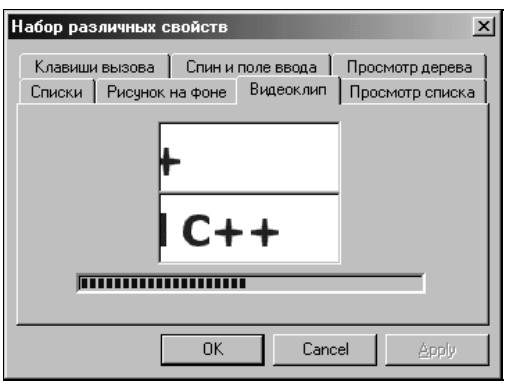



**Рис. 16.5.** Страница свойств с просмотром видеоклипов и индикатором

# Счетчик

Элемент управления "счетчик" (SPIN BUTTON или UP-DOWN) представляет собой пару кнопок со стрелками, назначение которых — смена значения, например, уменьшение либо увеличение числовой величины. 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 счетчика. 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 вместе 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.

**Примечание**

Да, да, именно так. О причинах такого решения можно только догадываться.

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 три из которых — уведомляющие. Все они посылаются родительскому окну и никак не воздействуют на ассоциированное со счетчиком окно. Эти 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 — 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.

```
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 отклика-

ются на воздействие со стороны счетчика, а другие, такие как индикатор (PROGRESS BAR), никак не реагируют.

## Параметры элемента управления

В случае, если счетчик не имеет стиля UDS_AUTOBUDDY, класс предоставляет возможность связывания с ним окна (функция *SetBuddy* возвращает указатель на 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 не может быть сохранен для дальнейшего использования, если указывает на временный объект.

Для установки значения счетчика и получения текущего значения вызываются функции *SetPos* и *GetPos*, соответственно. Тип текущего значения — целое число, следовательно, для изменения содержимого ассоциированного окна, которое не является числом, требуется некоторое преобразование. 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 обоих параметров должны лежать в пределах от UD_MINVAL до UD_MAXVAL. Кроме того, разница между максимальным и минимальным значениями границ не должна быть больше UD_MAXVAL. В настоящий момент действуют 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.

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: одно нажатие кнопки изменяет значение счетчика 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*:

```
BOOL CSpinButtonCtrl::SetAccel (
    int nAccel,
    UDACCEL* pAccel)
```

Позволяет задать массив, включающий в качестве элементов структуры UDACCEL, которые содержат необходимую информацию для динамического изменения значения приращения:

```
typedef struct {
  int nSec;
  int nInc;
} UDACCEL;
```

Поле *nSec* содержит время в секундах, по истечении которого приращение устанавливается равным значению поля *nInc*.

Массив должен быть упорядочен по возрастанию поля *nSec*, которое содержит время от начала взаимодействия пользователя со счетчиком.

И, наконец, последняя группа функций позволяет 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 служит для назначения системы счисления, а вторая — для получения информации 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*.

В рассматриваемом нами приложении *Pro* мы воспользовались только возможностью установки диапазона:

```
BOOL CMyPropertyPage6::OnSetActive()
{
```

```
  // Элемент управления создан в редакторе ресурсов, и поэтому
  // для установки диапазона счетчика
  // нам достаточно просто получить указатель на имеющийся объект
  ((CSpinButtonCtrl *)GetDlgItem(IDC_SPIN1))->SetRange(0, 20);
  return CPropertyPage::OnSetActive();
}
```

Соответствующая страница представлена на рис. 16.6. А на рис. 16.7 показаны стили, которые необходимо установить для этого элемента управления, чтобы "подключить" его к предварительно созданному полю ввода.

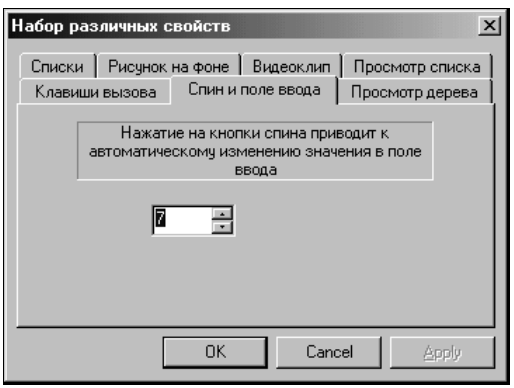



**Рис. 16.6.** Страница свойств со счетчиком и ассоциированным с ним полем ввода

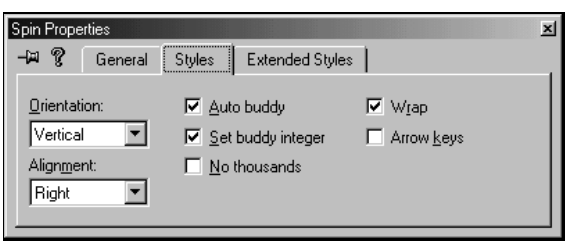



**Рис. 16.7.** Окно свойств элемента управления "счетчик"

# 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 пиктограмм различного размера, отображение которых несет дополнительную визуальную информацию.

## Режимы вывода

Элемент "просмотр списка" 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 |
|---|---|
| Режим больших пиктограмм (Icon view) | Задается стилем LVS_ICON; каждая запись представляет собой пиктограмму, под которой расположена надпись; записи могут перемещаться пользователем внутри списка просмотра |
| Режим малых пиктограмм (Small icon) | Задается стилем LVS_SMALLICON; каждая запись выводится в виде маленькой пиктограммы с надписью, расположенной справа от нее; записи могут перемещаться пользователем внутри списка |
| Режим списка (List view) | Задается стилем LVS_LIST; каждая запись выводится в виде маленькой пиктограммы с надписью, расположенной справа от нее; в отличие от предыдущего способа вывода, записи выводятся столбцами, и пользователь не может менять их расположение внутри списка |
| Режим таблицы (Report view) | Задается стилем LVS_REPORT; каждая запись с информацией о ней выводится в отдельной строке; пиктограмма 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 пиктограмм — 32×32 пиксела, а малых — 16×16.

Примером применения элемента "просмотр списка" может служить правая часть окна программы Windows Explorer, которая позволяет установить перечисленные выше способы вывода записей в окне.

Для задания выравнивания пиктограмм и надписей при первых двух способах вывода служат 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 реализовать другие возможности, например, автоматически выбирать элемент, когда курсор мыши "задержи-

вается" над ним определенное время, редактировать надписи под пиктограммами непосредственно в просмотре списка, выбирать больше одной записи, в режиме таблицы изменять порядок расположения столбцов и т. д.

## Создание элемента "просмотр списка"

Библиотека MFC имеет в своем составе класс, инкапсулирующий возможности рассматриваемого элемента управления. Это класс *CListCtrl*, позволяющий существенно упростить использование самого элемента управления. Как и большинство других элементов управления, "просмотр списка" создается или в блоке диалога путем добавления в шаблон блока диалога соответствующего элемента, или в обычном окне при помощи вызова функции *Create* в функциях окна *OnInitDialog* (для диалогов) или *OnCreate*.

Для того чтобы добавить записи в список, нужно воспользоваться одной из версий функции *InsertItem*. Первая версия использует заранее подготовленную структуру *LV_ITEM*, а последняя в качестве параметров принимает значения, аналогичные значениям полей структуры *LV_ITEM*. Вторая и третья версии функции полезны для списков, элементы которых не требуют для отображения дополнительных данных.

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 быть комбинацией 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 |

Поле *iItem* задает индекс записи, перед которой осуществляется вставка новой записи; поле *iSubItem* содержит порядковый номер (начиная с 1) дополнительного поля (в функции *InsertItem* не используется); поля *state* и *stateMask* задают текущее и возможные состояния для данной записи. Эти поля 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 |

Поле *pszText* является указателем на текстовую строку (буфер) (если параметр имеет значение LPSTR_TEXTCALLBACK, то это строка обратного вызова); поле *cchTextMax* задает размер буфера, на который указывает поле *pszText*, используется только для получения параметров записи в функции *GetItem* (а в функции *InsertItem* значение этого параметра игнорируется); поле *iImage* задает индекс пиктограммы в списке изображений пиктограмм (если параметр имеет значение I_IMAGECALLBACK, то это запись обратного вызова); поле *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 элемента "просмотр списка" в виде таблицы, следует в обязательном порядке, независимо от количества полей представления записей списка, создать столбцы. Для определения требуемой ширины столбца служит функция:

```
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 следует использовать функцию *InsertColumn*:

```
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. Рассмотрим ту реализацию, которая в качестве параметров, в данном случае для нового столбца, использует структуру *LV_COLUMN*:

```
typedef struct _LV_COLUMN {
  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 быть равно нулю или содержать 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 |

Поле *cx* определяет ширину столбца, при значении этого поля, равным –1, ширина не задается, т. е. содержимое столбца не будет отображаться; поле *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 образов |
| LVCFMT_COL_HAS_IMAGES | Элемент заголовка имеет рисунок в списке образов |

Поле *pszText* содержит 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*); поле *iSubItem* связывает столбец с полем записи. Поле *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 обеспечивается функцией

```
CHeaderCtrl* CListCtrl::GetHeaderCtrl().
```

## Параметры просмотра списка

Как уже упоминалось, стиль просмотра списка можно менять в любое время уже после его создания. Соответствующие функции для этого библиотекой MFC не предусмотрены, поэтому следует использовать функции API —

*::GetWindowLong* и *::SetWindowLong*. Следующий пример показывает, как это сделать.

```
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* реализованы 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.

## Работа со списком в целом

Мы рассмотрим только несколько функций данной группы.

```
BOOL CListCtrl::RedrawItems (
    int nFirst,
    int nLast)
```

Заставляет элемент "просмотр списка" перерисовать записи с индексами, лежащими в диапазоне от *nFirst* и до *nLast* включительно. Однако указанная группа записей не перерисовывается немедленно — это происходит по получении списком сообщения WM_PAINT. Для немедленной перерисовки конкретной записи предназначена функция *Update*.

```
BOOL CListCtrl::Update (int nItem)
```

Перерисовывает заданную запись списка. При этом возможно переупорядочивание записей (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 | Пиктограммы 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 |

В режиме "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.

Для создания (а также переопределения) рабочей области необходимо вызвать функцию

```
void CListCtrl::SetWorkAreas(
    int nWorkAreas,
    LPRECT prc)
```

Параметр *nWorkAreas* определяет номер создаваемой рабочей области, а *prc* указывает на адрес памяти, где записаны ее координаты (в единицах клиентской области).

**Примечание**

Создавать рабочие области можно только для режимов больших и малых пиктограмм.

Получить координаты конкретной рабочей области можно с помощью функции

```
void CListCtrl::GetWorkAreas(
    int nWorkAreas,
    LPRECT prc)
```

Параметры такие же, как у функции *SetWorkAreas*.

Общее число созданных в элементе "просмотр 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

Пиктограммы для записей элемента "просмотр списка" содержатся в списках изображений, которые необходимо создать и определить, если требуется сопровождать записи пиктограммами. Один список изображений содержит полноразмерную пиктограмму (для режима больших пиктограмм), второй — пиктограммы 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, а вторая устанавливает текущий новый список изображений. Параметр *nImageList* в обеих функциях может принимать одно из следующих значений:

| | |
|---|---|
| LVSIL_NORMAL | Список изображений с большими пиктограммами |
| LVSIL_SMALL | Список изображений с малыми пиктограммами |
| LVSIL_STATE | Список с изображениями для отображения состояний |

Наличие пиктограмм только одного размера (только малых или только больших) позволяет при помощи масштабирования использовать имеющиеся пиктограммы для всех режимов вывода. В любом случае для поддержки всех режимов отображения требуется создание обоих списков 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 необходимой пиктограммы в обоих списках.

Если список состояний изображений задан, то элемент "просмотр списка" резервирует место слева от каждой пиктограммы в записи для изображения состояния. Можно использовать эти изображения, например, выбрана/не выбрана, 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 элемент "просмотр списка" использует режим вывода "большие пиктограммы", изображения состояний обычно не используются.

По умолчанию, когда уничтожается элемент "просмотр списка", уничтожаются и его списки изображения. Однако если элемент имеет стиль LVS_SHAREIMAGELISTS, то приложение само отвечает за уничтожение списка изображений в случае, когда он больше не нужен. Этот стиль, например, следует указывать, 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 (видимое — *fCheck* = TRUE, или невидимое — *fCheck* = FALSE) изображения элемента.

## Виртуальные списки

Виртуальный список — это список, число записей которого неограниченно. Список такого типа удобно использовать при работе с базами данных. Для его создания необходимо вызвать одну из функций *CreateWindow* или *CreateWindowEx*, передав в качестве части параметра *dwStyle* значение LVS_OWNERDATA, которое может использоваться 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 списке — параметр *iCount*. Параметр *dwFlags* позволяет определить поведение виртуального списка после установки нового размера. Возможны комбинации следующих значений:

| | |
|---|---|
| LVSICF_NOINVALIDATEALL | Элемент управления не перерисовывается до того, как не обработается какой-либо его компонент. 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, пиктограммы, 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. выше раздел

"Режимы вывода") может быть выведена в столбце справа от пиктограммы и основного поля. Все записи в одном просмотре списка могут иметь одинаковое количество полей. Используя сообщения элемента "просмотр 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 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 пиктограммы из списка изображений, состояние и данные записи. Под данными записи здесь понимается определяемое приложением значение, связанное с конкретной записью в списке.

## Записи по запросу (Callback Items)

*Записи по запросу* (иначе, записи обратного вызова) — это записи, для которых приложение (а не элемент управления) сохраняет текст или пиктограммы, или то и другое. Несмотря на то, что элемент "просмотр списка" может сохранять их, возможно создание записей 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 обработку 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 операции вырезания и вставки (cut-and-paste) |
| LVIS_DROPHILITED | Требует обработки назначения целевой записи списка для выполнения операции переноса (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 пиктограммы из списка изображений для каждого элемента списка |
| LVIS_STATEIMAGEMASK | Приложение берет на себя выбор индекса пиктограммы из списка изображений состояний для каждого элемента списка |

## Изменение содержимого 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 списке.

```
void CListCtrl::SetItemCount (int nItems)
```

Подготавливает список к добавлению большого количества новых записей. При этом перед добавлением 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 списке (для списков множественного выбора).

```
int CListCtrl::SetSelectionMark(int iIndex)
```

и

```
int CListCtrl::GetSelectionMark()
```

Устанавливает / возвращает индекс первого выделенного элемента при множественном выборе.

После вставки новой записи ее атрибуты можно запросить или изменить. Для этого применяются функции, использующие структуру *LV_ITEM*:

```
BOOL CListCtrl::GetItem (LV_ITEM* pItem)
```

Возвращает атрибуты записи в списке. Для получения текстовых строк следует предусмотреть буфер и указать его размер. В случае, если запись содержит строку большего размера, чем размер буфера, она будет усечена.

```
BOOL CListCtrl::SetItem (const LV_ITEM* pItem)
```

и

```
BOOL CListCtrl::SetItem (
    int nItem,
```

```
    int nSubItem,
    UINT nMask,
    LPCTSTR lpszItem,
    int nImage,
    UINT nState,
    UINT nStateMask,
    LPARAM lParam)
```

Устанавливают все или только некоторые атрибуты записи в просмотре списка.

Следующие функции позволяют работать с частью атрибутов записи списка без использования структуры *LV_ITEM* и громоздких функций, описанных ранее.

Для получения и назначения 32-битного значения, ассоциированного с записью, предназначены следующие функции:

```
DWORD CListCtrl::GetItemData (int nItem)
```

Возвращает значение, связанное с записью.

```
BOOL CListCtrl::SetItemData (
    int nItem,
    DWORD dwData)
```

Связывает новое значение с записью.

Для получения и программного изменения состояния записи предназначены следующие функции:

```
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 нового текста используются функции *GetItemText* и *SetItemText*.

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 больших пиктограмм 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.

Структура *LV_FINDINFO* имеет следующий вид:

```
typedef struct tag LV_FINDINFO {
  UINT flags;
  LPCSTR psz;
  LPARAM lParam;
  POINT pt;
  UINT vkDirection;
} LV_FINDINFO;
```

Поле *flags* задает тип поиска. Оно может содержать комбинацию следующих флагов:

| | |
|---|---|
| LVFI_PARAM | При поиске используется значение, содержащееся в поле *lParam*; если задано данное значение, то все другие значения игнорируются |
| LVFI_PARTIAL | Поиск ведется до совпадения строки, заданной в поле *psz*, и текстового значения записи, т. е. его основного поля; длина текста записи может быть больше, чем строка, заданная в поле *psz*; этот параметр предполагает использование также флага LVFI_STRING |
| LVFI_STRING | Поиск ведется до совпадения строки, заданной в поле *psz*, и текстового значения записи, т. е. его основного поля; если дополнительно не задан флаг LVFI_PARTIAL, то поиск идет до полного совпадения проверяемой строки и строки, указываемой параметром *psz* |

| | |
|---|---|
| LVFI_WRAP | Продолжение поиска с начала списка, если строка не найдена по достижению его конца |
| LVFI_NEARESTXY | Задает поиск ближайшей в пространстве поля списка записи от позиции, заданной индексом записи |

Поле *psz* является указателем на текстовую строку, которая должна быть найдена, если в параметре *flags* заданы значения LVFI_STRING или LVFI_PARTIAL. Поле *lParam* задает 32-битное значение, которое сравнивается со значениями, ассоциированными с элементами списка. Начальная точка поиска задается параметром *pt*, а направление — параметром *vkDirection*. Последние два параметра используются только с флагом LVFI_NEARESTXY.

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 есть необходимость узнать, например, в обработчике нажатия кнопки мыши, на какой записи находится ее курсор, следует воспользоваться одной из функций *HitTest* или *SubItemHitTest*. При этом надо иметь в виду, что из-за зависимости "системы координат" списка от режима отображения в случае, когда все элементы просмотра списка не могут быть одновременно отображены на экране, для правильной работы этих функций следует осуществлять преобразование координат.

```
int CListCtrl::HitTest (LV_HITTESTINFO* pHitTestInfo)
```

или

```
int CListCtrl::HitTest (
    CPoint pt,
    UINT* pFlags = NULL)
```

и

```
int CListCtrl::SubItemHitTest(LPLVHITTESTINFO pInfo)
```

Возвращают индекс записи (подзаписи) в просмотре списка, расположенной в заданных координатах, если такой записи нет, функция возвращает –1. Обе версии функции *HitTest* отличаются только способом передачи данных для поиска. Структура *LV_HITTESTINFO*, указатель на которую передается в параметре *pHitTestInfo*, содержит поля, аналогичные параметрам второй версии функции *HitTest*. Описание структуры следующее:

```
typedef struct _LV_HITTESTINFO {
  POINT pt;
  UINT flags;
  int iItem;
  int iSubItem;
} LV_HITTESTINFO;
```

Поле *pt* задает координаты точки, которая предположительно находится в рамках занимаемой некоторым элементом области. Поле *flags* используется для получения дополнительной информации и может быть заполнено комбинацией из следующих флагов:

| | |
|---|---|
| LVHT_ABOVE | Проверяемая точка расположена выше клиентской области элемента управления |
| LVHT_BELOW | Проверяемая точка расположена ниже клиентской области элемента управления |
| LVHT_NOWHERE | Проверяемая точка расположена в клиентской области просмотра списка, но лежит не в области собственно списка |
| LVHT_ONITEMICON | Проверяемая точка расположена в границах пиктограммы записи |
| LVHT_ONITEMLABEL | Проверяемая точка расположена в границах надписи записи |
| LVHT_ONITEMSTATEICON | Проверяемая точка расположена в границах пиктограммы состояния записи |
| LVHT_TOLEFT | Проверяемая точка расположена левее клиентской области элемента управления |
| LVHT_TORIGHT | Проверяемая точка расположена правее клиентской области элемента управления |

Флаги LVHT_ABOVE, LVHT_BELOW, LVHT_TOLEFT и LVHT_TORIGHT можно использовать для определения необходимости прокрутки списка в соответствующем направлении. Значение LVHT_ONITEM можно использовать для определения того, находится ли вообще точка над записью, это значение является комбинацией значений LVHT_ONITEMICON, LVHT_ONITEMLABEL и LVHT_ONITEMSTATEICON. Поля *iItem* и *iSubItem* определяют индекс записи и подзаписи в просмотре списка.

Следующая функция осуществляет сортировку записей в просмотре списка:

```
BOOL CListCtrl::SortItems (
    PFNLVCOMPARE pfnCompare,
    DWORD dwData)
```

Использует определяемую приложением функцию для сравнения записей. 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 параметра *dwData* функции *CListCtrl::SortItems*. Функция возвращает отрица-

тельное 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 сопровождающем раздел примере. Обработчик уведомления получает указатель на новую строку, которую ввел пользователь. Для изменения 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 надо.

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 пиктограммы, редактирует надписи и т. п., элемент "просмотр списка" посылает родительскому окну уведомления, которые можно обрабатывать в функции родительского окна *OnNotify*.

Элемент "просмотр списка" посылает следующие уведомления:

| | |
|---|---|
| LVN_BEGINDRAG | Сигнализирует о начале операции перемещения при помощи левой кнопки мыши (drag-and-drop) |
| LVN_BEGINLABELEDIT | Сигнализирует о начале операции редактирования надписи записи |
| LVN_BEGINRDRAG | Сигнализирует о начале операции перемещения 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 операции 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 в просмотре списка |
| LVN_ITEMCHANGING | Запрашивает подтверждение изменения записи в просмотре 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 так же, как и для других элементов управления, за исключением того, что приложение получает информацию в структуре *LV_DISPINFO*, которая, кроме обычного для уведомлений заголовка, содержит структуру *LV_ITEM* 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;
```

## Реализация просмотра списка с возможностью перемещения записей

Перемещение записей можно реализовать только для режимов больших и малых пиктограмм, что вполне естественно. Главное, что нужно при этом помнить, это то, что индексы записей после изменения положения записей

не будут отражать их фактический порядок на экране — порядок размещения записей в просмотре списка автоматически не меняется.

Для поддержки перемещения записей в просмотре списка требуется обрабатывать следующие уведомления: LVN_BEGINDRAG, LVN_BEGINRDRAG, а также сообщения от мыши: отпускание левой и правой кнопок и 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 индекс записи, а параметр *lpPoint* — текущие координаты ее левого верхнего угла.

Порядок расположения столбцов можно узнать и установить с помощью функций

```
BOOL CListCtrl::GetColumnOrderArray(
    LPINT piArray,
    int iCount = -1)
```

и

```
BOOL CListCtrl::SetColumnOrderArray(
    int iCount,
    LPINT piArray)
```

Параметр *piArray* получает указатель на массив значений индексов столбцов (которые расположены в новом порядке — для функции *SetColumnOrderArray*). Число столбцов задается параметром *iCount*. Если в функции *GetColumnOrderArray* этот параметр равен –1, то число столбцов определяет сама библиотека.

## 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 в просмотре списка.

## Пример реализации просмотра списка

В этом разделе приведен пример реализации просмотра списка, в качестве записей которого используется информация о файлах текущего (для приложения) каталога, а именно — имя, тип (определяемый по расширению), длина в килобайтах, а также дата и время последней операции записи.

```
BOOL CMyPropertyPage4::OnSetActive()
{
  static BOOL bFlag = TRUE;
  if(bFlag)
  {
    // 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 = (CListCtrl *)GetDlgItem(IDC_LIST_CTRL);
    m_List->SetImageList(&m_ImageNormal, LVSIL_NORMAL);
    m_List->SetImageList(&m_ImageSmall, LVSIL_SMALL);
    CRect rect;
    m_List->GetClientRect(rect);
    // Добавление в список просмотра
    // 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
      {
        // Директории игнорируем
        if (FindFileData.dwFileAttributes &
            FILE_ATTRIBUTE_DIRECTORY)
          continue;
        item.mask = LVIF_TEXT | LVIF_IMAGE | LVIF_PARAM;
```

```
        // Для реализации сортировки сохраняем дополнительно
        // 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);
        // Длина 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));
    }
    bFlag = FALSE;
  }
  OnLargeicons();
  return CPropertyPage::OnSetActive();
}
// Обработчики команд
void CMyPropertyPage4::OnSmallIcon()
{
  SetStyle(LVS_SMALLICON);
}
void CMyPropertyPage4::OnLargeicons()
{
  SetStyle(LVS_ICON);
}
```

```
void CMyPropertyPage4::OnDetails()
{
  SetStyle(LVS_ICON | LVS_REPORT);
}
void CMyPropertyPage4::OnLists()
{
  SetStyle(LVS_LIST);
}
// Функция изменения стиля для просмотра списка
// при смене режимов отображения
void CMyPropertyPage4::SetStyle(DWORD dwViewStyle)
{
  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);
}
// т. к. было динамическое выделение памяти
// для хранения информации о файлах,
// при уничтожении списка просмотра
// необходимо освободить эту память
BOOL CMyPropertyPage4::DestroyWindow()
{
  for (int i = 0; i < m_List->GetItemCount(); i++)
    delete (WIN32_FIND_DATA *)m_List->GetItemData(i);
  return CPropertyPage::DestroyWindow();
}
```

Внешний вид страницы представлен на рис. 16.8.

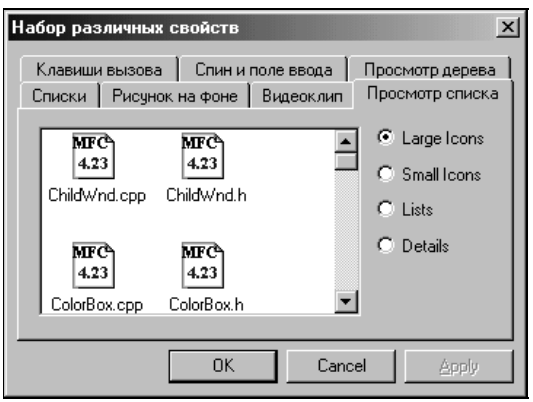



**Рис. 16.8.** Окно свойств с просмотром списка в режиме больших пиктограмм

# Просмотр дерева

Кроме возможности выводить записи 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 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 поэтому естественно, что данный элемент управления может применяться только для информации, имеющей иерархическую структуру. Для создания этого элемента в библиотеке MFC предназначен класс *CTreeCtrl*. Каждая запись в просмотре дерева состоит из текстовой строки (надписи) и заданного битового изображения и может иметь одну или более подзаписей, связанных с ней. Щелкнув мышью по записи, пользователь может развернуть или свернуть список связанных с записью подзаписей. 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 реализации просмотра дерева в блоке диалога, особенно при наличии в блоке диалога других элементов управления. Напротив, если просмотр дерева должен быть реализован 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 элемента управления "просмотр 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*.

## Списки изображений

Каждая запись в просмотре дерева может иметь две ассоциированных с ней пиктограммы. Первая пиктограмма выводится на экран, когда запись развернута, вторая — когда запись свернута (например, пиктограммы, изображающие открытый и закрытый каталог).

Кроме того, приложение может задать список изображений для обозначения состояний записи. В этом случае элемент "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*.

## Функции для работы с просмотром дерева в целом

Когда ветка дочерних записей развернута, то они изображаются с некоторым сдвигом относительно родительской записи. Для определения и изменения сдвига дочерней записи относительно родительской предназначены функции:

`UINT CTreeCtrl::`**`GetIndent`** `()`

Возвращает сдвиг (в пикселах) дочерней записи относительно родительской.

`void CTreeCtrl::`**`SetIndent`** `(UINT nIndent)`

Задает сдвиг (в пикселах) дочерней записи относительно родительской. Если задается значение меньшее, чем определяемое системой, то функция игнорируется.

## Изменения 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 в просмотре дерева.

Для того чтобы удалить все записи из просмотра дерева, можно воспользоваться функцией *DeleteAllItems*.

Начальное положение записи при ее добавлении в просмотр дерева определяется с помощью функции *InsertItem*. При вызове этой функции задаются дескриптор родительской записи и дескриптор записи, после которой должна быть вставлена новая запись. Второй из указанных дескрипторов 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.

Функция, осуществляющая вставку новых записей, реализована в нескольких версиях:

```
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 этой 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. Их целесообразно использовать в относительно простых реализациях, когда от просмотра дерева не требуется поддержка всех функциональных возможностей. Кроме того, существуют функции, позволяющие впоследствии дополнить информацию о записи.

Чтобы не дублировать описание, приведем более подробно только описание структур *TV_INSERTSTRUCT* и *TV_ITEM*, которые вместе содержат полный набор параметров, описывающих добавляемую запись и необходимых для реализации добавления.

Структура *TV_INSERTSTRUCT* имеет следующий вид:

```
typedef struct _TV_INSERTSTRUCT {
  HTREEITEM hParent;
  HTREEITEM hInsertAfter;
#if (_WIN32_IE >= 0x0400)
  union {
    TVITEMEX itemex;
    TVITEM item;
  } DUMMYUNIONNAME;
#else
  TVITEM item;
#endif
} TV_INSERTSTRUCT;
```

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. Оно 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.

Структура *TV_ITEM* имеет следующий вид:

```
typedef struct _TV_ITEM {
  UINT mask;
  HTREEITEM hItem;
  UINT state;
  UINT stateMask;
```

```
  LPSTR pszText;
  int cchTextMax;
  int iImage;
  int iSelectedImage;
  int cChildren;
  LPARAM lParam;
} TV_ITEM;
```

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 | Оба поля *state* и *stateMask* содержат данные |
| TVIF_TEXT | Поле *pszText* содержит данные |

Поле *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.

Поля *state* и *stateMask* задают текущее и возможные состояния для данной записи. Эти поля могут содержать комбинации следующих флагов (за исключением двух последних):

| | |
|---|---|
| TVIS_BOLD | Запись отображается полужирным шрифтом |
| TVIS_CUT | Запись помечена для операции вырезания или вставки (cut-and-paste) |
| TVIS_DISABLED | Запись недоступна и для ее перерисовки применяется стандартный стиль и цвет |
| TVIS_DROPHILITED | Запись выделена для переноса (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 |

Поле *pszText* является указателем на строку, содержащую надпись записи, если заданы соответствующие атрибуты в структуре. Если параметр имеет значение LPSTR_TEXTCALLBACK, то это строка обратного вызова (подробнее об обратном вызове см. "Записи по запросу" в разделе, посвященном просмотрам списка), 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 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 пиктограмм 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 *TV_ITEMEX* имеет одно дополнительное поле в дополнение к полям структуры *TV_ITEM* — int iIntegral , в котором 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 используется функция:

```
BOOL CTreeCtrl::DeleteItem (HTREEITEM hItem)
```

Обратите внимание, 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 следующие функции:

```
BOOL CTreeCtrl::SetItem (TV_ITEM* pItem)
```

и

```
BOOL CTreeCtrl::SetItem (
    HTREEITEM hItem,
    UINT nMask,
    LPCTSTR lpszItem,
    int nImage,
    int nSelectedImage,
    UINT nState,
    UINT nStateMask,
    LPARAM lParam)
```

Устанавливают параметры записи в просмотре дерева. Назначение и тип параметров аналогичны рассмотренным ранее.

```
BOOL CTreeCtrl::GetItem (TV_ITEM* pItem)
```

Возвращает параметры записи в структуре, на которую указывает параметр *pItem*.

Каждая запись в просмотре дерева имеет текущее состояние, например, она может быть выбрана, недоступна, развернута и т. д. Во многих случаях элемент "просмотр дерева" автоматически устанавливает состояния записей, отражая действия, производимые пользователем. Для того чтобы узнать или изменить только состояние записи, можно воспользоваться следующими функциями:

```
UINT CTreeCtrl::GetItemState (
    HTREEITEM hItem,
    UINT nStateMask)
```

Возвращает состояние записи.

```
BOOL CTreeCtrl::SetItemState (
    HTREEITEM hItem,
    UINT nState,
    UINT nStateMask)
```

Устанавливает состояние записи.

Для того чтобы узнать или изменить только изображения, связанные с записью, можно воспользоваться следующими функциями:

```
BOOL CTreeCtrl::GetItemImage (
    HTREEITEM hItem,
    int& nImage,
    int& nSelectedImage)
```

Записывает номера изображений в параметры *nImage* и *nSelectedImage*.

```
BOOL CTreeCtrl::SetItemImage (
    HTREEITEM hItem,
    int nImage,
    int nSelectedImage)
```

Устанавливает изображения для заданной записи.

Для того чтобы узнать или изменить только надпись записи, можно воспользоваться следующими функциями:

```
CString CTreeCtrl::GetItemText (HTREEITEM hItem)
```

Возвращает объект типа *CString*, содержащий надпись записи.

```
BOOL CTreeCtrl::SetItemText (
    HTREEITEM hItem,
    LPCTSTR lpszItem)
```

Заменяет текущее название записи новым.

Следующая пара функций предназначена для получения и изменения 32-битного значения, ассоциированного с записью в просмотре дерева:

```
DWORD CTreeCtrl::GetItemData (HTREEITEM hItem)
```

Позволяет получить 32-битное значение.

```
BOOL CTreeCtrl::SetItemData (
    HTREEITEM hItem,
    DWORD dwData)
```

Устанавливает 32-битное значение, заданное параметром *dwData*.

В случае, когда необходимо получить размеры прямоугольника, ограничивающего запись в просмотре дерева, и узнать при этом, видима запись или нет, используется функция *GetItemRect*:

```
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 прямоугольник содержит только размеры надписи для данной записи. Иначе он определяет размеры прямоугольника, который содержит запись целиком.

Следующая описываемая функция возвращает значение TRUE, если указанная запись имеет дочерние записи:

```
BOOL CTreeCtrl::ItemHasChildren (HTREEITEM hItem)
```

Функцией *GetChildItem* стоит воспользоваться, если нужно получить дескриптор первой дочерней записи:

```
HTREEITEM CTreeCtrl::GetChildItem (HTREEITEM hItem)
```

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 нажатием на специальную кнопку, расположенную 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 для родительской записи, заданной 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 этого параметра:

| | |
|---|---|
| TVGN_CARET | Установить выбор заданной записи |
| TVGN_DROPHILITE | Перерисовать запись как запись назначения операции перетаскивания (drag-and-drop) |
| TVGN_FIRSTVISIBLE | Прокрутить записи так, чтобы заданная запись стала видимой |

Выполняемые данной функцией операции могут также выполняться следующей парой 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 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 для записи, заданной параметром *hItem*:

```
HTREEITEM CTreeCtrl::GetParentItem (HTREEITEM hItem)
```

Класс *CTreeCtrl* позволяет определить запись по координатам и, если необходимо, узнать, на какой записи находится курсор мыши. Для этого следует воспользоваться функцией *HitTest*, например, в обработчике нажатия кнопки мыши:

```
HTREEITEM CTreeCtrl::HitTest (
    CPoint pt,
    UINT* pFlags)
```

или

```
HTREEITEM CTreeCtrl::HitTest (TV_HITTESTINFO* pHitTestInfo)
```

Возвращает дескриптор записи, которой принадлежит указанная точка, или NULL, если точка не принадлежит ни одной записи. Параметр *pFlags* первой версии функции является указателем на переменную целого типа, принимающую результаты проверки. Может иметь одно или больше значений, указанных в описании параметра *flags* структуры *TV_HITTESTINFO*. Параметр *pHitTestInfo* является указателем на структуру типа *TV_HITTESTINFO*, содержащую координаты точки для проверки и принимающую результаты проверки.

Приведем описание структуры *TV_HITTESTINFO*:

```
typedef struct _TV_HITTESTINFO {
  POINT pt;
  UINT flags;
  HTREEITEM hItem;
} TV_HITTESTINFO;
```

Поле *flags* используется для хранения результатов проверки и может содержать комбинацию следующих значений:

| | |
|---|---|
| TVHT_ABOVE | Проверяемая точка расположена выше клиентской области элемента управления |
| TVHT_BELOW | Проверяемая точка расположена ниже клиентской области элемента управления |
| TVHT_NOWHERE | Проверяемая точка расположена в клиентской области просмотра дерева, но лежит вне области собственно дерева |
| TVHT_ONITEM | Проверяемая точка расположена в границах записи |
| TVHT_ONITEMBUTTON | Проверяемая точка расположена в границах кнопки записи |
| TVHT_ONITEMICON | Проверяемая точка расположена в границах пиктограммы записи |
| TVHT_ONITEMINDENT | Проверяемая точка расположена в границах отступа записи |
| TVHT_ONITEMLABEL | Проверяемая точка расположена на метке, ассоциированной с записью |

| | |
|---|---|
| TVHT_ONITEMRIGHT | Проверяемая точка расположена правее надписи записи |
| TVHT_ONITEMSTATEICON | Проверяемая точка расположена в границах пиктограммы состояния записи |
| TVHT_TOLEFT | Проверяемая точка расположена левее клиентской области элемента управления |
| TVHT_TORIGHT | Проверяемая точка расположена правее клиентской области элемента управления |

К этой же группе я отнес функции, осуществляющие сортировку записей в просмотре дерева. Их две. Первая предназначена для сортировки по алфавиту дочерних записей родительской записи. Это функция *SortChildren*. 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 это позволяет 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 _TV_SORTCB {
  HTREEITEM hParent;
  PFNTVCOMPARE lpfnCompare;
  LPARAM lParam;
} TV_SORTCB;
```

Поле *lpfnCompare* является указателем на определяемую приложением функцию сортировки записей. Она вызывается всякий раз, когда необходимо сравнить две записи и имеет описание:

```
int CALLBACK CompareFunc(
    LPARAM lParam1,
    LPARAM lParam2,
    LPARAM lParamSort)
```

Возвращает 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 для просмотра списка, которая была описана в соответствующем разделе.

## Обработка уведомлений

При работе с просмотром дерева пользователь посылает различные уведомления. Если необходимо, то для их обработки можно назначить функции-обработчики. Перечень уведомляющих сообщений, посылаемых элементом "просмотр дерева", приведен ниже:

| | |
|---|---|
| TVN_BEGINDRAG | Сигнализирует о начале операции перемещения при помощи левой кнопки мыши (drag-and-drop) |
| TVN_BEGINLABELEDIT | Сигнализирует о начале операции редактирования надписи записи |
| TVN_BEGINRDRAG | Сигнализирует о начале операции перемещения при помощи правой кнопки мыши (drag-and-drop) |
| TVN_DELETEITEM | Сигнализирует об удалении конкретной записи просмотра дерева |
| TVN_ENDLABELEDIT | Сигнализирует о завершении операции редактирования надписи записи |
| TVN_GETDISPINFO | Запрашивает информацию о записи по запросу элемента "просмотр дерева" для ее отображения |
| TVN_ITEMEXPANDED | Сигнализирует о смене состояния записи в просмотре дерева (сворачивании или разворачивании) |
| TVN_ITEMEXPANDING | Запрашивает подтверждение смены состояния записи в просмотре дерева (сворачивании или разворачивании) |
| TVN_KEYDOWN | Сигнализирует о нажатии клавиши на клавиатуре |
| TVN_SELCHANGED | Сигнализирует о смене выбора записи в просмотре дерева |
| TVN_SELCHANGING | Запрашивает подтверждение смены выбора записи в просмотре дерева |
| TVN_SETDISPINFO | Уведомляет родительское окно о необходимости изменения информации о записи по запросу, которая предоставляется приложением |

Элемент "просмотр дерева" посылает уведомления в форме сообщения WM_NOTIFY, например, так:

```
BOOL CMarsWnd::OnNotify(
  WPARAM wParam, LPARAM lParam, LRESULT* pResult)
{
  ...
  TVDISPINFO *ptvdi = (TV_DISPINFO *)lParam;
  TVITEM item = ptvdi->item;
  ...
}
```

Некоторые уведомления работают с другой структурой, а именно, со структурой *NMTREEVIEW*:

```
NMTREEVIEW *pnmtv = (NMTREEVIEW *)lParam;
```

Механизм обработки уведомлений в принципе такой же, как и рассмотренный в предыдущем разделе для просмотра списка.

Так же, как и класс *CListCtrl* для просмотра 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 при просмотре дерева, функции *GetBkColor*, *SetBkColor*, *GetTextColor*, *SetTextColor*, *GetInsertMarkColor* и *SetInsertMarkColor* для работы с цветами фона, текста и вставляемой записи, функции *GetToolTips* и *SetToolTips* для работы с всплывающими подсказками, функцию *EnsureVisible*, гарантирующую видимость конкретного элемента дерева и некоторые другие, в работе которых вы без труда сможете разобраться.

## Пример реализации просмотра дерева

В этом разделе приведен пример просмотра дерева, в качестве записей которого используется информация о файлах текущего (для приложения) каталога (рис. 16.9).

```
BOOL CMyPropertyPage7::OnSetActive()
{
  HTREEITEM hTreeCurrent;
  CTreeCtrl *pTree = (CTreeCtrl *)GetDlgItem(IDC_TREE1);
  // Поиск файлов и создание записей с
  // информацией о найденных файлах
  WIN32_FIND_DATA FindFileData;
  HANDLE hFile = FindFirstFile("*.*", &FindFileData);
  if (hFile != INVALID_HANDLE_VALUE)
  {
    TV_INSERTSTRUCT tis;
    TV_ITEM item;
    tis.hInsertAfter = TVI_LAST;
    item.mask = TVIF_TEXT;
    CString csText;
    do
    {
      // Отображаем только имена каталогов,
      // игнорируя их содержимое — кроме корневого
      if (FindFileData.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY)
      {
        item.pszText = FindFileData.cFileName;
```

```
        tis.hParent = TVI_ROOT;
        tis.item  = item;
        // Если каталог корневой — вставляем его
        if(!strcmp(item.pszText, ".."))
        {
          hTreeCurrent = pTree->InsertItem(&tis);
          continue;
        } else if(!strcmp(item.pszText, "."))
        {
          continue;
        } else
        {
          // Поскольку мы просматриваем файлы только текущего
          // каталога, то не меняем "текущего корня"
          pTree->InsertItem(&tis);
          continue;
        }
      }
      // Имя файла
      item.pszText = FindFileData.cFileName;
      tis.hParent = hTreeCurrent;
      tis.item  = item;
      pTree->InsertItem(&tis);
    }while (FindNextFile(hFile, &FindFileData));
  }
  return CPropertyPage::OnSetActive();
}
```

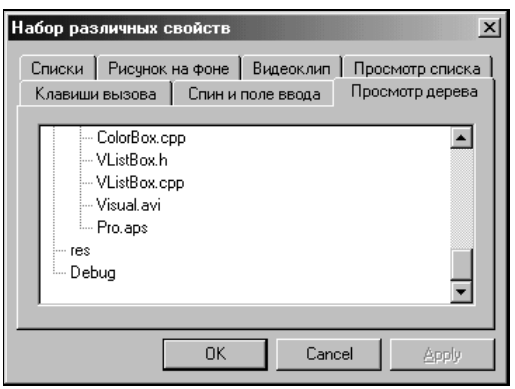



**Рис. 16.9.** Окно свойств с просмотром дерева

Конечно же мы рассмотрели далеко не все имеющиеся элементы управления. Но я убежден, что теперь вы в состоянии самостоятельно разобраться с тем, как создавать и работать с любыми элементами управления.



# Часть VI

## Архитектура "документ/представление"

**Глава 17.** Основы архитектуры "документ/представление"

**Глава 18.** Документ и его представления

**Глава 19.** Печать и предварительный просмотр документов

# ГЛАВА 17



# Основы архитектуры "документ/представление"

Нешто я да не пойму
При моем-то при уму?..
*Леонид Филатов*

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, представляющую собой 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 MFC) является объектом, в состав которого входят функции, определяющие его поведение, т. е. взаимодействие как с другими объектами, так и с данными. Любые действия пользователя во время выполнения приложения вызывают генерацию сообщений, содержащих 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 библиотеке MFC, а для других реализуются специальные обработчики. Повторю — все перечисленные свойства характерны (с той или иной степенью полноты) для каждого Windows-приложения.

Оставим в стороне приложения, базирующиеся на блоках диалога, и сосредоточимся на тех, 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 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 рассмотрены 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.

Рассматриваемая архитектура основывается на 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 части 4 данного руководства.

Архитектура "документ/представление" разделяет между собой 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. 17.1).

Когда запускается приложение под Windows, пользователь взаимодействует с документами посредством их изображений во фреймах. Фреймы документов 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 содер-

жимым. 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.

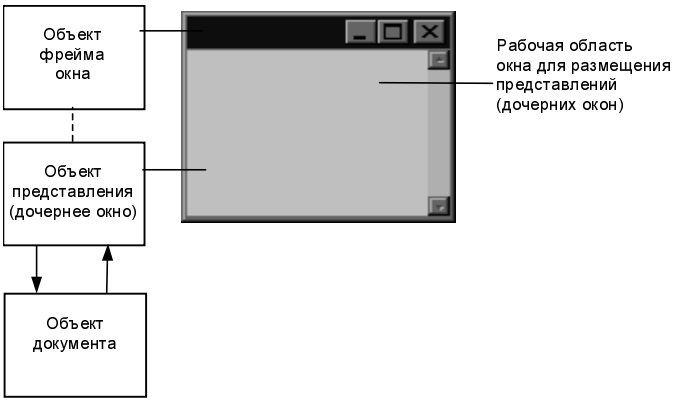



**Рис. 17.1.** Взаимосвязь между фреймом, документом и представлением

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 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 создания и взаимодействия фреймов, документов и представлений.

Приложение может поддерживать произвольное число типов документов. Единственное, что для этого нужно сделать — это 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.

# Создание различных типов документов

Из иерархии классов, представленной на рис. 17.2, наглядно видно, что базовым классом для создания шаблонов является *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* используется 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.

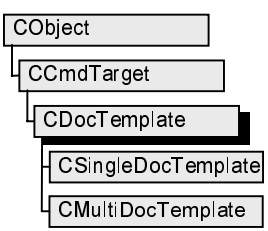



**Рис. 17.2.** Место классов для создания шаблонов документов в иерархии библиотеки MFC

Рассмотрим основные компоненты класса *CDocTemplate*. 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, командные клавиши 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, а полученный указатель передать в качестве параметра в функцию *CWinApp::AddDocTemplate*.

```
virtual void CDocTemplate::AddDocument (CDocument *pDoc),
virtual void CDocTemplate::RemoveDocument (CDocument *pDoc)
```

Добавляет (удаляет) документ, на который указывает *pDoc*, в 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 MFC 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 явный вызов *LoadTemplate*.

Ресурсами, используемыми документами этого типа, могут быть меню, командные клавиши и единственный строковый ресурс. Рассмотрим более подробно структуру строки описания типа.

```
// 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, строка состоит из семи подстрок (первая — пустая), разделенных символом "\n", которые содержат дополнительную информацию о типе документа:

- Заголовок главного окна SDI-приложения (Main frame caption). Не используется для MDI-приложений (обратите внимание, что в приведенном примере это поле отсутствует).

- 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 диалога Advanced Options (Дополнительные параметры) мастера создания каркаса приложения AppWizard.

Если какая-либо из этих подстрок не включается в строку, то вместо нее обязательно используется разделяющий символ "\n". Заключительный символ "\n" в строке является необязательным.

Доступ к этой информации осуществляется с помощью функции

```
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 зна- |

| | |
|---|---|
| | чение не определено, то по умолчанию используется *Untitled* (для русской версии Windows — *БезИмени*) |
| CDocTemplate::fileNewName | Имя типа 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. Данная строка изображается в списке типов файлов в блоке диалога File Open (Открытие файла); 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 данный тип документа недоступен при выполнении команды File\Open (Файл\Открыть) |
| CDocTemplate::regFileTypeId | Идентификатор типа документа, хранящийся в реестре Windows. Эта строка предназначена только для внутреннего использования; если не определена, то тип документа не может быть зарегистрирован в File Manager (Диспечер файлов) |
| CDocTemplate::regFileTypeName | Имя типа документа, хранящегося в регистрационной базе данных; эта строка может быть отображена в блоках диалога приложения, вызываемых для доступа к регистрационной базе данных |

Кроме того, в классе *CDocTemplate* объявлены еще четыре "чистые" 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.

**Примечание**

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 приведу 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*, 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.

Как я уже сказал, *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 MFC реализованы два другие класса — *CSingleDocTemplate* и *CMultiDocTemplate*, которые специально адаптированы для работы соответственно в SDI- и MDI-приложениях.

## Шаблоны однодокументных приложений

Для создания шаблона однодокументного интерфейса в библиотеке MFC реализован класс *CSingleDocTemplate*. Как вы помните, в SDI-приложениях главный фрейм является одновременно и фреймом документа, другими сло-

вами — в каждый 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;
...
};
```

Принцип действия и параметры конструктора полностью повторяют параметры базового класса *CDocTemplate*. Помимо конструктора, в классе реализованы все четыре "чистые" 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::`**`GetNextDoc`** `(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*, где содержится 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 "чистых" функций, в классе переопределены также функции базового класса *AddDocument* и *RemoveDocument*.

## Шаблоны многодокументных приложений

За решение этой задачи в библиотеке MFC отвечает класс *CMultiDocTemplate*, который определяет шаблон документа для многодокументного интерфейса. MDI-приложения используют главный фрейм как рабочее место, в котором пользователь может открывать произвольное число окон, отображающих данные того или иного документа. Возможна поддержка нескольких шаблонов 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 командных клавиш:

```
HMENU  CMultiDocTemplate::m_hMenuShared;
HACCEL CMultiDocTemplate::m_hAccelTable;
```

Поскольку в многодокументных приложениях одновременно могут быть открыты несколько документов, в классе *CMultiDocTemplate* предусмотрен защищенный член:

```
CPtrList CMultiDocTemplate::m_docList;
```

хранящий список открытых документов данного 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 CMultiDocTemplate::m_nUntitledCount;
```

Значение, хранящееся в этой переменной, используется для генерации имени по умолчанию и автоматически *увеличивается* при создании новых документов.

Для MDI-приложений определен, конечно же, конструктор класса. Помимо него, в классе *CMultiDocTemplate* реализованы также все четыре "чистые" функции базового класса: *GetFirstDocPosition*, *GetNextDoc*, *OpenFileName* и *SetDefaultTitle*, описание которых приведено выше. Кроме того, "наполне-

ны новым содержанием" еще три функции: *LoadTemplate*, *AddDocument* и *RemoveDocument*.

В заключение темы создания и использования шаблонов документов рассмотрим, каким образом с документом ассоциируется некоторая пиктограмма. Начнем с того, что написано в документации. "Пиктограмма, зарегистрированная 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 пиктограммы самому приложению, следующий — первому документу и т. д.". А теперь попытаемся разобраться, что же все-таки здесь имеется в виду, ведь при работе пиктограмма большей частью определяется своим 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 пиктограммы
                          // текстового документа
#define IDI_DRAWICON  130 // идентификатор пиктограммы
                          // графического документа
...
```

Прежде всего следует обратить внимание на то, что числа, характеризующие идентификаторы ресурсов документов и пиктограмм, принадлежат одному ряду. Это условие является обязательным, т. к. в противном случае на экране вы увидите пиктограмму по умолчанию вместо ожидаемой. И, наконец, числовые значения идентификаторов ресурсов документов и пиктограмм должны совпадать, иначе можно в окне текстового документа увидеть пиктограмму графического и наоборот. Во избежание подобных недоразумений рекомендуем для пиктограмм использовать тот же идентификатор, что и для ресурсов документов: IDR_NOTETYPE и IDR_DRAWTYPE.

# Место объекта-приложения в архитектуре "документ/представление"

Класс *CWinApp* был нами достаточно подробно рассмотрен, и нет смысла повторяться. Однако мы пока не касались его свойств, связанных с архитектурой "документ/представление", отложив обсуждение этого вопроса до более удобного момента. Теперь пришло время заполнить этот пробел. Рассмотрим кратко функции класса *CWinApp*, которые отвечают за взаимо-

действие 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:

```
// Фрагмент кода из файла NoteDraw.cpp
...
CMultiDocTemplate* pDocTemplate;
// Создаем объект шаблона документа для работы с текстом
pDocTemplate = new CMultiDocTemplate(IDR_NOTETYPE,
                                     RUNTIME_CLASS(CNoteFrame),
                                     RUNTIME_CLASS(CNoteDoc),
                                     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 и этот шаблон в список
AddDocTemplate(pDocTemplate);
...
// Фрагмент кода из файла NoteDraw.rc
...
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
...
```

Как вы прекрасно понимаете, если к созданным и добавленным шаблонам не будет доступа, то зачем они вообще нужны. Естественно, в библиотеке есть соответствующие функции:

`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 шаблон создает 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 этих команд возлагается на библиотеку MFC, в карту сообщений класса приложения 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)
```

если вы хотите обрабатывать команды самостоятельно. При этом на программиста ложится также реализация обработчика *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 класса *CWinApp*.

```
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*.

Наибольший интерес здесь представляет тот факт, что приложение может одновременно поддерживать несколько шаблонов (или типов) документов (рис. 17.3).

Для выбора необходимого шаблона при обработке команды ID_FILE_NEW создается и выводится на экран специальный блок диалога (рис. 17.4) на основе реализованного в библиотеке MFC (файл <docmgr.cpp>) класса *CNewTypeDlg*.

Он позволяет пользователю просматривать список имеющихся у приложения (добавленных) шаблонов документов и выбирать один из них.

**Примечание**

В принципе, вместо класса *CNewTypeDlg* можно использовать и созданный самостоятельно, но я не вижу для этого серьезных причин.

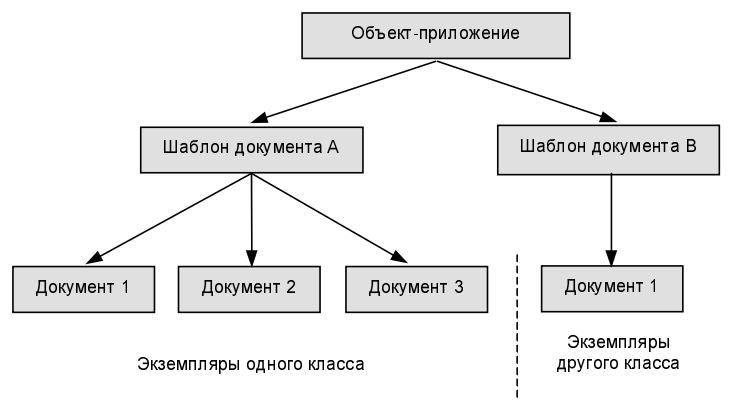



**Рис. 17.3.** Несколько документов двух типов, используемых одним приложением




**Рис. 17.4.** Блок диалога для выбора типа вновь создаваемого документа

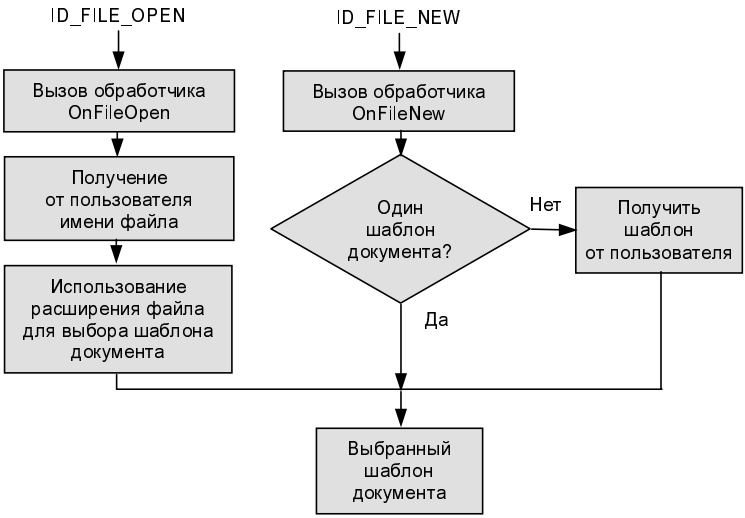



**Рис. 17.5.** Процесс создания шаблона документа

Таким образом, если внимательно посмотреть на перечисленные функции и переменные 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. 17.5.

# Роль фреймов в архитектуре "документ/представление"

Фрейм документа, который мы будем здесь рассматривать, имеет две основные составляющие — собственно фрейм и его окно, представляющее на экране данные документа. Эти два компонента представлены и управляются различными группами классов библиотеки MFC:

- классы фреймов (*CFrameWnd* и *CMDIChildWnd*);
- классы представлений (*CView*, *CScrollView* и другие).

Процесс создания главного и дочерних фреймов подробно обсуждался в главах 6 и 13, 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 используется макрос
  DECLARE_DYNCREATE(CNoteFrame)
protected:
  CNoteFrame();  // конструктор
...
};
```

и

```
// Реализация класса (файл Childfrm.cpp)
// Для возможности динамического создания объекта
// класса CNoteFrame используется макрос
IMPLEMENT_DYNCREATE(CNoteFrame, CMDIChildWnd)
...
```

Как вы уже поняли, интерес здесь представляют два макроса, подробно описанные в главе 1 — 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*, т. е. 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 во время выполнения программы и сериализации. Забыв включить эти макросы в объявление и, соответственно, в реализацию некоторого класса, вы тем самым исключаете для него возможность корректной работы механизмов RUNTIME_CLASS и сериализации.

Рассмотрим теперь функции класса *CFrameWnd*, которые связаны с архитектурой "документ/представление".

```
void BOOL CFrameWnd::OnCreateClient(
    LPCREATESTRUCT lpcs,
    CCreateContext *pContext)
```

Вызывается только библиотекой 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();
};
```

Эта структура создается шаблоном документа (при создании документа и ассоциированных с ним компонентов). Перечислим ее поля (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 выполняет то же самое 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 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*.

**Примечание**

Об областях мы подробно поговорим немного позже.

```
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 рассмотрим фрагмент кода из приложения NoteDraw, где показывается, как можно переопределить функцию *OnCreateClient* для создания статического разделенного окна с двумя горизонтальными областями, в каждой из которых создается свое представление (рис. 17.6).

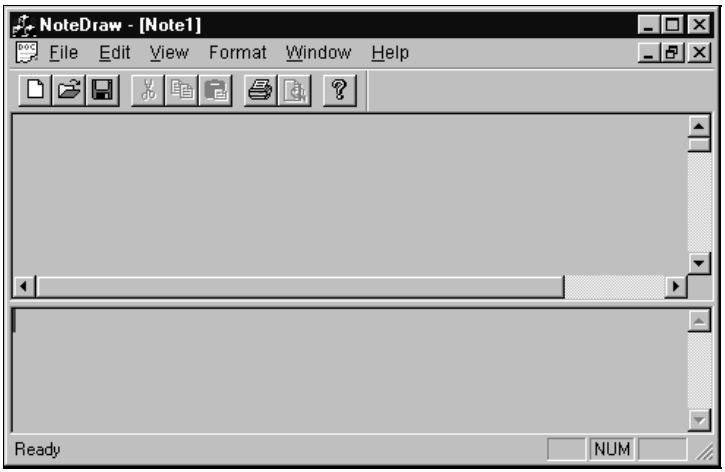



**Рис. 17.6.** Результат выполнения переопределенной функции *OnCreateClient*

**Примечание**

Более подробно все перечисленные действия мы рассмотрим 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 вторую горизонтальную область
  SetActiveView((CView*)m_spltWnd.GetPane(1, 0));
  return TRUE;
}
```

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 рассмотрены нами в главе 5.

И, наконец, последнее. Мы уже говорили, что с каждым документом ассоциируются определенные ресурсы меню и командных клавиш. 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 18);
- создать необходимое число представлений каждого документа (см. главу 18);
- 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.

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.

# Создание каркаса приложения на базе архитектуры "документ/представление"

В этом разделе мы познакомимся с тем, как создать проект, в котором реализована поддержка всего того, о чем мы только что говорили:

1. Вызвав диалог New, выберите в нем тип приложения MFC AppWizard (exe), а в поле Project Name — имя создаваемого приложения NoteDraw. После нажатия кнопки OK запустится мастер AppWizard и вы увидите его первое окно — рис. 17.7.

   В этом окне представлены все три основных типа приложений Windows: однодокументное, многодокументное и на базе блока диалога.

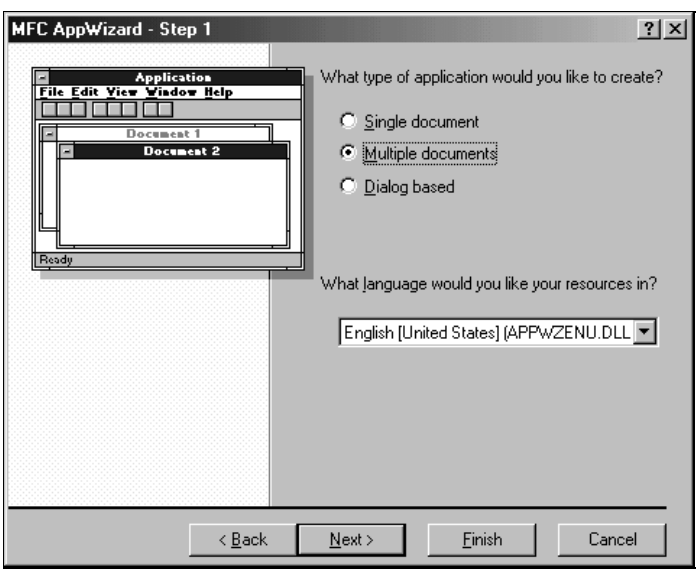



**Рис. 17.7.** Первое окно мастера AppWizard

2. Установите флажок Document/View architecture support?, если вы работаете с Visual C++ 6.0, выберите переключатель Multiple documents, и нажмите кнопку Next.
3. Смело дважды нажимайте кнопку Next, чтобы перейти к третьему окну мастера, поскольку во втором окне задается поддержка для работы с ба-

замами данных. В третьем окне сбросьте флажок ActiveX Controls (элементы управления ActiveX) и переходите к следующему окну.

4. В этом окне (рис. 17.8) предлагается выбор некоторых дополнительных возможностей. Можно спокойно оставить предложенные по умолчанию установки, поскольку именно этими возможностями мы и будем заниматься.

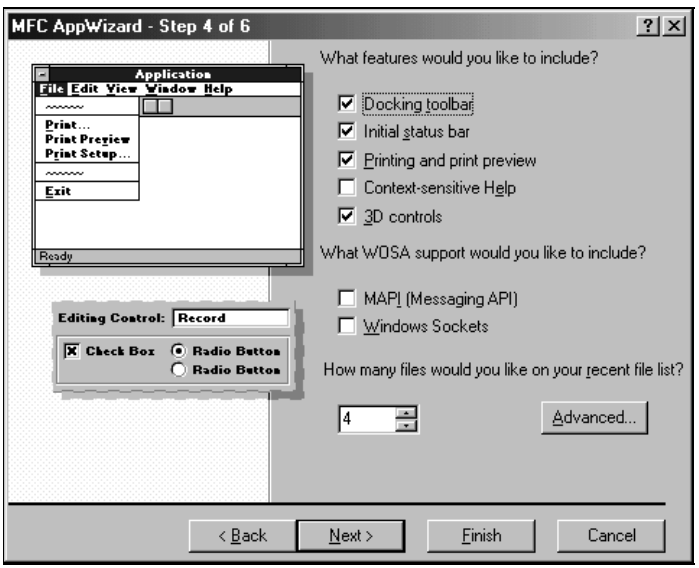



**Рис. 17.8.** Четвертое окно мастера AppWizard

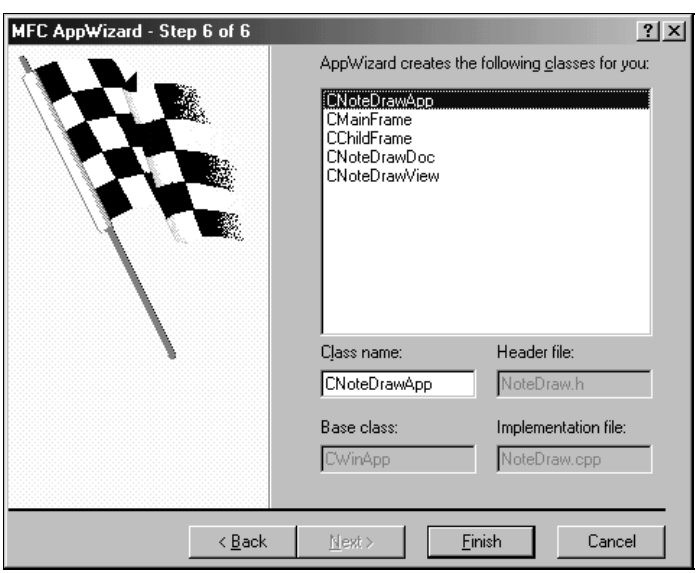



**Рис. 17.9.** Последнее окно мастера AppWizard

5. Дважды нажмите кнопку Next, чтобы перейти к последнему окну мастера (рис. 17.9), в котором можно изменить имена создаваемых классов, а также сами базовые классы.

6. После того как все изменения выполнены, нажмите кнопку Finish (Готово). На экране появится окно с информацией о проекте. После нажатия кнопки OK мастер создаст для вас каркас приложения и завершит свою работу.

Именно в этот созданный каркас приложения мы и будем добавлять новый код.

# ГЛАВА 18



# Документ и его представления

Если ты изложишь дело так ясно, что тебя сможет понять каждый, кто-нибудь обязательно не поймет.

*Уилл Роджерс*

Как я уже неоднократно отмечал, одним из основных достоинств архитектуры "документ/представление" является то, что в ней данные отделены от их изображения.

**Примечание**

Точнее, архитектура позволяет это сделать, и мы рекомендуем к этому стремиться.

Любые изменения данных осуществляются с использованием 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 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. 18.1. 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 основные компоненты архитектуры "документ/представление" подробно. Начнем с документов.

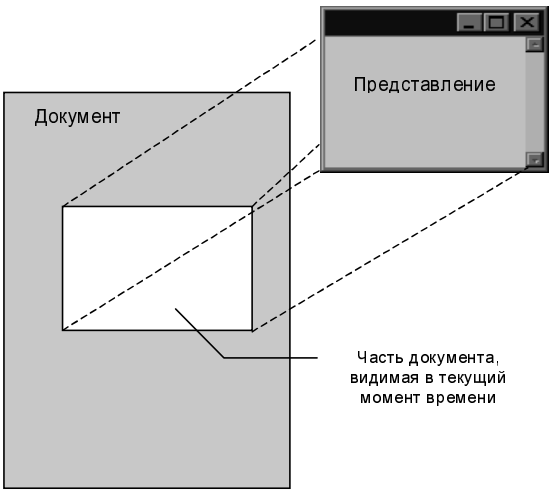



**Рис. 18.1.** Взаимоотношения документа и представления

# 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.

Для реализации документа в типичном приложении необходимо проделать следующую последовательность действий (рис. 18.2):

- 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 и 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*,

представляющий 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;
...
};
```

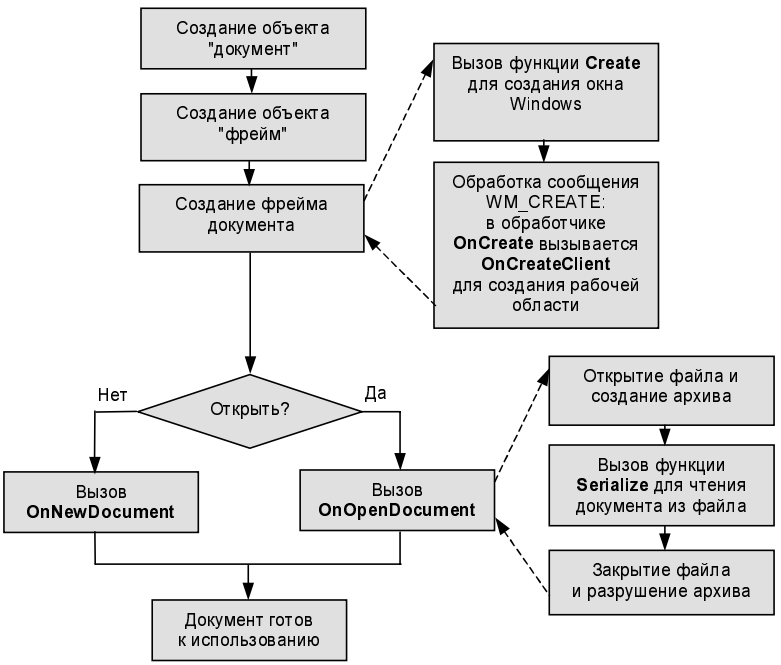



**Рис. 18.2.** Последовательность действий при создании документа

В данном случае список *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 (рис. 18.3). Он поддерживает все стандартные операции, такие как создание документа, его загрузка и

сохранение. Библиотека MFC работает с документами, используя интерфейс, определенный в *CDocument*.

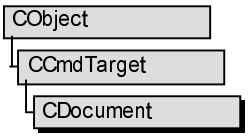



**Рис. 18.3.** Место класса *CDocument* в иерархии библиотеки MFC

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 некоторые 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.

`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*) и передачи 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 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 переини-

циализирует 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.

Еще раз внимательно посмотрите на рис. 18.2. На нем наглядно показано, что документ может быть создан в двух случаях. В первом случае по команде ID_FILE_NEW создается новый, пустой документ. В этом случае для его инициализации достаточно переопределить функцию класса документа *OnNewDocument*. Во втором 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 MFC. Речь, конечно же, идет о сериализации или, другими словами, о преобразовании в последовательную форму. 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 необходимость включить специальные макросы в объявление:

```
class CParagraph : public CObject
{
protected:
  CParagraph();
  // 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);
  }
}
```

Последний представленный фрагмент приведен для иллюстрации следующего положения. При сохранении параметров выбранного шрифта, которые хранятся 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 удобная, а во многих случаях и необходимая функция.

## 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 составляющих.

С одной стороны, наличие определенной виртуальной функции (*Serialize*) позволяет унифицировать процесс сохранения/восстановления объектов. Впрочем, это обычная практика для программ, написанных с использованием практически любого объектно-ориентированного языка. Для реализации процесса следует только переопределить виртуальную функцию, вызов которой по указателю на объект одного из базовых классов приведет к вызову нужной функции. Библиотека 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 неизвестного заранее типа. Например, приложение должно сохранять и восстанавливать некоторое количество объектов различного типа. Естественно, что для восстановления объекта посредством вызова соответствующего конструктора необходимо точно знать тип создаваемого объекта. 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 обеспечить систему соответствующей информацией, используя макросы DECLARE_SERIAL и IMPLEMENT_SERIAL при объявлении и в реализации соответственно.

Теперь давайте рассмотрим все вышесказанное на примере. Класс *CGraphicsView* имеет список указателей на объекты класса *CDrawObject* и производных 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.

Посмотрите 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: во-первых, это конструкторы без параметров (реализация конструкторов по умолчанию), которые объявлены как защищенные, а во-вторых, это 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:

1. `CObject::CObject()`
2. `CDrawObject::CDrawObject()`
3. `CLine::CLine()`
4. `CRectangle::CRectangle()`
5. `CLine::Serialize()`

Поскольку класс *CRectangle* не переопределяет функцию *Serialize*, используется реализация функции базового класса.

Ниже представлено содержимое файла, в котором располагаются три объекта-прямоугольника и два объекта-линии в следующем порядке: прямоугольник, линия, прямоугольник, прямоугольник, линия. Одинарной чертой под-

черкнуто описание типа объекта, а 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, а далее тип определяется номером, указанным в ссылке. Не правда ли, удобно?

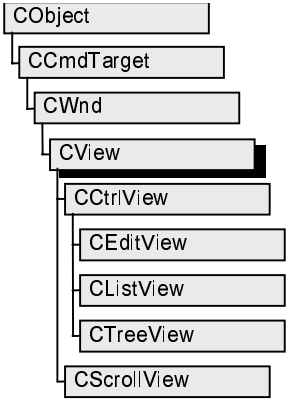
```
0000: 05 00 00 00 FF FF 01 00 0A 00 43 52 65 63 74 61 ¦ _...___...CRecta
0010: 6E 67 6C 65 26 00 00 00 19 00 00 00 58 00 00 00 ¦ ngle&..._...X...
0020: 4B 00 00 00 01 00 00 00 00 00 00 00 01 00 00 00 ¦ K..._........_...
0030: 00 00 00 00 00 00 00 00 26 00 00 00 19 00 00 00 ¦ ........&..._...
0040: 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...
```

Закончив знакомство с документами, пора переходить к тому, каким образом можно взаимодействовать с хранящимися в них данными. Осуществить это взаимодействие и визуализацию данных документа призваны объекты специальных классов — представления.

# Представления

Это специальная группа классов дочерних окон фрейма (рис. 18.4), которая отвечает за отображение данных документа и за взаимодействие с пользователем. В этом разделе мы представим возможности, предоставляемые для этих задач библиотекой 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.

**Рис. 18.4.** Место классов представлений в иерархии библиотеки MFC

Достаточно часто возникает желание иметь различные представления 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. 18.5).

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.

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 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. Если представление само не обрабатывает команду, то передает ее дальше "по маршруту", как описано в главе 5.

Последовательность действий по созданию представлений, выполняемая библиотекой MFC, представлена на рис. 18.6.

Теперь, после того как сложилось некоторое общее представление о назначении и возможностях этой группы классов, можно переходить к рассмотрению компонентов группы и свойств ее составляющих.

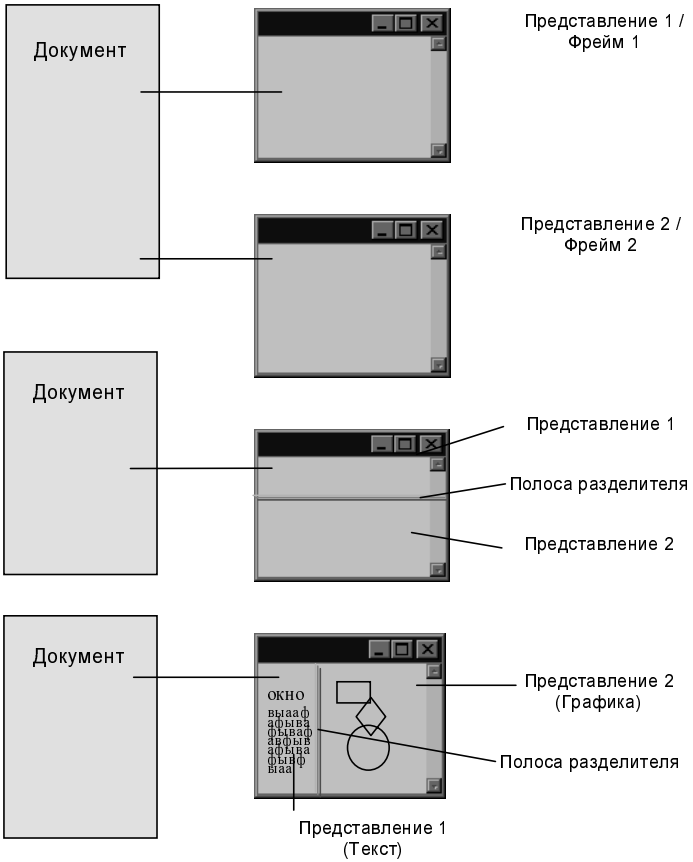



**Рис. 18.5.** Использование нескольких представлений для отображения данных одного документа

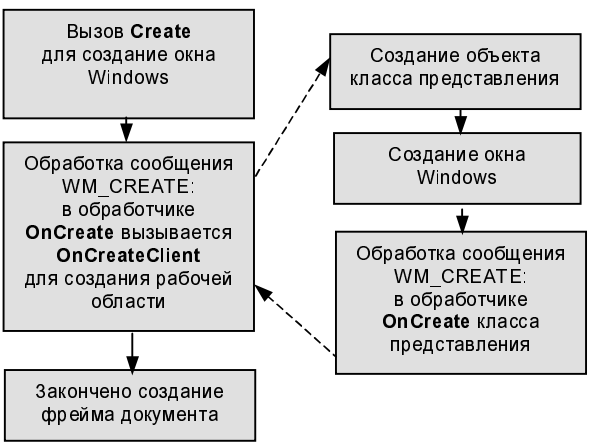



**Рис. 18.6.** Последовательность создания представления

В полном соответствии с общей идеологией построения библиотеки MFC на вершине иерархии 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 MFC, начиная с версии 4.0, для того, чтобы можно было использовать в архитектуре "документ/представление" деревья (класс *CTreeView*), списки (*CListView*), а также простейшие (*CEditView*) и расширенные (*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.

Всего в библиотеке MFC реализованы двенадцать классов представлений, некоторые из них мы рассмотрим подробнее. 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 данного 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 перечисляемый 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 объекта во время выполнения программы. Если оба параметра *lHint* и *pHint* 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* функции *CDocument::UpdateAllViews*, рассмотренной ранее, 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 действия,
      // необходимые для инициализации представления CNoteView
      ...
    }
  }
  // Обновляем все, хотя можно и ограничить размеры области
  Invalidate(TRUE);
}
```

В приведенной выше 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. Однако 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 следующих 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 MFC для определения возможности прокрутки. Параметр *bDoScroll* определяет, нужно ли осуществлять прокрутку. Если *bDoScroll* = TRUE, то, когда представление получает сообщение о прокрутке, ее действительно нужно осуществить. Если же *bDoScroll* = FALSE, то при перенесении элемента OLE 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 BOOL CView::OnScrollBy (
    CSize sizeScroll,
    BOOL bDoScroll)
```

Вызывается библиотекой MFC, когда курсор мыши находится вне области, изображаемой представлением документа, что происходит или при переносе элемента OLE к границе текущего представления, или при манипуляциях горизонтальной или вертикальной полосами прокрутки. Реализация по умолчанию не делает ничего. В производных классах функция проверяет, видимо ли представление, прокручиваемое в направлении, запрошенном пользователем, и, при необходимости, обновляет новый регион. Она автоматически вызывается функциями *CWnd::OnHScroll* и *CWnd::OnVScroll* для выполнения прокрутки. Параметр *sizeScroll* определяет число пикселов (по горизонтали и по вертикали), на которое осуществляется прокрутка. Функция возвращает TRUE, если представление может быть прокручено, и FALSE — в противном случае.

```
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 ис-

пользует 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 рассматриваемой группы.

## Класс *CCtrlView*

Это базовый класс для группы классов элементов управления, включающей в себя *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 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 его присоединения 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 используемый по умолчанию стиль — *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 применяться для реализации простейшего текстового редактора. При этом, поскольку класс создан на базе *CView*, его объекты 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.

Теперь, после небольшого обзора возможностей класса, рассмотрим его основные члены.

`static AFX_DATA const DWORD CEditView::dwStyleDefault`

Определяет для всех объектов класса *CEditView* стиль, используемый по умолчанию.

`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.

**Примечание**

Он же является ярким примером нарушения принципа разделения данных документа и их изображения.

`CFont* CEditView::`**`GetPrinterFont`** `()`

Возвращает указатель на объект *CFont*, который описывает текущий шрифт принтера, или NULL, если его шрифт не был установлен. В последнем случае печать осуществляется экранным шрифтом.

`void CEditView::`**`SetPrinterFont`** `(CFont* pFont)`

Устанавливает для принтера шрифт, определяемый указателем *pFont* на объект класса *CFont*. Данную функцию следует вызывать из функции *OnPreparePrinting* для того, чтобы установка нового шрифта произошла до начала печати содержимого представления. Используется в случае, когда для принтера нужен особый шрифт. Если параметр *pFont* равен NULL, то шрифт, используемый для печати, основывается на экранном шрифте.

```
UINT CEditView::PrintInsideRect (
    CDC *pDC,
    RECT &rectLayout,
    UINT nIndexStart,
    UINT nIndexStop)
```

Печатает текст в прямоугольнике, заданном в параметре *rectLayout*. Если у объекта класса не установлен стиль ES_AUTOHSCROLL, то текст переносится на следующую строку внутри прямоугольника. В противном случае текст отсекается по его правой границе. Поле *rectLayout.bottom* изменяется так, что размеры определяют ту часть начального прямоугольника, которую занимает текст. Параметры *nIndexStart* и *nIndexStop* показывают, соответственно, индексы первого и следующего за последним полученных символов относительно начала буфера. Функция возвращает индекс следующего символа, который будет печататься.

`void CEditView::GetSelectedText (CString &strResult)`

Копирует выделенный текст или символы, предшествующие первому символу "возврат каретки", в объект класса *CString*, на который ссылается параметр *strResult*.

`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), выводимым на экран в ответ на стандартные команды ID_EDIT_FIND и ID_EDIT_REPLACE, и предоставляет 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 *bNex:* 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 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). Функция 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 через вызов 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. 18.7 и 18.8 представлен внешний вид блоков диалога Find и Replace в режимах поиска и замены, соответственно.

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:




**Рис. 18.7.** Стандартный блок диалога Find




**Рис. 18.8.** Стандартный блок диалога Replace

```
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);
}
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");
}
```

## 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 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, основываясь 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.
- Если 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.
- Если 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 класса *CScrollView*.

`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)
```

Функция 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. 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 про-

крутки, 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. 18.9) . Это наиболее часто используется для реализации операции "Zoom to fit".

Обе 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 (рис. 18.10—18.12):

```
void CScrollView::ResizeParentToFit (BOOL bShrinkOnly = TRUE)
```

Приводит размеры родительского фрейма в соответствие с размерами представления. Характер изменения размера определяется параметром *bShrinkOnly*. Если его 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.

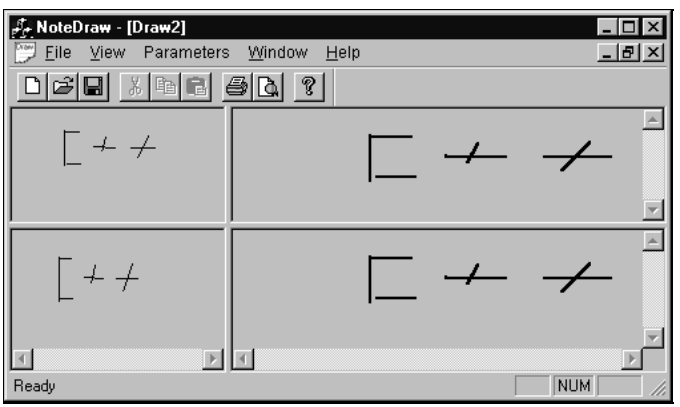



**Рис. 18.9.** Разделенное окно представления в режиме масштабирования

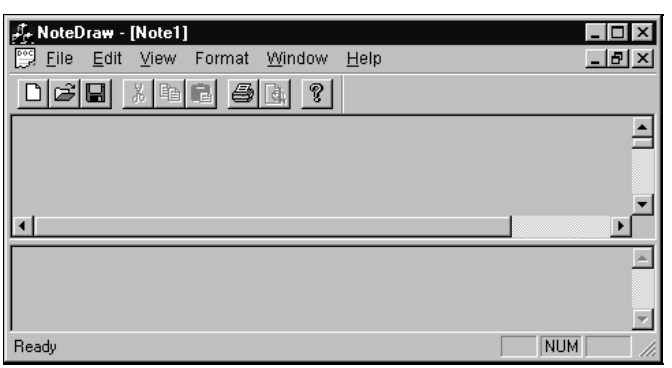



**Рис. 18.10.** Фрейм документа до выполнения функции *ResizeParentToFit*

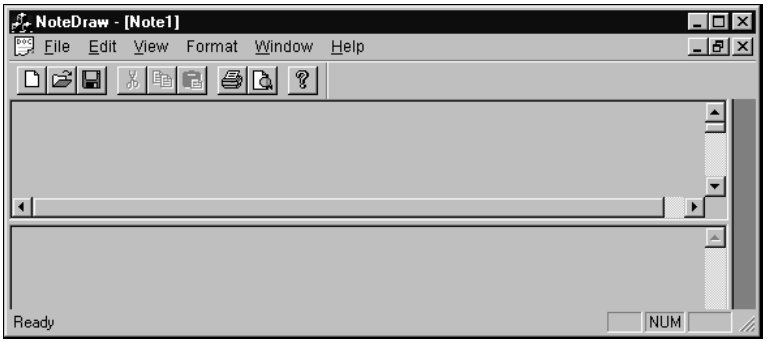



**Рис. 18.11.** Фрейм документа после выполнения функции *ResizeParentToFit*(FALSE)

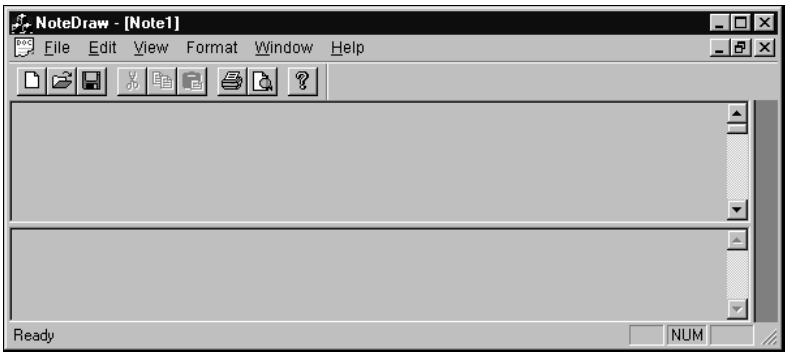



**Рис. 18.12.** Фрейм документа после выполнения функции *ResizeParentToFit*(TRUE)

## Класс *CSplitterWnd*

Прокрутка позволяет работать с документом, размер которого больше, чем можно отобразить в окне, при этом большая часть документа скрыта от пользователя. Вторым вариантом работы с большим документом, который мы также рассматривали, является масштабирование. Но если размеры документа достаточно велики, то такой способ хорош только для получения общей информации об 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 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*, имеющим 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. 18.13) 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.

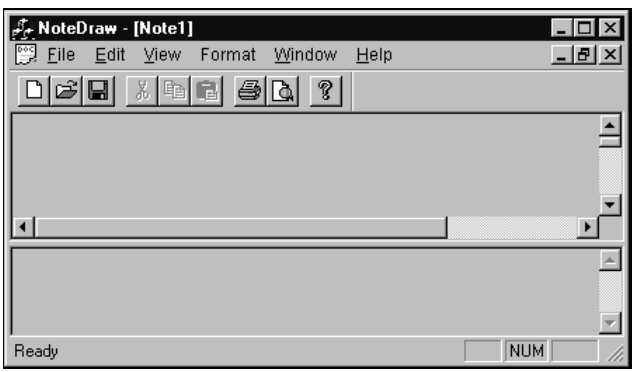



**Рис. 18.13.** Статическое разделенное окно с двумя горизонтальными областями

В динамических разделенных окнах (рис. 18.14) 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 созда-

ется 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.

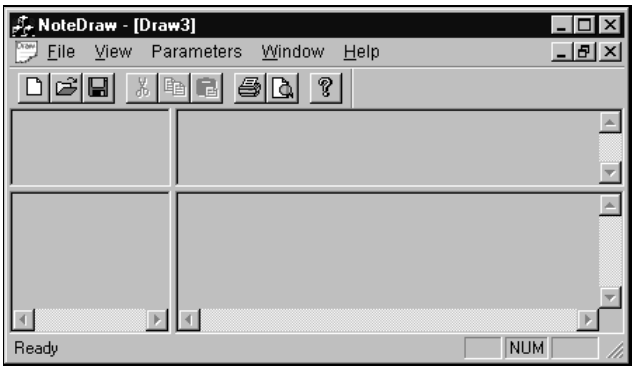



**Рис. 18.14.** Динамическое разделенное окно с двумя горизонтальными и двумя вертикальными областями

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 |
|---|---|
| **Для статических окон** | |
| 1 строка × 1 столбец | Для областей, основанных на различных объектах |
| 2 строки × 1 столбец | |
| 2 строки × 2 столбца | Для областей, основанных 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 она 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 при перемещении пользователем ползунка полосы прокрутки соответствующее сообщение посылается во все области.

Полосы прокрутки формируются при создании разделяемого окна. Например, объект класса *CSplitterWnd*, имеющий 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 предоставляются 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 ()*.

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 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 случае — 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, может иметь 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, что указывает на создание динамического разделяемого окна.

```
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 допускается и создание любого оконного класса, не обязательно основанного на представлении. Реализовать данную процедуру следует до завершения функции *OnCreateClient* родительского фрейма, чтобы обеспечить корректное изображение 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:

*GetRowCount, GetColumnCount, GetRowInfo, GetColumnInfo, SetRowInfo, SetColumnInfo, GetPane, IsChildPane, IdFromEowCol, RecalcLayout, GetScrollStyle, SetScrollStyle.*

Следующая группа функций может быть переопределена в производных классах для реализации индивидуальных возможностей:

*OnDrawSplitter, OnInvertTracker, DeleteView, SplitRow, SplitColumn, DeleteRow, DeleteColumn, GetActivePane, SetActivePane, CanActivateNext, ActivateNext, DoKeyboardSplit, DoScroll, DoScrollBy, CreateScrollBarCtrl.*

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 19

# Печать и предварительный просмотр документов

По почерку принтера можно судить о том,
как нервничает компьютер.
*Дмитрий Пашков*

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: сначала 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.

Следует совершенно четко представлять, что вывод на принтер осуществляется исключительно через буфер, который, однако, служит не только 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 документа, 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 формируется приложением Print Manager (Диспетчер печати). Для печати документа приложения помещают в эту очередь свои данные (задания на печать), выводимые 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 немного. 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 обеспечивается возможность принудительного (аварийного) завершения печати, заполняется структура DOCINFO и вызывается функция *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 либо 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 MFC и рассмотрим, как реализован в ней 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 этого стандартного обработчика вполне достаточно, 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. Прежде всего добавляем в карту сообщений класса приложения оператор

```
ON_COMMAND(<ID_PRINT_SETUP>, <OnPrintSetup>)
```

**Примечание**

Как вы понимаете, 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 $8^1/_2 \times 11$ in и альбомную ориентацию (рис. 19.1).

```
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
    ...
  }
}
```

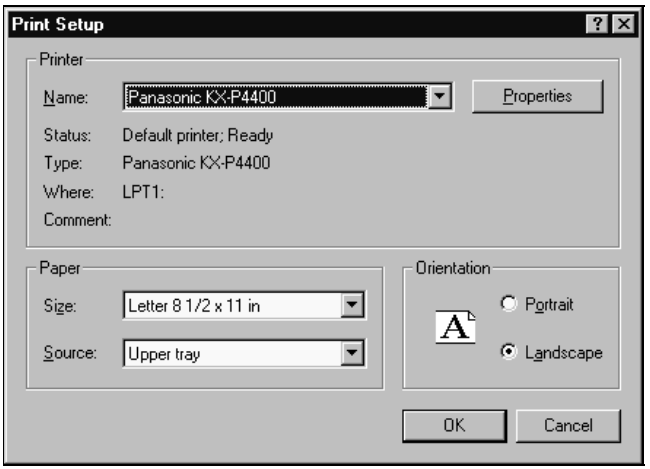



**Рис. 19.1.** Стандартный блок диалога Print Setup

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:

```
...
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 процесса печати, 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. 19.2 представлена схема цикла печати, реализованная в библиотеке 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*

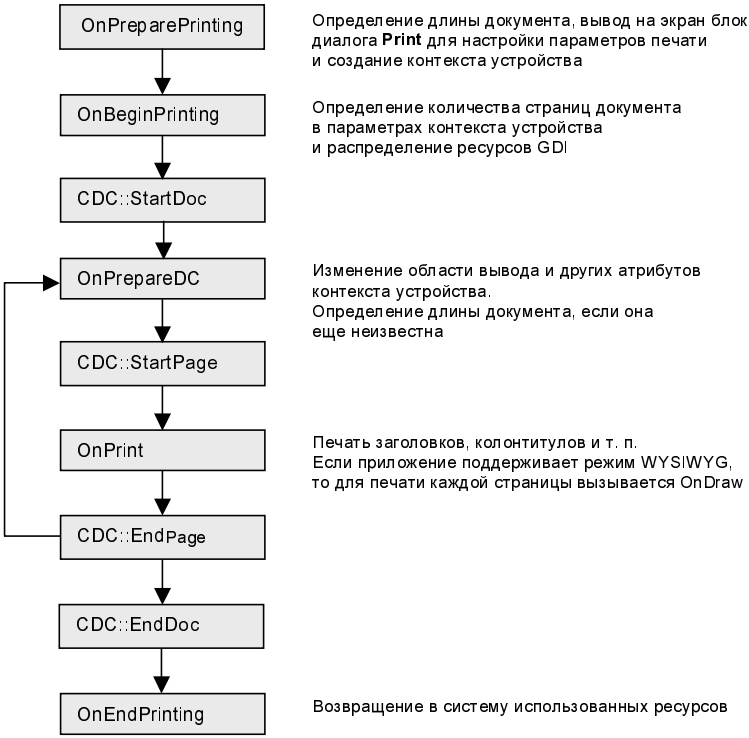



**Рис. 19.2.** Цикл печати библиотеки MFC

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 в библиотеке MFC как сам цикл, так и все участвующие в нем функции уже реализованы. Наряду с этим, благодаря механизму виртуальных функций, программисту предоставляются широкие возможности для "более тонкой" настройки всего процесса печати. Виртуальные функции *OnPreparePrinting*, *OnBeginPrinting*, *OnPrepareDC*, *OnPrint* и *OnEndPrinting* определены в классе *Cview*, и при их переопределении необходимо соблюдать логику, заложенную в библиотеке MFC. Но сначала рассмотрим сами функции:

`virtual BOOL CView::`**`OnPreparePrinting`** `(CPrintInfo *pInfo)`

Вызывается библиотекой 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:

**Примечание**

О режиме предварительного просмотра документа см. далее в этом разделе.

```
struct CPrintInfo
{
  CPrintInfo();              // Конструктор
  ~CPrintInfo();             // Деструктор
  CPrintDialog *m_pPD;       // Указатель на объект "блок диалога Print"
  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 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. 19.3).

```
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);
}
```

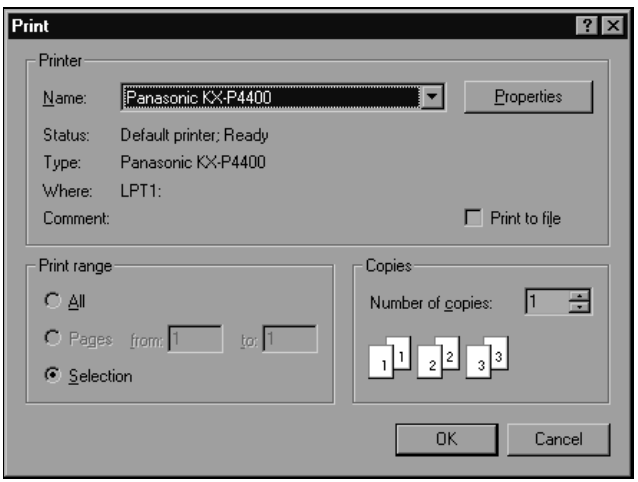



**Рис. 19.3.** Стандартный блок диалога Print

Если необходим предварительный доступ к контексту устройства, то это можно сделать, переопределив функцию:

```
virtual void CView::OnBeginPrinting (
    CDC *pDC,
    CPrintInfo *pInfo)
```

Вызывается библиотекой 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 печа-

ти. Кроме того, эту функцию можно использовать для инициализации, зависящей от свойств контекста устройства принтера, например, от числа печатаемых 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 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 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. Можно принудительно остановить процесс печати, 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 главе 18 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.

В принципе, можно было бы просто переопределить функцию *OnDraw* для организации вывода на принтер, но ведь печать осуществляется на листы бумаги, имеющие вполне определенные размеры, которые 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 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 только 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 MFC.

Начиная по командам ID_FILE_PRINT или ID_FILE_PRINT_DIRECT процесс печати, библиотека MFC вызывает функцию *OnPreparePrinting* нашего 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 определить 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 перед вызовом этой функции их можно изменить. Те параметры, которые пользователь установил в блоке диалога, доступны через поля структуры *CPrintInfo*. Так, например, если пользователь задал 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;
- посылка 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*, и 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*.

**Примечание**

Действительно — зачем дублировать код, если в Windows организация вывода не зависит от конкретного устройства (аппаратно-независимый вывод).

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.

**Примечание**

...если вы все-таки последовали рекомендациям и запросили их у системы в функции *OnBeginPrinting*.

Таким образом, для того чтобы "организовать" процесс печати, используя возможности библиотеки 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);
  // Формируем строку с номером страницы
  TCHAR szBuf[32];
  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 по выводу документа на экран и на принтер, библиотека 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 пред-

варительного 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 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. 19.2). 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 на примере вывода рамки вокруг текста в режиме предварительного просмотра (рис. 19.4).

```
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);
  }
}
```

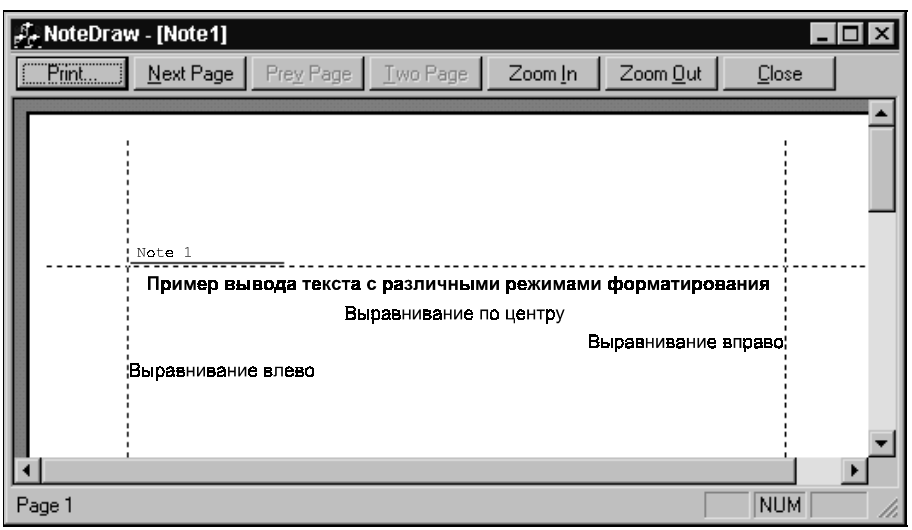



**Рис. 19.4.** Режим предварительного просмотра документа

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 который и осуществляется вывод. И когда приложение выполняет какие-либо операции, изменяющие 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 надобности 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 последовательность действий.

Поскольку все начинается с выбора команды ID_FILE_PRINT_PREVIEW, которой соответствует обработчик *OnFilePrintPreview*, рассмотрим его реализацию, предоставляемую по умолчанию — ведь 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 командных клавиш
  CPrintPreviewState();  // Конструктор
};
```

Основным ее назначением является сохранение параметров активного фрейма, которые он имел перед переходом в режим предварительного просмотра. При необходимости их в этом режиме можно использовать.

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 runtime-структуру 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*, обеспечивающего 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 *DoPrintPreview* захочет реализовать свою функцию, то не следует выпускать какую-либо из них из поля зрения. Но прежде чем это делать, стоит задаться вопросом: "А зачем?", и только после получения четкого 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 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 занимаемую структурой *CPrintPreviewState* память. На структуру указывает *pState*, значение которого устанавливается функцией *DoPrintPreview*.

Представленных функций в подавляющем большинстве случаев вполне достаточно. Переопределяя функции, можно без лишних усилий решать такие, например, задачи:

- ввести полосу прокрутки для доступа к страницам документа в режиме предварительного просмотра;
- в режиме предварительного просмотра отображать страницы не с первой, а с текущей;
- 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. 19.5.

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. 19.6).

```
class CNoteView : public CScrollView
{
...
public:
  //{{AFX_MSG(CNoteView)
  afx_msg void OnPageSetup();
  //}}AFX_MSG
DECLARE_MESSAGE_MAP()
};
```

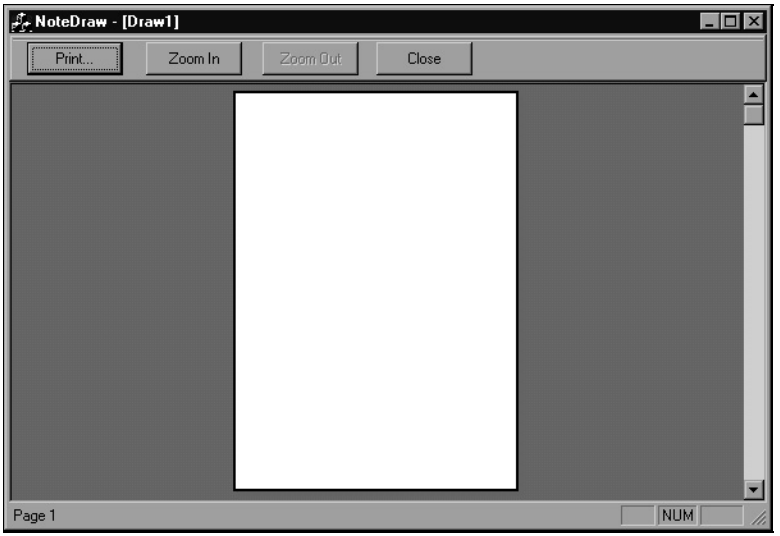



**Рис. 19.5.** Окно предварительного просмотра с измененной панелью инструментов

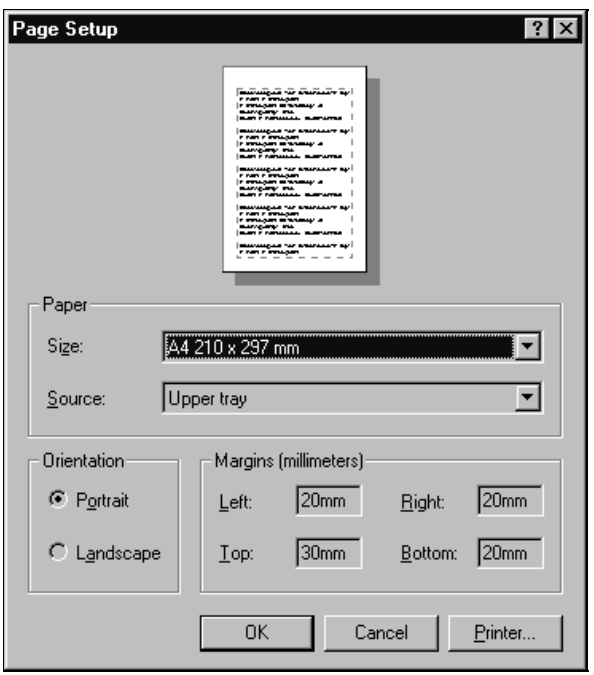



**Рис. 19.6.** Блок диалога Page Setup

```
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);
  }
}
```



# Часть VII

## Работаем с базами данных

**Глава 20.** Библиотека MFC и базы данных

**Глава 21.** Настройка приложения на работу с базами данных

ГЛАВА 20



# Библиотека MFC и базы данных

Соединяй приятное с полезным.
*Козьма Прутков*

Прежде чем начинать создавать приложения для работы с базами данных нужно познакомиться с некоторыми основными понятиями данной темы. Одним из них является понятие ODBC — Open Database Connectivity (Открытый интерфейс для подключения к базам данных), обозначающее наиболее универсальную на настоящий момент технологию, посредством которой можно получить доступ к базам данных.

## Что такое ODBC

ODBC является одной из самых старых технологий для работы с базами данных, которые выпустила фирма Microsoft. Одной из основных причин разработки ODBC была необходимость предоставления программистам простого способа доступа к содержимому баз данных с минимальной ориентацией на какой-либо конкретный язык. Среди платформ программирования, на которые фирма Microsoft изначально ориентировала на работу с ODBC, находится и Visual C++.

Основные принципы ODBC стандартны для Windows — для выполнения работы используются соответствующие драйверы, содержащиеся в DLL (Dynamic Link Library, Библиотека динамической компоновки). Драйверы, которые могут оказаться необходимыми для работы, можно получить непосредственно от поставщика базы данных — чаще всего прямо в пакете программ. В частности, Visual C++ поставляется со следующими входящими в пакет драйверами:

- SQL Server
- Microsoft Access
- Microsoft FoxPro
- Microsoft Excel
- dBase
- Paradox
- Text File (специальным образом отформатированные текстовые файлы)

Список драйверов ODBC, инсталлированных на вашем компьютере, можно увидеть на вкладке Drivers (Драйверы) диалогового окна ODBC Data Source Administrator (рис. 20.1).

**Примечание**

Для вывода на экран диалогового окна ODBC Data Source Administrator просто дважды щелкните левой кнопкой мыши по значку ODBC в Control Panel (панель управления).

Чтобы получить доступ к базе данных с использованием ODBC, необходимо создать источник данных. Но мы пойдем по несколько некорректному, но упрощенному пути. Для создания примера мы не будем создавать ни источник данных, ни саму базу данных, а воспользуемся уже готовым продуктом. А именно, возьмем поставляемую с Microsoft Access базу данных Northwind и созданный для нее источник данных.

После этого небольшого отступления переходим непосредственно к созданию каркаса приложения, "умеющего" работать с базами данных.

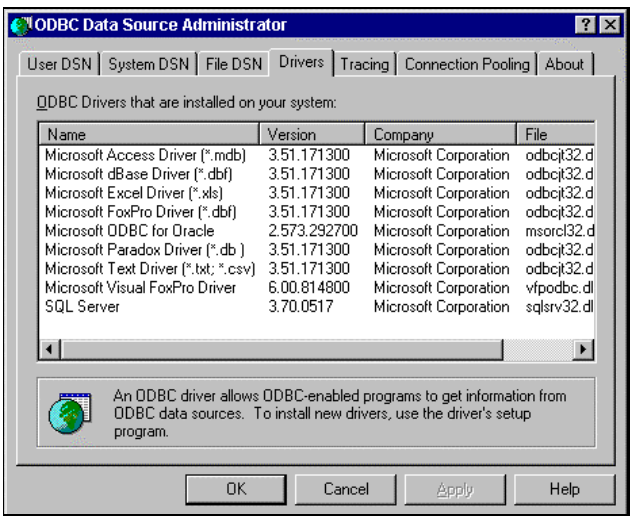



**Рис. 20.1.** Список инсталлированных драйверов ODBC

# Проект MFC AppWizard

Прежде всего надо создать (или выбрать) саму базу данных. Однако, если пользователю, который знать не знает баз данных, предоставить просто

удобный для программирования интерфейс, хорошего получится мало. Вывод отсюда такой — необходимо создание специализированных приложений, основная цель которых, помимо оптимальности выполнения запросов к системе управления базами данных, заключается в удобном интерфейсе, не требующем *никаких* специальных знаний о базах данных. И здесь неоценимую помощь может оказать именно Visual C++, с помощью которого такое приложение создать достаточно просто.

Первым шагом на этом пути является выбор в качестве проекта MFC AppWizard, первое окно которого представлено на рис. 20.2.

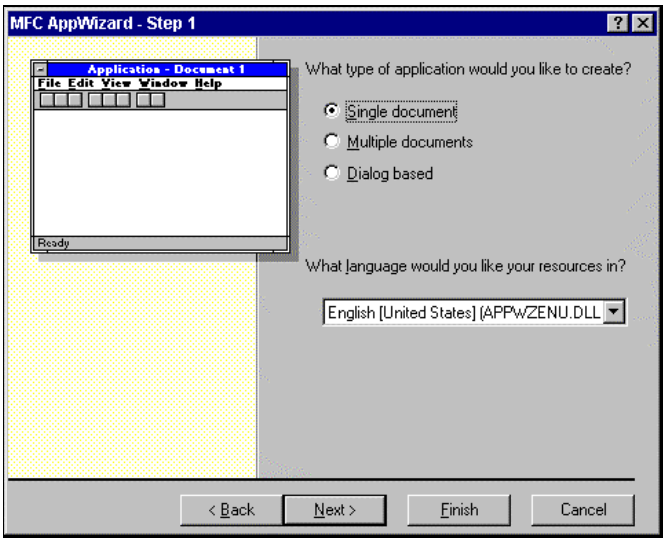



**Рис. 20.2.** Первое окно мастера MFC AppWizard

Начнем с создания самого простого приложения для работы с базой данных. Поэтому выберите переключатель Single document и нажмите кнопку Next. В результате мы перейдем к самому интересному для нас окну мастера MFC AppWizard — Step 2 of 6, показанному на рис. 20.3 — именно здесь мы будем определять параметры базы данных.

Первое, что сразу бросается в глаза — это группа переключателей What database support would you like to include? (Какую поддержку базы данных вы хотите включить в проект) и кнопка Data Source... (Источник данных). Сначала рассмотрим поддержку. Здесь возможны четыре варианта:

- None (Нет) — не обеспечивается поддержка баз данных (по умолчанию). Не используйте этот переключатель, если есть хоть малейший шанс, что приложению потребуется работа с базой данных — добавление такой поддержки позднее потребует значительных усилий.
- Header files only (Только заголовочные файлы) — AppWizard создает и включает файл заголовка <afxdb.h>, обеспечивая основную поддержку базы данных. Вы можете создавать наборы записей (recordset) и затем

проверять и модифицировать их. При выборе этого переключателя, если в дальнейшем потребуется обеспечить работу с базами данных, основную работу можно будет "поручить" мастеру ClassWizard.

- Database view without file support (Просмотр базы данных без файловой поддержки) — предписывает AppWizard создать основанное на формах приложение, в котором просмотр и модификация записей производится через набор записей. Приложение не поддерживает сериализацию документов, так как большинство приложений баз данных работает на основе записей быстрее, чем на основе файлов.
- Database view with file support (Просмотр базы данных с файловой поддержкой) — предписывает создать основанное на формах приложение, в котором просмотр и модификация записей также производится через набор записей. В данном случае приложение поддерживает сериализацию, что может использоваться, например, для обновления файла настроек пользователя.

Выберите переключатель Database view with file support.

Теперь необходимо указать источник данных, с которым мы будем работать. Для этого нажмите кнопку Data Source..., после чего откроется диалоговое окно Database Options (Параметры базы данных), показанное на рис. 20.4.

Обратите внимание, что здесь можно выбирать не только источник данных, но и тип доступа. Установите переключатель на ODBC и в комбинированном списке выберите источник данных (MS Access 97 Database). Теперь требуется выбрать один из переключателей Recordset type (Тип набора записей). *Набор записей* (Recordset) — это, говоря несколько упрощенно, выборка из таблицы (или запроса) базы данных. Существуют три вида набора записей:

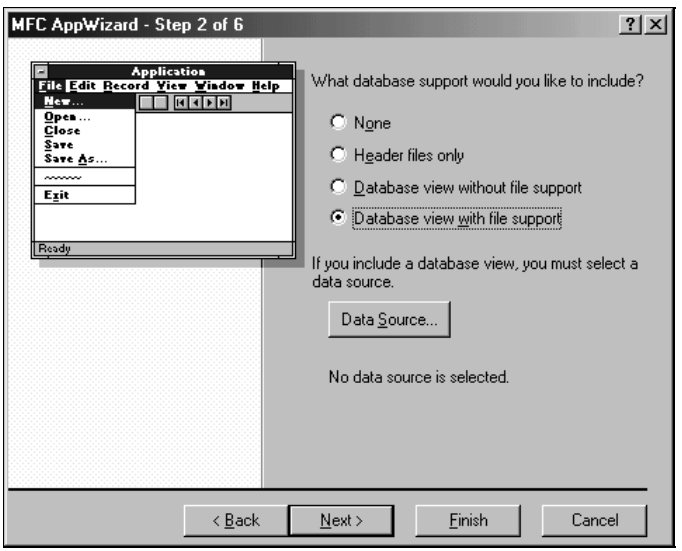



**Рис. 20.3.** Диалоговое окно MFC AppWizard — Step 2 of 6

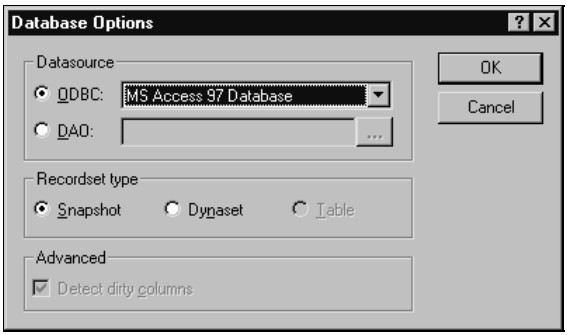



**Рис. 20.4.** Диалоговое окно Database Options

- Snapshot (Моментальный снимок) — *статическая* копия данных, которые содержатся в одной или нескольких таблицах базы данных. При выборе этого переключателя результат выполнения запроса будет загружен в один мгновенный список, т. е. вы получаете полную картину интересующего вас содержимого базы данных и можете использовать ее в качестве основы для последующей работы. Этот тип набора записей используется в основном для поиска и обобщения данных, и у него есть три основных недостатка. Во-первых, вы не увидите обновлений, если они сделаны по сети другими пользователями, а это означает, что ваше решение будет основываться на устаревшей информации. Во-вторых, загрузка всех этих записей одновременно означает, что вы сильно нагружаете сеть. В-третьих, пользователю, скорее всего, надоест ждать, пока записи загрузятся, и он начнет проявлять недовольство по поводу медленной работы сети. Но есть и два преимущества. Во-первых, как только записи загрузятся, активность в сети со стороны рабочей станции будет достаточно мала — сеть освобождается для других запросов, и в общем случае пропускная способность сети улучшится. Во-вторых, так как все запрашиваемые записи находятся теперь на машине пользователя, общая производительность приложения для него также повысится.
- Dynaset (Динамический набор) — *динамическая* копия данных, которые содержатся в одной или нескольких таблицах базы данных. В отличие от моментального снимка, любые внесенные в набор изменения сразу же становятся доступны всем остальным пользователям базы данных. При выборе этого переключателя Visual C++ создает отдельный указатель для каждой запрашиваемой записи. Кроме того, с сервера загружаются только те записи, которые действительно нужны для заполнения формы. Преимущества такого подхода очевидны — записи на экране появляются практически немедленно. Помимо этого, вы также будете в курсе тех изменений, которые другие пользователи осуществляют в базе данных. И, наконец, другие пользователи будут знать об изменениях, которые делаете вы, так как если записи изменились, то они сразу же загружаются на сервер. Совершенно очевидно, что этот подход требует практически постоянного доступа к серверу, что будет снижать общую пропускную

способность сети, а также производительность приложения. Этот переключатель следует выбирать для создания приложений, где пользователь больше всего времени будет тратить на корректировку данных. Это также наилучший выбор для больших баз данных, потому что будет загружаться только та информация, которая действительно нужна пользователю.

- Table (Таблица). Предыдущие две опции работают на уровне записей, при использовании же табличного подхода (доступен только для DAO и нами использоваться не будет) результат выполнения запроса помещается во временную таблицу. Это не только сокращает объем информации, загружаемой с сервера, но и означает, что вы, как программист, сразу можете непосредственно манипулировать полями и записями временной таблицы. Каковы затраты при таком режиме работы? Если невнимательно подойти к формированию запроса, то процесс загрузки с сервера может оказаться более длительным, чем при использовании набора записей. Кроме того, можно не увидеть никаких изменений, сделанных другими пользователями, а они не смогут увидеть изменения, сделанные вами.

Для нашего примера установите переключатель Snapshot и нажмите кнопку OK. После регистрации соединения с базой данных вы увидите блок диалога Select Database Tables (Выберите таблицы базы данных), аналогичный представленному на рис. 20.5, где необходимо выделить те таблицы или запросы (queries), которые требуется включить в набор записей. В нашем простейшем случае это пока одна таблица Products.

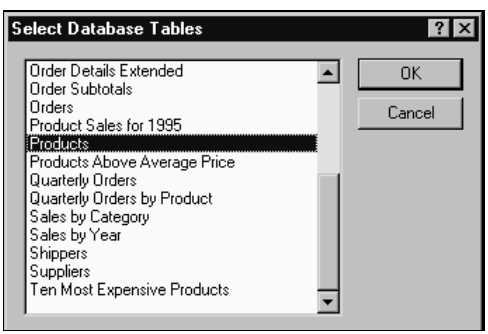



**Рис. 20.5.** Диалоговое окно Select Database Tables

После нажатия на кнопку OK вы вернетесь к диалоговому окну MFC AppWizard — Step 2 of 6, где, правда, изменилась надпись под кнопкой Data Source..., которая теперь сообщает о выбранных таблицах. На данном этапе конфигурирование приложения для использования базы данных Northwind завершается.

Три следующие окна мастера для нас неинтересны, поэтому просто четыре раза нажмите кнопку Next. В результате на экране появится диалоговое окно MFC AppWizard — Step 6 of 6, аналогичное показанному на рис. 20.6.

В отображенном списке выделите элемент *CDBView*, чтобы обратить внимание на его базовый класс — *CRecordView*, с которым мы подробно познако-

мимся чуть ниже. Нажмите кнопку Finish, и Visual C++ отобразит диалоговое окно New Project Information.

**Примечание**

Безусловно, мастер AppWizard выполняет огромную часть работы по созданию приложения. Но наполнять его содержимым в любом случае придется вам самим. Если сейчас скомпилировать (или как рекомендует называть этот процесс фирма Microsoft) построить приложения, то никаких данных из базы данных вы не увидите (рис. 20.7).

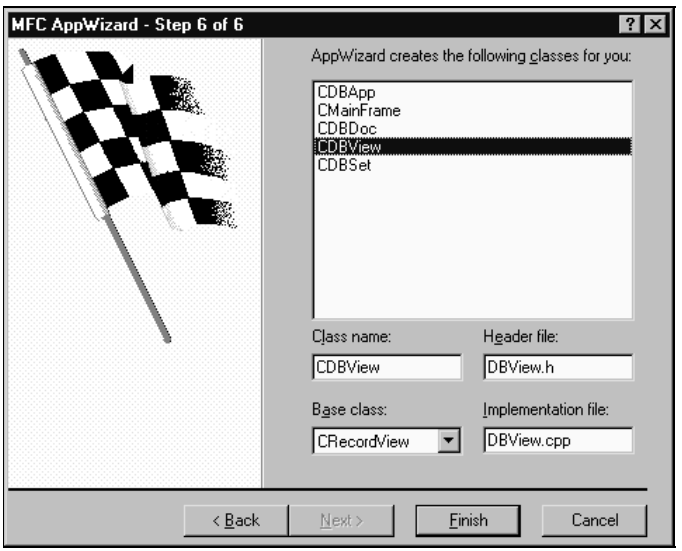



**Рис. 20.6.** Диалоговое окно MFC AppWizard — Step 6 of 6

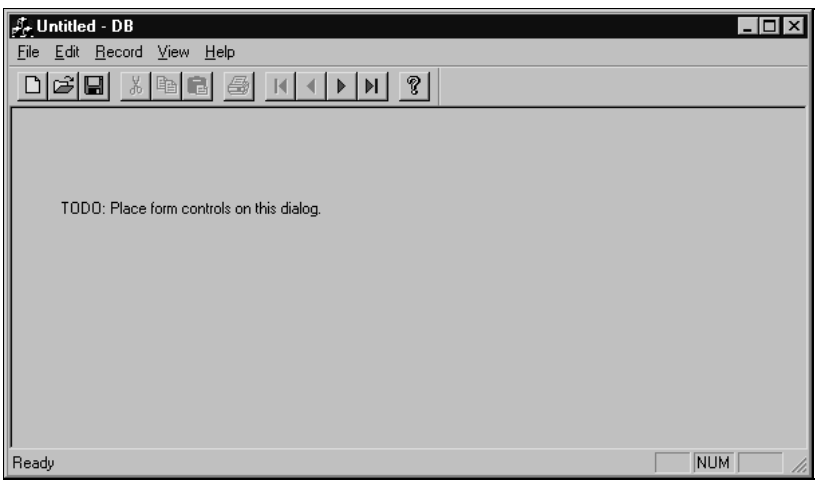



**Рис. 20.7.** Приложение, созданное мастером AppWizard, не предоставляет пользователю никакой информации о содержимом базы данных

А теперь перейдем к непосредственному рассмотрению классов библиотеки MFC, предназначенных для работы с базами данных с использованием механизма ODBC.

# Классы для работы с ODBC

Эти классы, взаимодействуя с другими классами приложения, обеспечивают простой доступ к различным базам данных, использующим драйверы ODBC. Приложениям, которые работают с такими базами данных, 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 соединения с базами данных, посредством которого можно манипулировать источником данных (рис. 20.8). Чтобы у вас сразу же сложилось представление о заложенных в этот класс возможностях, приведу список категорий, на которые можно условно разделить все его компоненты и методы:

- *Данные*. Эти компоненты класса *CDatabase* хранят информацию, используемую в том случае, когда вы хотите работать непосредственно с базой данных, к которой присоединен объект *CDatabase*.
- *Создание соединения*. В эту категорию входят конструктор и методы для открытия/закрытия базы данных.
- *Атрибуты данных*. Сюда относятся десять функций, используемых для получения информации о базе данных, к которой присоединен объект *CDatabase*.
- *Операции* — пять функций, позволяющих обрабатывать транзакции и непосредственно выполнять команды SQL.
- *Переопределяемые методы*. Один метод, позволяющий программисту более конкретно настроить функционирование объекта *CDatabase*.

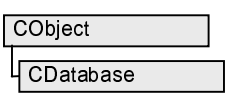



**Рис. 20.8.** Место класса *CDatabase* в иерархии библиотеки MFC

### Создание соединения

`CDatabase::CDatabase ()`

Служит для создания объекта *CDatabase*:

```
class CPublDoc : public CDocument
{
```

```
public:
   // Объявляем объект CDatabase в документе
   CDatabase m_dbPubl;
   ...
};
```

После того как объект создан, необходимо установить 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)
```

Параметр *lpszDSN* определяет имя источника данных, которое должно быть зарегистрировано с помощью программы ODBC Administrator. Это значение должно быть равно NULL, если DSN (Data Source Name, Имя источника данных) определено в строке *lpszConnect*, или может быть равно NULL, если необходимо предоставить пользователю блок диалога для выбора источника данных. Параметр *bExclusive* (в Visual C++ 5.0 не поддерживается): источник данных всегда открывается для совместного использования, и значение параметра должно быть равно FALSE, в противном случае будет выдано сообщение об ошибке. Параметр *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. Параметр *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. Для совместимости с будущими версиями требуется, чтобы эта строка начиналась с подстроки "ODBC", указывающей на то, что соединение устанавливается с источником данных ODBC. Параметр *bUseCursorLib* указывает на необходимость (TRUE) или необязательность загрузки динамической библиотеки ODBC Cursor Library, позволяющей работать с курсорами базы данных.

Упрощенная версия рассмотренной функции имеет вид:

```
virtual BOOL CDatabase::OpenEx (
  LPCTSTR lpszConnectString,
  DWORD dwOptions = 0)
```

Параметр *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. Параметр *dwOptions* — битовая маска, которая определяет комбинацию следующих значений:

| | |
|---|---|
| CDatabase::openExclusive | В Visual C++ до версии 6.0 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, заданное 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.

Как видите, обе функции выполняют одну и ту же задачу — установить соединение с источником данных. Разница заключается только в способе задания параметров.

Чтобы закончить эту тему, рассмотрим несколько примеров:

```
// Создаем объект класса CDatabase
CDatabase m_dbSamp;
// Открываем соединение с источником данных, указав имя источника
   и идентификатор пользователя (без пароля)
m_dbSamp.Open(_T("Samples"), FALSE, FALSE, _T("ODBC;UID=sa");
// или запрашиваем всю информацию у пользователя
m_dbSamp.Open(NULL);
```

К этой же категории относится и функция закрытия соединения после того, как работа с ним закончена:

```
virtual void CDatabase::Close()
```

Эта функция не имеет параметров и не возвращает никакого значения:

```
// Закрываем текущее
m_dbSamp.Close()
// ... и открываем новое соединение
m_dbPubl.OpenEx(_T("DSN=Authors;UID=sa"),
                CDatabase::openReadOnly |
                CDatabase::noOdbcDialog));
```

Но, как вы понимаете, никому не нужно просто открыть и закрыть соединение с источником данных. Поэтому рассмотрим методы остальных перечисленных категорий класса *CDatabase*.

## Атрибуты данных

Входящие в эту категорию функции используются для предоставления информации о соединении, драйвере и источнике данных, а также для установки некоторых опций источника данных. Наиболее часто они применяются в интерфейсных приложениях. Рассмотрим основные из них.

`const CString& CDatabase::GetConnect()`

Вызов этой функции позволяет получить описывающую соединение строку, 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 ()`

Позволяет определить, имеется ли (возвращается ненулевое значение) или нет (0) текущее соединение объекта *CDatabase* с источником данных.

`BOOL CDatabase::CanUpdate ()`

Устанавливает, может ли пользователь обновлять базу данных. Сама возможность обновления 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, а в противном случае 0. Фактор, влияющий на возможность обновления, можно задать, вызвав функцию *::SQLGetInfo* с параметром SQL_DATASOURCE_READ_ONLY.

## Операции

Функции этой категории используются для непосредственной работы с базой данных. К ним относятся функции обработки транзакций (используемые для обновления базы данных) и непосредственного выполнения команд SQL:

`void CDatabase::ExecuteSQL (LPCSTR lpszSQL)`

Позволяет непосредственно выполнить команду SQL, задаваемую в завершающейся 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:

```
CString strCmd = "UPDATE Products SET UnitPrice = 30";
try
{
  m_dbCust.ExecuteSQL(strCmd);
}
catch(CDBException, e)
{
   // Код ошибки находится в e->m_nRetCode
}
```

Другие составляющие класса *CDatabase* мы рассмотрим в следующей главе, непосредственно при разработке нашего приложения. А пока познакомимся с классом, который позволяет работать с наборами записей (рис. 20.9).

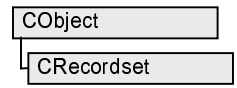



**Рис. 20.9.** Место класса *CRecordset* в иерархии библиотеки MFC

# Класс *CRecordset*

Все компоненты и методы этого класса можно разбить на семь категорий:

- *Компоненты данных*. Служат для хранения информации, используемой для непосредственной работы с базой данных, к которой объект этого класса был присоединен.
- *Конструирование*. В эту категорию входят конструктор и методы для открытия/закрытия форм базы данных.
- *Атрибуты результирующего набора*. Функции, используемые для получения информации о результирующем наборе, к которому присоединен объект класса *CRecordset*.
- *Операции обновления результирующего набора*. Четыре операции, предназначенные для обработки транзакций.
- *Операции перемещения по результирующему набору*. Функции, позволяющие перемещаться по записям результирующего набора.
- *Другие операции над результирующим набором*. Восемь функций, предоставляющие дополнительные функциональные возможности.
- *Переопределяемые методы*. Пять переопределяемых функций, позволяющие программисту настроить функционирование объекта класса *CRecordset*.

## Компоненты данных

В классе *CRecordset* определены следующие основные компоненты данных:

`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 воспользоваться мастером ClassWizard, то это будет сделано автоматически.

`UINT CRecordset::m_nParams`

Содержит число параметров в результирующем наборе — число параметров, посылаемых в параметризированном запросе результирующего набора. Если

эта переменная используется, то она должна быть корректно инициализирована в конструкторе. По умолчанию инициализируется нулем.

`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 же 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 *m_pDatabase*. И хотя обычно нет необходимости отслеживать ее состояние, все же есть случаи, когда делать это необходимо, например, при запуске транзакции или непосредственного выполнения оператора SQL.

`CString CRecordset::m_strFilter`

Используется в качестве фильтра, что позволяет выбирать только записи, удовлетворяющие заданному критерию. Для осуществления "фильтрации" эту переменную следует определять после создания объекта класса *CRecordset*, но до вызова функции *Open*, или же "перечитать" запрос с помощью функции *Requery*.

**Примечание**

Для тех, кто знаком с языком SQL, скажу, что эта строка служит для определения предложения WHERE оператора SQL. Ключевое слово WHERE не нужно включать в строку фильтра, поскольку библиотека подставляет его сама. Те же читатели, кто еще не познакомился с "языком запросов" к базе данных SQL, должны обязательно это сделать. Думаю, что причины объяснять не надо, и поэтому только порекомендую свою книгу "Microsoft SQL Server 7.0", в которой приведено подробное и простое описание этого языка.

`CString CRecordset::m_strSort`

Используется в качестве фильтра, позволяющего сортировать записи, удовлетворяющие заданному критерию. Для того чтобы осуществить "сортировку", эту переменную следует определять после создания объекта класса *CRecordset*, но до вызова функции *Open*, или же "перечитать" запрос с помощью функции *Requery*.

**Примечание**

Эта строка служит для определения предложения ORDER BY оператора SQL. Не нужно включать в строку фильтра ключевое слово ORDER BY, поскольку библиотека подставляет его сама.

Помимо перечисления полей, по которым производится сортировка, можно указать так же одно из ключевых слов (отдельно для каждого поля), 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 производного от *CRecordset* класса 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)
```

При успешном выполнении функции возвращается ненулевое значение и 0 — в противном случае. Если полученный результирующий набор не пустой, то текущей является первая запись. Параметр *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, отображаются не 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.

Параметр *lpszSQL* — указатель на строку, содержащую одно из значений: NULL, имя таблицы, оператор SQL, не обязательно с предложениями WHERE или ORDER BY, или оператор CALL, определяющий имя предопределенного запроса или сохраненной процедуры.

**Примечание**

Порядок столбцов в результирующем наборе должен совпадать с порядком вызова в переопределенной функции *DoFieldExchange*.

Параметр *dwOptions* — битовая маска, которая определяет комбинацию перечисленных ниже 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 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* |

Функцию *CRecordset::Open* следует вызывать для выполнения запроса, определяющего результирующий набор. Очевидно, что перед ее вызовом объект класса *CRecordset* должен быть создан. 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 функции *Open* 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* | "Products" |
| Имя предопределенного запроса (сохраненной процедуры) | Столбцами, определенными в запросе | "(call ProductList)" |
| SELECT список-столбцов FROM список-таблиц | Заданными столбцами из определенных таблиц | SELECT ProductName, UnitPrice FROM Products |

**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 библио-

тека 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.

Перед вызовом функции *Open* можно определить дополнительные условия, соответствующие параметрам предложений WHERE и ORDER BY оператора SQL, используя для этого описанные выше переменные класса *m_strFilter* и *m_strSort*. Если же их определить после того, как результирующий набор открыт, то они не окажут никакого влияния на результат выполнения запроса. В этом случае необходимо будет воспользоваться функцией *Requery* для обновления записей.

`virtual void CRecordset::`**`Close`** `()`

Используется для закрытия результирующего набора. Если он не был открыт, то функция просто завершается. После ее выполнения вся память, выделенная для результирующего набора, и дескриптор HSTMT ODBC, ассоциированный с ним, возвращается 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 функции, работающие непосредственно с атрибутами *CRecordset* и позволяющие получить информацию о результирующем наборе.

`BOOL CRecordset::`**`CanAppend`** `()`

Вызов этой функции 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`** `()`

Вызов этой функции 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*, и определяется только тем, поддерживают ли закладки используемый драйвер ODBC.

**Примечание**

Закладки не поддерживаются в результирующих наборах с возможностью просмотра "только вперед".

`BOOL CRecordset::`**`CanRestart`** `()`

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::CanUpdate ()`

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*.

`long CRecordset::GetRecordCount ()`

Позволяет определить размер результирующего набора. Она возвращает или число записей в результирующем наборе, или 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. 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.

`void CRecordset::GetStatus(CRecordsetStatus& rStatus) const`

Позволяет определить индекс текущей записи в результирующем наборе и/или найдена ли последняя запись. В качестве параметра функция принимает ссылку на структуру *CRecordsetStatus*

```
struct CRecordsetStatus{
  long m_lCurrentRecord;
  BOOL m_bRecordCountFinal;
}
```

Поля этой структуры имеют следующий смысл:

- *m_lCurrentRecord* содержит индекс (начиная с 0) текущей записи в результирующем наборе. Если индекс не может быть определен, то в это поле записывается значение AFX_CURRENT_RECORD_UNDEFINED (–2). Если результирующий набор пустой или была выполнена попытка переместиться до первой записи, то в поле записывается AFX_CURRENT_RECORD_BOF (–1).
- *m_bRecordCountFinal* содержит ненулевое значение, если было определено общее число записей в результирующем наборе, 0 — если он не содержит записей, и –1, если число записей не может быть определено.

`const CString& CRecordset::GetTableName ()`

Позволяет получить имя таблицы, на которой основывается запрос результирующего набора, или пустую строку. Эта функция работает корректно только в том 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 ()`

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*, тем, 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 ()`

Позволяет определить, открыт ли уже результирующий набор. Хорошим стилем программирования является использование этой функции перед тем, как вызывать функцию *Open*.

`BOOL CRecordset::IsBOF ()`

Позволяет определить, является ли текущая запись первой в наборе данных. Она возвращает ненулевое значение, если результирующий набор не содержит записей или указатель помещен до первой записи, и 0 — в противном случае. Если функция возвратила ненулевое значение, то текущая запись не определена, и при вызове функции *MovePrev* возникает ошибка.

`BOOL CRecordset::IsEOF ()`

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 функция возвратила ненулевое значение, то текущая запись не определена, и при вызове функции *MoveNext* возникает ошибка.

`BOOL CRecordset::IsDeleted ()`

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.

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 и каково 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 ()`

Используется для подготовки новой записи к добавлению в результирующий набор. Все поля записи исходно устанавливаются в Null (в терминологии баз данных это означает " значение не задано" 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 будет потеряна без какого-либо предупреждения.

Для динамического набора (dynaset) новая запись автоматически добавляется в конец результирующего набора. Чтобы увидеть эту новую запись в "мгновенном снимке", необходимо вызвать функцию *Requery*.

При попытке вызвать функцию *Update* 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 тех случаях, когда требуется записать в некоторый столбец значение Null (не содержит данных), необходимо вызвать функцию *SetFieldNull* с параметром TRUE. При необходимости оставить без изменений какое-либо поле вызовите функцию *SetFieldDirty* с параметром TRUE, даже 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, осуществленное операциями *AddNew* или *Edit*, до вызова функции *Update*. Если установлен режим автоматической отметки поля как "грязного", 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 при вызове функции *Open* необходимо установить параметр *dwOptions* в значение *CRecordset::noDirtyFieldCheck*).

**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 следующие: *Move*, *MoveFirst*, *MoveLast*, *MoveNext* и *MovePrev*. Их мы и рассмотрим.

```
virtual void CRecordset::Move (
  long nRows,
  WORD wFetchType = SQL_FETCH_RELATIVE)
```

Функция имеет два параметра: *nRows* — количество строк, на которое необходимо переместиться вперед (положительное значение) или назад (отрицательное) и *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 |

*(окончание)*

| wFetchType | Выбираемый набор строк |
|---|---|
| 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 условие начало файла (BOF) |
| 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, то возникает исключение.

Чтобы как следует разобраться с работой функции *Move*, рассмотрим фрагмент кода.

```
// rs — объект класса CRecordset или производного от него
// Устанавливаем размер набора строк в значение 5
rs.SetRowsetSize(5);
// Переходим к первой записи в результирующем наборе
rs.MoveFirst();
// Переходим к шестой записи
rs.Move(5);
// К ней же можно было перейти и другим способом
// rs.Move(6, SQL_FETCH_ABSOLUTE);
// rs.SetAbsolutePosition(6);
// После установки абсолютной позиции на шестую запись
// она становится первой в следующем наборе строк;
// поэтому следующие вызовы эквивалентны
// rs.Move(1, SQL_FETCH_NEXT);
// rs.MoveNext();
```

**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 использованием данной функции рекомендуется вызвать функцию *IsBOF*.

`void CRecordset::`**`MoveLast`** `()`

Делает текущей последнюю запись результирующего набора. Перед использованием данной функции рекомендуется вызвать функцию *IsEOF*.

`void CRecordset::`**`MoveNext`** `()`

Делает текущей первую запись следующего набора строк. Перед использованием рекомендуется вызвать функцию *IsBOF*. Если работа идет не с блоками, а с отдельными записями, то текущей становится следующая запись.

`void CRecordset::`**`MovePrev`** `()`

Делает текущей первую запись предыдущего набора строк. Перед ее использованием рекомендуется вызвать функцию *IsEOF*. При работе не с блоками, а с отдельными строками текущей становится предыдущая запись.

Из оставшихся трех функций этой категории, использующихся не так часто, однако тоже весьма полезных, рассмотрим две:

`void CRecordset::`**`GetBookmark`** `(CDBVariant& varBookmark)`

Позволяет получить значение закладки текущей записи, которое записывается по ссылке в передаваемый ей параметр.

`void CRecordset::`**`SetBookmark`** `(const CDBVariant& varBookmark)`

Позволяет перейти к записи, содержащей определенную (*varBookmark*) закладку. Чтобы получить закладку для текущей записи, необходимо вызвать функцию *GetBookmark*, после чего можно перейти непосредственно к этой записи, вызвав функцию *SetBookmark* с сохраненным значением закладки.

Если закладки не поддерживаются, то вызов этих двух функций приведет к возникновению исключения. Если же они поддерживаются, то, чтобы сделать их доступными, необходимо при вызове функции *Open* задать значение *CRecordset::useBookmarks* для параметра *dwOptions*.

**Примечание**

После определенных операций с результирующим набором необходимо проверить сохранность закладки перед тем, как вызывать функцию *SetBookmark*. Например, если после получения закладки с помощью функции *GetBookmark* вы вызвали функцию *Requery*, то закладка может быть утеряна. Поэтому перед использованием закладки необходимо вызвать функцию *CDatabase::GetBookmarkPersistence*, чтобы проверить ее сохранность.

## Другие операции над результирующим набором

В эту категорию входят функции, которые трудно отнести к какому-то одному типу.

`void CRecordset::`**`Cancel`** `()`

Вызов этой функции является просьбой к источнику данных отменить либо обрабатываемую асинхронную операцию, либо второй поток. Библиотека MFC не поддерживает асинхронные операции, и для них необходимо непосредственно вызвать функцию SQLSet*ConnectOption* из ODBC API.

`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.

```
virtual BOOL CRecordset::Requery ()
```

Позволяет обновить результирующий набор. Функцию 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 значение FALSE — ошибка, то результирующий набор закрывается.

**Примечание**

Функцию *Requery* можно вызывать только для открытого результирующего набора. Проверить это можно с помощью функции *CanRestart*.

```
void CRecordset::SetFieldDirty (
  void* pv,
  BOOL bDirty = TRUE)
```

Позволяет пометить поле данных результирующего набора (параметр *pv*) как измененное (*bDirty* = TRUE) или не изменившееся (*bDirty* = FALSE). Если в качестве *pv* передается NULL, помечаются все поля. Использование этой функции позволяет снизить трафик SQL, если изменились не все поля записи.

```
void CRecordset::SetFieldNull (
  void* pv,
  BOOL bNull = TRUE)
```

Позволяет пометить поле данных результирующего набора (параметр *pv*) как содержащие (*bNull* = TRUE) или не содержащие (*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::SetParamNull (
  int nIndex,
  BOOL bNull = TRUE)
```

Позволяет установить параметр, заданный своим индексом *nIndex*, в значение Null (параметр *bNull* равен TRUE), или не Null (*bNull* = FALSE). Обычно эта функция используется в предопределенных запросах (сохраненных процедурах).

## Переопределяемые методы

В эту категорию входят восемь функций, которые можно переопределить в каждом конкретном приложении, чтобы настроить его на выполнение текущих задач. Мы рассмотрим только некоторые из них.

```
virtual void CRecordset::DoFieldExchange (CFieldExchange* pFX)
```

Вызывается для организации обмена данными между полями результирующего набора и соответствующими столбцами текущей записи в источнике данных.

В качестве параметра функция принимает указатель на объект *CFieldExchange*, который автоматически создается и передается библиотекой MFC.

**Примечание**

Эти функция доступна только в том случае, если для результирующего набора используется класс, производный от *CRecordset*. Если же применяется непосредственно класс *CRecordset*, то для получения данных необходимо вызывать функцию *GetFieldValue*.

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. Ниже приведен фрагмент, демонстрирующий переопределение функции *DoFieldExchange*.

```
void CDBSet::DoFieldExchange(CFieldExchange* pFX)
{
  //{{AFX_FIELD_MAP(CDBSet)
    pFX->SetFieldType(CFieldExchange::outputColumn);
    RFX_Text(pFX, _T("[ProductName]"), m_ProductName);
    RFX_Long(pFX, _T("[CategoryID]"), m_CategoryID);
    RFX_Text(pFX, _T("[QuantityPerUnit]"), m_QuantityPerUnit);
    RFX_Text(pFX, _T("[UnitPrice]"), m_UnitPrice);
  //}}AFX_FIELD_MAP
}
```

**Примечание**

Мастер ClassWizard не поддерживает обмен полями блочных записей, и необходимо переопределить соответствующую функцию вручную, реализовав в ней вызовы функций *Bulk RFX* (блочный обмен полями записи). Функции для собственно обмена данными мы рассмотрим при обсуждении класса *CFieldExchange*.

`virtual CString CRecordset::`**`GetDefaultConnect`** `()`

Библиотека MFC вызывает данную функцию, чтобы получить строку, содержащую источник данных, на котором базируется результирующий набор. Ниже приведен фрагмент кода, подробно рассматриваемый в следующей главе и иллюстрирующий использование этой функции.

```
CString CDBSet::GetDefaultConnect()
{
  return _T("ODBC;DSN=MS Access 97 Database");
}
```

`virtual CString CRecordset::`**`GetDefaultSQL`** `()`

Библиотека 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. Ниже приведен фрагмент кода, подробно рассматриваемый в следующей главе, который иллюстрирует использование этой функции

```
CString CDBSet::GetDefaultSQL()
{
  return _T("[Products]");
}
```

Если библиотека MFC не сможет найти имя таблицы или корректно интерпретировать оператор CALL, функция возвращает пустую строку.

**Примечание**

Между круглой скобкой и ключевым словом, например, CALL или SELECT, не должно быть пробелов.

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 сделают эту работу достаточно простой (рис. 20.10).

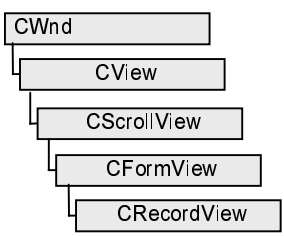



**Рис. 20.10.** Классы представлений библиотеки MFC

## Класс *CRecordView*

Объекты этого класса предоставляют для изображения записей базы данных в элементах управления форму, которая непосредственно соединена с объ-

ектом *CRecordset*. Объекты *CRecordView* используют механизм DDX (Dialog Data Exchange, Обмен данными с блоком диалога) и RFX (Record Field Exchange, Обмен полями записей) для автоматического перемещения данных между элементами управления формы и полями результирующего набора. Кроме того, можно воспользоваться реализованными возможностями перемещения по записям и обновления текущей записи.

Все компоненты и методы этого класса можно условно разбить на три категории.

- *Создание объекта*. Имеется только конструктор для создания объекта.
- *Атрибуты данных*. Три функции, используемые для получения информации о результирующем наборе, к которому присоединена форма.
- *Операции*. Единственная функция, позволяющая программисту изменить указатель на текущую запись.

Ниже приведено описание каждой из этих категорий.

## Создание объекта

В эту категорию входит только конструктор, имеющий две реализации.

`CRecordView::`**`CRecordView`**`(LPCSTR lpszTemplateName)`

`CRecordView::`**`CRecordView`**`(UINT nIDTemplate)`

Создает объект класса. В качестве параметра конструктор принимает идентификатор шаблона блока диалога, задаваемый либо строкой (*lpszTemplateName*), либо номером (*nIDTemplate*). При создании класса, производного от *CRecordView*, в нем можно определить только один конструктор, в котором необходимо вызвать конструктор базового класса *CRecordView::CRecordView* с идентификатором ресурса в качестве параметра, как это показано в приведенном ниже фрагменте:

```
CDBView::CDBView() : CRecordView(CDBView::IDD)
{
  //{{AFX_DATA_INIT(CDBView)
    m_pSet = NULL;
    m_strPrice = _T("");
  //}}AFX_DATA_INIT
  m_bAdd  = FALSE;
  m_nSort = ID_SORT_TITLE;
}
```

## Атрибуты данных

Функции этой категории позволяют получить информацию о представлении записи.

`virtual CRecordset* CRecordView::`**`OnGetRecordset`**` ()`

Возвращает указатель на объект *CRecordset*, ассоциированный с формой, позволяя тем самым работать с некоторым результирующим набором. Чисто вир-

туальная функция, которая требует обязательного переопределения. Причина этого понятна — библиотека не может знать, с каким результирующим набором, т. е. объектом класса *CRecordset*, вы работаете.

Ниже показан фрагмент кода, который выдает мастер AppWizard при создании приложения для работы с базой данных:

```
CRecordset* CDBView::OnGetRecordset()
{
  return m_pSet;
}
```

`BOOL CRecordView::`**`IsOnFirstRecord`** `()`

Позволяет определить, является ли текущая запись первой в результирующем наборе, ассоциированном с данной формой. Когда пользователь перемещается за первую запись, библиотека блокирует доступ к элементам пользовательского интерфейса для перемещения на первую и предыдущую запись. Реализация этой функции по умолчанию представлена в приведенном ниже фрагменте:

```
BOOL CRecordView::IsOnFirstRecord()
{
  ASSERT_VALID(this);
  CRecordsetStatus status;
  OnGetRecordset()->GetStatus(status);
  return status.m_lCurrentRecord == 0;
}
```

`BOOL CRecordView::`**`IsOnLastRecord`**`()`

Позволяет определить, является ли текущая запись последней в результирующем наборе, ассоциированном с данной формой. Когда пользователь перемещается за последнюю запись, библиотека блокирует доступ к элементам пользовательского интерфейса для перемещения на последнюю и следующую запись.

Реализация этой функции по умолчанию представлена в приведенном ниже фрагменте:

```
BOOL CRecordView::IsOnLastRecord()
{
  ASSERT_VALID(this);
  CRecordset* pRecordset = OnGetRecordset();
  CRecordsetStatus status;
  pRecordset->GetStatus(status);
  if (!status.m_bRecordCountFinal)
    return FALSE;
  return((status.m_lCurrentRecord+1==pRecordset->GetRecordCount()));
}
```

## Операции

В этой категории имеется единственная функция, позволяющая программисту перемещаться по записям результирующего набора.

`virtual BOOL CRecordView::OnMove(UINT nIDMoveCommand)`

Позволяет изменять указатель на текущую запись, или, другими словами, перемещаться по записям результирующего набора и отображать его поля в элементах управления формы. Параметр *nIDMoveCommand* задает направление перемещения и может принимать следующие значения:

| | |
|---|---|
| ID_RECORD_FIRST | Переход к первой строке в результирующем наборе |
| ID_RECORD_LAST | Переход к последней строке в результирующем наборе |
| ID_RECORD_NEXT | Переход к следующей строке в результирующем наборе |
| ID_RECORD_PREV | Переход к предыдущей строке в результирующем наборе |

Реализация этой функции по умолчанию обновляет текущую запись источника данных, если пользователь изменил ее в форме.

**Примечание**

Если результирующий набор не имеет записей, то вызов функции *OnMove* приводит к исключению. Поэтому перед ее использованием необходимо определить, имеются ли записи в результирующем наборе.

Как видите, функций совсем немного, но не следует забывать, что класс *CRecordView* базируется на многих других классах (рис. 20.10), откуда и наследует все их возможности.

## Класс *CFieldExchange*

Как видно из рис. 20.11, класс *CFieldExchange* не имеет базового.




**Рис. 20.11.** Место класса *CFieldExchange* в библиотеке MFC

Он используется для обмена данными между записями результирующего набора и переменными приложения, которые хранят столбцы данных. Для решения этой задачи класс реализует механизмы обмена полями записи RFX (Record Field eXchange) и обмена блоками полей записи Bulk RFX (Bulk Record Field eXchange).

Класс содержит всего две функции, которые мы и рассмотрим.

`BOOL CFieldExchange::IsFieldType(UINT* pnField)`

Позволяет определить, может ли текущая операция быть выполнена для текущего поля. Имеет один параметр — *pnField*, указатель на индекс поля, и возвращает ненулевое значение, если операция может быть выполнена.

Непосредственно в приложениях вызывать эту функцию практически нет необходимости, если пользоваться функциями RFX, где она применяется. Исключение составляет случай, когда вам необходимо написать некоторую свою функцию RFX. Ниже приведен фрагмент кода из файла <dbrfx.cpp>, демонстрирующий использование функции *IsFieldType*.

```
void AFXAPI RFX_Int(CFieldExchange* pFX, LPCTSTR szName, int& value)
{
  ASSERT(AfxIsValidAddress(pFX, sizeof(CFieldExchange)));
  ASSERT(AfxIsValidString(szName));
  UINT nField;
  if (!pFX->IsFieldType(&nField))
    return;
  LONG* plLength = pFX->m_prs->GetFieldLengthBuffer(
                                    nField - 1, pFX->m_nFieldType);
  switch (pFX->m_nOperation)
  {
    case CFieldExchange::BindFieldToColumn:
    {
    #ifdef _DEBUG
      // Принять все связанные поля ДО несвязанных полей
      CODBCFieldInfo* pODBCInfo =
                  &pFX->m_prs->m_rgODBCFieldInfos[nField - 1];
      if (pODBCInfo->m_nSQLType != SQL_C_SHORT)
      {
        // Предупредить о возможном несовпадении схемы полей
         if (afxTraceFlags & traceDatabase)
           TRACE1("Warning: int converted from SQL type %ld.\n",
                   pODBCInfo->m_nSQLType);
      }
    #endif // _DEBUG
    }
    default:
    LDefault:
        pFX->Default(szName, &value, plLength, SQL_C_LONG,
                     sizeof(value), 5);
        return;
    case CFieldExchange::Fixup:
      if (*plLength == SQL_NULL_DATA)
      {
        pFX->m_prs->SetNullFieldStatus(nField - 1);
        value = AFX_RFX_INT_PSEUDO_NULL;
      }
      return;
    case CFieldExchange::SetFieldNull:
      if ((pFX->m_pvField == NULL &&
           pFX->m_nFieldType == CFieldExchange::outputColumn) ||
           pFX->m_pvField == &value)
      {
        if (pFX->m_bField)
        {
          // Помечаем поля, как допускающие значение Null
          pFX->m_prs->SetNullFieldStatus(nField - 1);
```

```
        value = AFX_RFX_INT_PSEUDO_NULL;
        *plLength = SQL_NULL_DATA;
      } else
      {
        pFX->m_prs->ClearNullFieldStatus(nField - 1);
        *plLength = sizeof(value);
      }
    #ifdef _DEBUG
      pFX->m_nFieldFound = nField;
    #endif
    }
    return;
  case CFieldExchange::MarkForAddNew:
  // Можно принудительно записать псевдонулевое значение
  // при маркировке поля как изменившееся ("грязное")
    if (value != AFX_RFX_INT_PSEUDO_NULL)
    {
      pFX->m_prs->SetDirtyFieldStatus(nField - 1);
      pFX->m_prs->ClearNullFieldStatus(nField - 1);
    }
    return;
  case CFieldExchange::MarkForUpdate:
    if (value != AFX_RFX_INT_PSEUDO_NULL)
      pFX->m_prs->ClearNullFieldStatus(nField - 1);
    goto LDefault;
  case CFieldExchange::AllocCache:
  {
    CFieldInfo* pInfo = &pFX->m_prs->m_rgFieldInfos[nField - 1];
    // Данные кэшированы значением,
    // нет необходимости в распределении  памяти
    pInfo->m_nDataType = AFX_RFX_INT;
  }
  return;
  #ifdef _DEBUG
    case CFieldExchange::DumpField:
      *pFX->m_pdcDump << "\n" << szName << " = " << value;
      return;
  #endif // _DEBUG
 }
}
```

`void CFieldExchange::`**`SetFieldType`**`(UINT nFieldType)`

Используется для установки типа поля перед тем, как вызывать функцию RFX. Она имеет один параметр — *nFieldType*, типа enum, который может принимать одно из следующих значений:

| | |
|---|---|
| CFieldExchange::outputColumn | Поле является выходным столбцом |
| CFieldExchange::inputParam | Входной параметр — значение, посылаемое в запрос или сохраненную процедуру результирующего набора |
| CFieldExchange::param | То же самое, что и CFieldExchange::inputParam |
| CFieldExchange::outputParam | Выходной параметр — значение, которое возвращается сохраненной процедурой результирующего набора |
| CFieldExchange::inoutParam | Входной/выходной параметр — значение, которое посылается и возвращается из сохраненной процедуры результирующего набора |

Функцию *SetFieldType* необходимо вызывать в переопределенной функции класса результирующего набора. Для полей данных ее следует вызвать с параметром *CFieldExchange::outputColumn*, а затем уже реализовать вызовы функций RFX или Bulk RFX. Ниже приведен фрагмент, иллюстрирующий работу класса *CFieldExchange*:

```
void CSections::DoFieldExchange(CFieldExchange* pFX)
{
   //{{AFX_FIELD_MAP(CSections)
     pFX->SetFieldType(CFieldExchange::outputColumn);
     RFX_Text(pFX, "CourseID", m_strCourseID);
     RFX_Text(pFX, "InstructorID", m_strInstructorID);
     RFX_Text(pFX, "RoomNo", m_strRoomNo);
     RFX_Text(pFX, "Schedule", m_strSchedule);
   //}}AFX_FIELD_MAP
   // выходной параметр
   pFX->SetFieldType(CFieldExchange::outputParam);
   RFX_Long(pFX, "Instructor_Count", m_nCountParam);
   // входной параметр
   pFX->SetFieldType(CFieldExchange::inputParam);
   RFX_Text(pFX, "Department_Name", m_strNameParam);
}
```

Если среди имеющихся функций нет тех, которые поддерживают необходимый вам тип данных, можно создать такую функцию самостоятельно, что совсем несложно, если в качестве основы воспользоваться существующими (приведенными в файле <dbrfx.cpp>). Пример одной из этих функций приведен выше.

На этом мы закончим знакомство с возможностями, предоставляемыми библиотекой MFC для создания приложений, работающих с базами данных. Практические примеры использования приведенного материала рассматриваются в следующей главе.

ГЛАВА 21



# Настройка приложения на работу с базами данных

Старость наступает тогда,
когда вам уже ничего не хочется пробовать.
*Борис Васильев*

В главе 20 мы уже создали каркас простейшего приложения для того, чтобы можно было манипулировать информацией из базы данных Northwind, а также получили необходимую информацию о классах библиотеки MFC, предназначенных для доступа к базам данных на основе стандарта ODBC. Теперь, имея все это под рукой, можно приступать к добавлению необходимого кода для того, чтобы извлекать, обновлять, добавлять, удалять и отображать для пользователя информацию из базы данных. К этому мы и переходим.

## Извлечение информации из базы данных

Прежде чем рассматривать вопрос "Каким образом можно извлечь информацию из базы данных?", необходимо выполнить некоторые подготовительные действия, чтобы можно было видеть на экране запрашиваемую информацию. Поэтому сначала познакомимся с методами отображения данных.

### Подготовка формы для отображения данных

В нашем простейшем примере мы должны научиться манипулировать данными одной единственной таблицы — Products. Используемые нами столбцы этой таблицы перечислены в табл. 21.1.

***Таблица 21.1.*** *Используемые столбцы таблицы Products*

| Столбец | Назначение |
|---|---|
| ProductID | Числовой уникальный идентификатор, автоматически присваиваемый новому продукту (тип Autonumber) |
| ProductName | Текстовая строка, содержащая название продукта |
| QuantityPerUnit | Текстовая строка, характеризующая содержимое партии продукта |
| UnitPrice | Цена партии; денежный тип данных |

Теперь нам необходимо подготовить форму, которая будет отображать требуемую информацию. Примерный вид такой формы представлен на рис. 21.1.

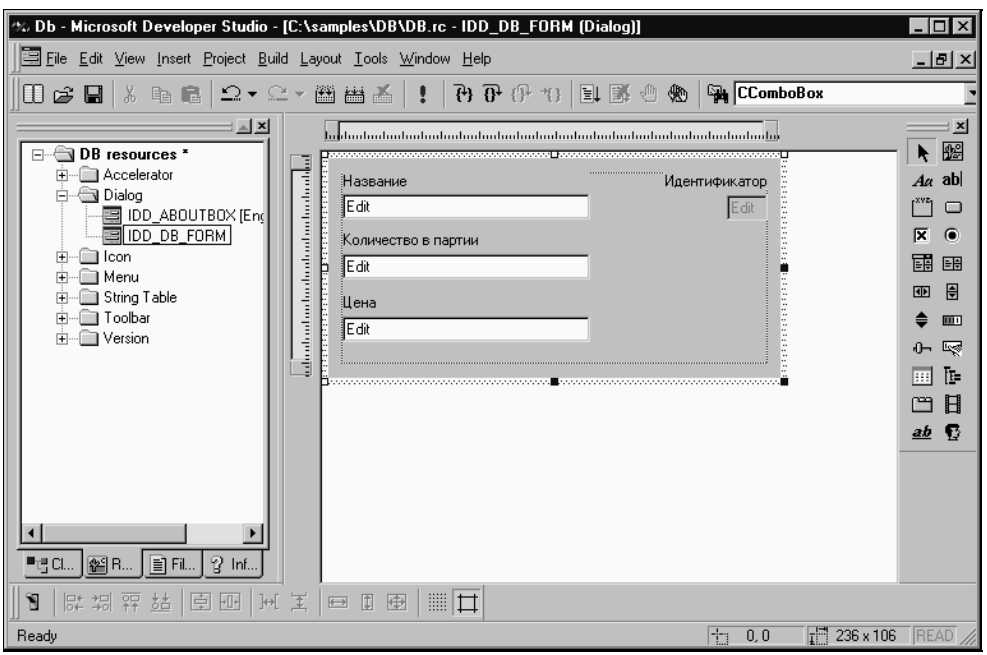



**Рис. 21.1.** Нужно создать форму для отображения информации о продуктах

**Примечание**

Если вы хотите, как и я, использовать русские названия для полей (столбцов) таблицы, то перед тем, как набирать эти названия, необходимо изменить основной язык для окна диалога. Сделать это очень просто — щелкните правой кнопкой мыши на идентификаторе окна диалога IDD_DB_FORM (на рис. 21.1 он обведен рамкой) и в появившемся контекстном меню выберите команду Prop-

erties (Свойства), в результате чего на экране появится диалоговое окно Dialog Properties (Свойства диалога) (рис. 21.2). В комбинированном списке Language (Язык) выберите Russian.

Если теперь скомпилировать и запустить приложение, то на экране появится окно, аналогичное представленному на рис. 21.3.

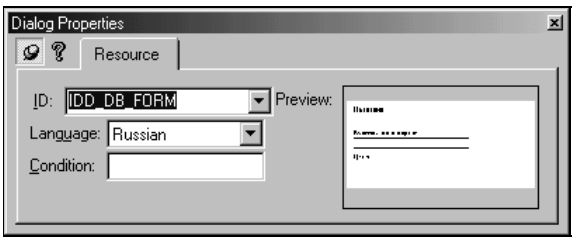



**Рис. 21.2.** Диалоговое окно Dialog Properties

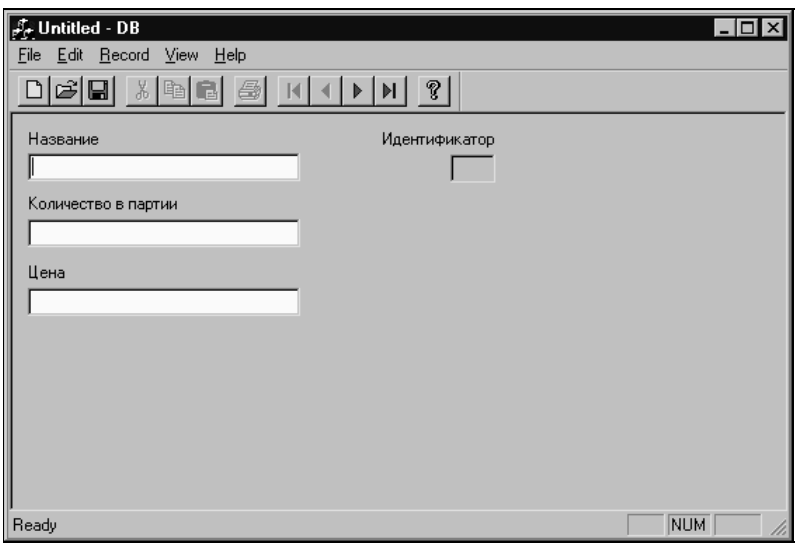



**Рис. 21.3.** Приложение DB отображает форму с пустыми полями

Здесь вы должны обратить внимание на две проблемы: первая — внешний вид формы оставляет желать лучшего и вторая — несмотря на то, что мы присвоили всем элементам управления формы соответствующие идентификаторы и знаем, что в таблице базы данных есть информация, она не отображается в окне.

И если в этот момент вы подумали, что первая проблема не имеет никакого отношения к рассматриваемому вопросу, то вы глубоко заблуждаетесь: даже если ваше приложение будет работать безукоризненно, мало кто, кроме вас, захочет им воспользоваться — дизайн пользовательского интерфейса имеет одно из основных значений при разработке любого программного продукта.

Пока я писал, а вы читали эти строки, можно было бы уже придать окну более привлекательный вид, настолько это просто: необходимо только немного модифицировать функцию *CMainFrame::PreCreateWindow*. Вот как она должна выглядеть (здесь и далее полужирным шрифтом выделен код, добавленный мною):

```
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
  // Изменяем размеры главного окна
  cs.cx = 376;
  cs.cy = 272;
  return CFrameWnd::PreCreateWindow(cs);
}
```

**Примечание**

Ваши значения могут отличаться от приведенных здесь, поскольку они зависят от размеров окна диалога. У меня они равны 242×106, и в качестве шрифта установлен MS Sans Serif, 8, что, как вы помните, имеет существенное значение.

В результате этих минимальных добавлений внешний вид окна формы существенно улучшился (рис. 21.4).

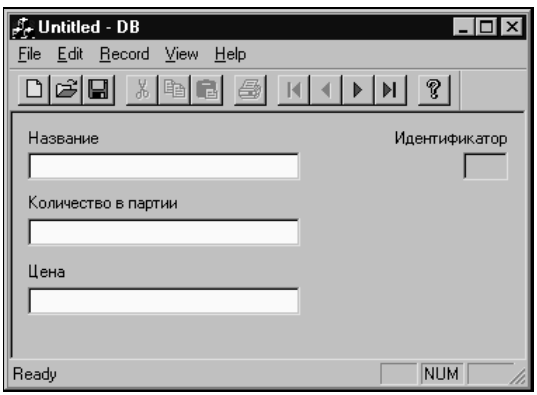



**Рис. 21.4.** Теперь форма выглядит значительно лучше

После усовершенствования внешнего вида формы переходим непосредственно к работе с базой данных.

## Отображение и обновление содержимого базы данных

Для того чтобы отобразить данные из таблицы базы данных, необходимо выполнить определенные действия, о которых вы узнали, читая главу 20. Перечислю их здесь еще раз в сжатой форме. Для получения информации из таблицы базы данных требуется:

- создать соединение с источником данных;
- создать объект, куда будут записываться данные из базы данных;
- сформировать и выполнить запрос;
- отобразить полученные результаты в предварительно созданной форме.

Мы пока ничего из этого не сделали. Правда, очень и очень многое за нас выполнила библиотека MFC, но это не избавляет нас от необходимости четко понимать все этапы процедуры.

Посмотрите еще раз на рис. 21.4, а лучше запустите приложение DB, обратив внимание на то, что на панели инструментов кнопки перемещения на следующую и последнюю записи доступны, а на предыдущую и первую — нет. Попробуйте понажимать на них. Как видите, они реагируют на нажатия, хотя в полях ничего и не отражается. О чем это говорит? О том, что библиотека MFC сама реализовала три первых этапа процесса получения информации, воспользовавшись для этого теми параметрами, которые мы указали при создании проекта.

```
CString CDBSet::GetDefaultConnect()
{
  // Параметры источника данных
  return _T("ODBC;DSN=MS Access 97 Database");
}
CString CDBSet::GetDefaultSQL()
{
  // Параметры таблицы
  return _T("[Products]");
}
class CDBSet : public CRecordset
{
public:
...
  // Переменные для отображения данных строки таблицы
  //{{AFX_FIELD(CDBSet, CRecordset)
    long     m_ProductID;
    CString  m_ProductName;
    long     m_CategoryID;
    CString  m_QuantityPerUnit;
    CString  m_UnitPrice;
  //}}AFX_FIELD
...
}
```

На нашу долю осталась реализация только последнего, четвертого, этапа — отображение результатов. Сделать это совсем просто — нужно только соз-

дать соответствующие переменные. Раскройте спроектированное ранее диалоговое окно IDD_DB_FORM и при нажатой клавише <Ctrl> дважды щелкните на элементе управления. В результате на экране появится диалоговое окно Add Member Variable (Добавить компонентную переменную), показанное на рис. 21.5. В комбинированном списке Member variable name (Имя компонентной переменной) этого окна необходимо выбрать соответствующую переменную — в данном случае m_pSet->m_ProductName.

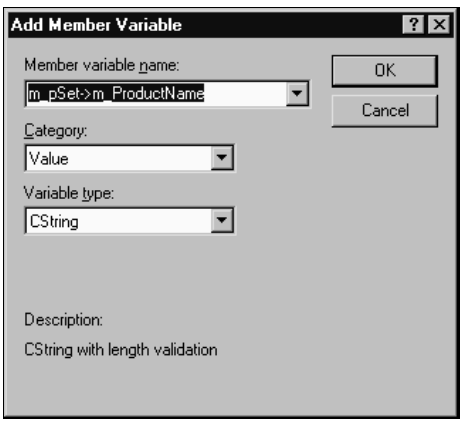



**Рис. 21.5.** Диалоговое окно Add Member Variable

**Примечание**

На мой взгляд, данный пример идеально иллюстрирует положение, согласно которому всем идентификаторам, переменным и т. д. необходимо давать содержательные имена. Например, если бы мы для элемента управления IDC_NAME использовали имя IDC_SOMETHING и нечто подобное для других элементов управления, то было бы трудно выбрать правильную переменную.

Здесь *m_pSet* — указатель на объект класса, производного от *CRecordset*, содержащий текущую запись. Переменная m_ProductName в этом объекте содержит текущее значение поля ProductName из таблицы Products. Если вы посмотрите содержимое комбинированного списка Member variable name, то увидите указатель для всех полей запроса, который для нас сконструировала библиотека MFC, и поэтому можно повторить рассмотренную процедуру сопоставления элементов управления формы с переменными для каждой из них.

После того как все элементы управления сопоставлены переменным, *обязательно* убедитесь, что все сделано корректно. Используйте для этого команду View/ClassWizard... (или одноименную кнопку на панели инструментов), чтобы отобразить диалоговое окно MFC ClassWizard, в котором выберите вкладку Member Variables (Компонентные переменные). В раскрывающемся списке Class name необходимо выбрать класс *CDBView*. Список компонентных переменных должен выглядеть так, как показано на рис. 21.6.

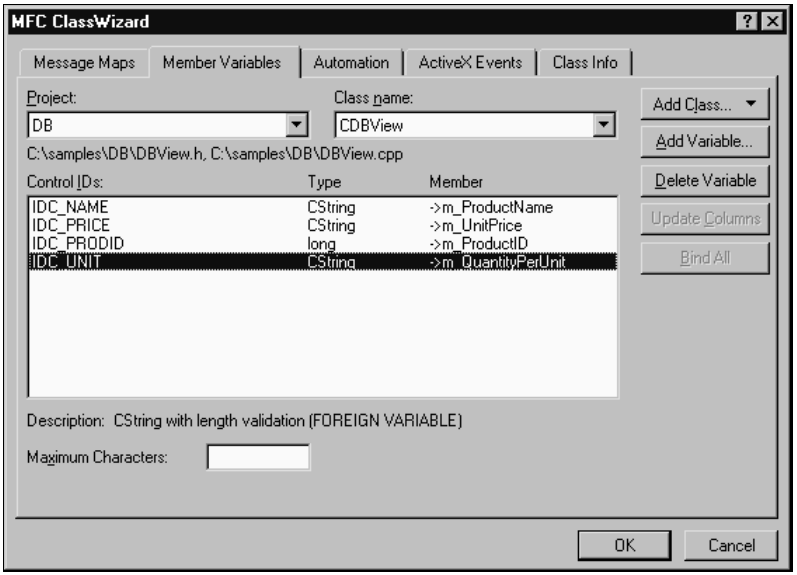



**Рис. 21.6.** Компонентные переменные класса *CDBView*

Теперь скомпилируйте и выполните приложение. Сразу после запуска вы увидите окно, показанное на рис. 21.7. Попробуйте перемещаться по записям базы данных, воспользовавшись соответствующими кнопками на панели инструментов или командами меню. И хотя мы пока не написали ни строчки кода, приложение работает прекрасно — можно переходить от записи к записи и даже изменять значения полей. Внесем некоторые изменения в приложение, чтобы добавить к значению цены обозначение денежной единицы.

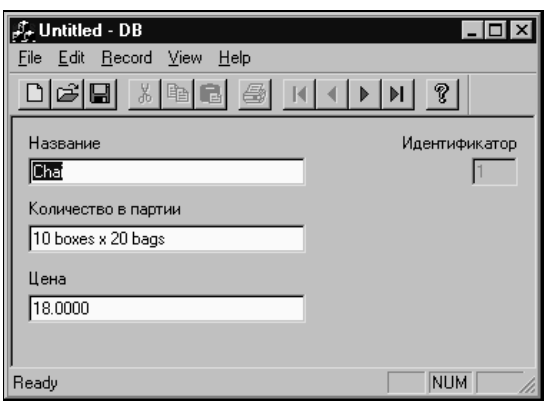



**Рис. 21.7.** Результат запуска приложения DB

Но перед этим необходимо внести некоторые изменения, связанные с компонентными переменными класса. Используйте команду View/ClassWizard,

чтобы отобразить диалоговое окно мастера MFC ClassWizard. Выберите вкладку Member Variables и класс *CDBView* аналогично тому, как вы это делали раньше. Первое, что необходимо выполнить — это удалить переменную, обозначенную в настоящий момент как IDC_PRICE. Выделите ее и нажмите кнопку Delete Variable. Теперь нажмите на кнопку Add Variable... и вы увидите диалоговое окно Add Member Variable. Наберите m_cPrice в качестве имени компонентной переменной (Visual C++ подставляет m_ автоматически). Нажмите кнопку OK, чтобы добавить переменную. Диалоговое окно MFC ClassWizard должно выглядеть аналогично показанному на рис. 21.8.

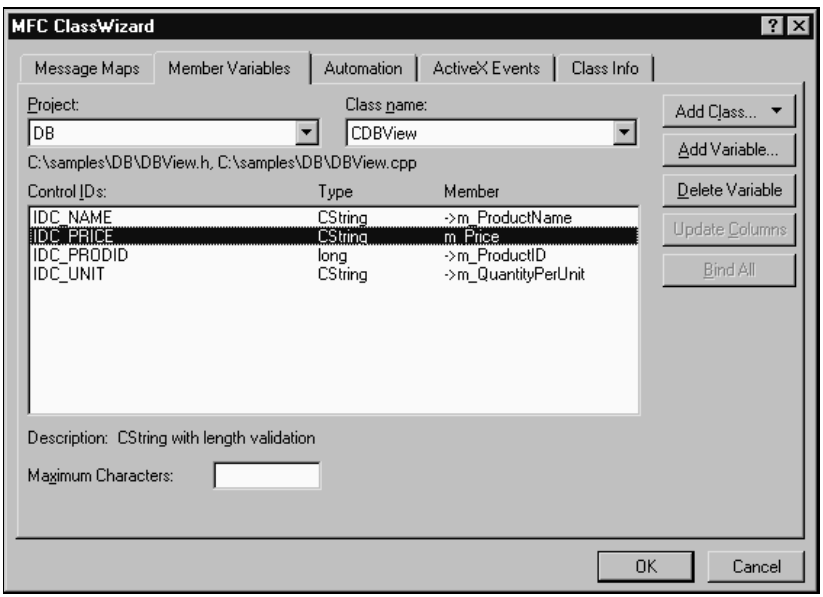



**Рис. 21.8.** Диалоговое окно MFC ClassWizard после внесенных изменений

Теперь, когда требуемая переменная создана, можно дополнить приложение необходимым кодом. Конечно, первое, что нам требуется — это отобразить информацию базы данных так, чтобы позднее мы могли видеть изменения. Ниже приведен фрагмент кода, который следует добавить в функцию *DoDataExchange()* класса *CDBView*. Убедитесь, что вы вставили новые строчки в начало, а не в конец функции, т. к. в противном случае получите странные результаты.

```
void CDBView::DoDataExchange(CDataExchange* pDX)
{
  int nLen, dec;
  CString tmp = "";
  // Получаем значение цены из результирующего набора
  m_Price = m_pSet->m_UnitPrice;
```

```
  // Получаем длину строки ...
  nLen = m_Price.GetLength();
  // и положение десятичной точки
  dec = m_Price.Find('.');
  // Удаляем заключительные нули и добавляем в начало '$'
  if((dec + 3) <= nLen)
    tmp = m_Price.Left(dec + 3);
  if(!tmp.IsEmpty())
    m_Price = "$";
  // Записываем результат в специальную переменную
  m_Price += tmp;
  // Обмен данными с результирующим набором,
  // на который указывает m_pSet
  CRecordView::DoDataExchange(pDX);
  //{{AFX_DATA_MAP(CDBView)
  DDX_FieldText(pDX, IDC_NAME, m_pSet->m_ProductName, m_pSet);
  DDX_FieldText(pDX, IDC_PRODID, m_pSet->m_ProductID, m_pSet);
  DDX_FieldText(pDX, IDC_UNIT, m_pSet->m_QuantityPerUnit, m_pSet);
  DDX_Text(pDX, IDC_PRICE, m_Price);
 //}}AFX_DATA_MAP
}
```

Скомпилируйте и запустите приложение. Результат представлен на рис. 21.9.

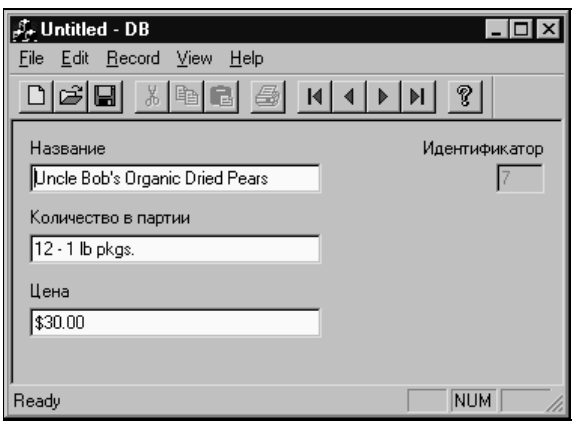



**Рис. 21.9.** Результат работы приложения DB после внесенных изменений

Как видите, все работает. Однако теперь возникла другая проблема — при попытке изменения поля "Цена" мы получаем странные результаты. Для устранения этой проблемы добавим некоторый код в конец функции *DoDataExchange()* класса *CDBView*, чтобы обновить базу данных после того, как изменили данные с помощью элемента управления в форме. Окончательный, на данный момент, вид функции *DoDataExchange()* приведен ниже.

```
void CDBView::DoDataExchange(CDataExchange* pDX)
{
  int nLen, dec;
  CString tmp = "";
  // Получаем значение цены из результирующего набора
  m_Price = m_pSet->m_UnitPrice;
  // Получаем длину строки ...
  nLen = m_Price.GetLength();
  // и положение десятичной точки
  dec = m_Price.Find('.');
  // Удаляем заключительные нули и добавляем в начало '$'
  if((dec + 3) <= nLen)
    tmp = m_Price.Left(dec + 3);
  if(!tmp.IsEmpty())
    m_Price = "$";
  // Записываем результат в специальную переменную
  m_strPrice += tmp;
  // Обмен данными с результирующим набором,
  // на который указывает m_pSet
  CRecordView::DoDataExchange(pDX);
  //{{AFX_DATA_MAP(CDBView)
  DDX_FieldText(pDX, IDC_NAME, m_pSet->m_ProductName, m_pSet);
  DDX_FieldText(pDX, IDC_PRODID, m_pSet->m_ProductID, m_pSet);
  DDX_FieldText(pDX, IDC_UNIT, m_pSet->m_QuantityPerUnit, m_pSet);
  DDX_Text(pDX, IDC_PRICE, m_Price);
  //}}AFX_DATA_MAP
  // Получаем длину строки
  nLen = m_Price.GetLength();
  // Удаляем знак '$', если он есть
  if(m_Price.Find('$') != -1)
    m_pSet->m_UnitPrice = m_Price.Right(nLen - 1);
  else
    m_pSet->m_UnitPrice = m_Price;
}
```

Если сейчас выполнить приложение, то можно будет перемещаться по записям базы данных, а также их корректировать. Кому-то может показаться, что для этого потребовалось достаточно много времени. Но уверяю вас, что это только в начале — по мере того, как вы будете разрабатывать новые и новые приложения, необходимое для этого время будет в основном тратиться на разработку логики работы, а отнюдь не на реализацию кода.

## Добавление и удаление записей в таблице

Всем понятно, что созданный только что пример не имеет практического значения — мало кому нужна программа, которая позволяет только про-

сматривать, пусть даже и обновлять, данные в одной таблице. Поэтому необходимо продолжить работу над приложением и добавить возможность добавления и удаления записей. Этим мы сейчас и займемся.

## Добавление записей в таблицу

Как и многое другое, эту возможность реализовать достаточно просто, поскольку большую часть работы мы можем переложить на библиотеку MFC. Основная наша задача — это корректным образом предоставлять ей информацию.

Как обычно, сначала позаботимся о некоторых связанных с основной задачей вопросах. Прежде всего необходимо создать либо элемент меню, либо кнопку на панели инструментов, либо и то, и другое. Чтобы не перегружать текст и основываясь на собственных симпатиях, остановимся на создание кнопки.

1. Раскройте вкладку Resource View (ресурсы) окна проектов. Щелкните по значку +, расположенному рядом с папкой Toolbar, а затем по IDR_MAINFRAME. Появится панель инструментов.
2. Выделите пустую кнопку, расположенную справа, и нарисуйте какой-либо символ New Record (Новая запись), например, такой, как показан на рис. 21.10. Переместите кнопку на место, расположенное сразу после четырех кнопок перемещения по базе данных.

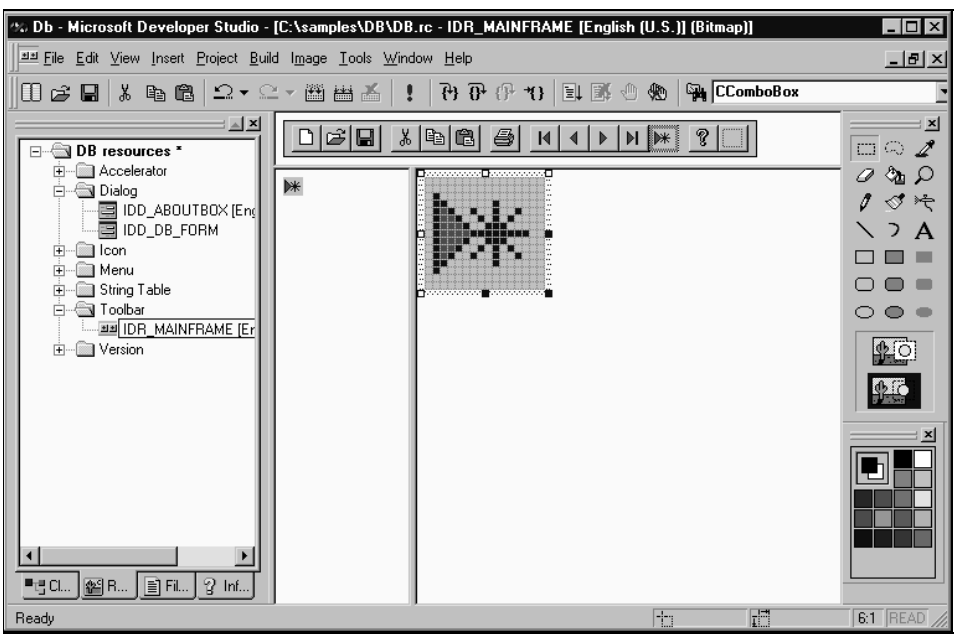



**Рис. 21.10.** Новая кнопка на панели инструментов для добавления новой записи

3. Дважды щелкните по вновь созданной кнопке на разрабатываемой панели инструментов. Раскроется диалоговое окно Toolbar Button Properties (Свойства кнопки панели инструментов). Введите в качестве идентификатора в список ID (Идентификатор) ID_RECORD_APPEND.
4. В поле Prompt (Подсказка) наберите текст — Append a new record to the table.\nNew Record (Добавление новой записи в таблицу\nНовая запись), как показано на рис. 21.11.
5. Нажмите кнопку закрытия в правом верхнем углу диалогового окна Toolbar Button Properties, и Visual C++ автоматически присвоит новой кнопке идентификатор ID_RECORD_APPEND.




**Рис. 21.11.** Диалоговое окно свойств кнопок панели инструментов

Теперь необходимо сопоставить созданной кнопке некоторые действия. Откройте окно ClassWizard, выполнив команду View/ClassWizard, выберите вкладку Message Maps (карта сообщений) и в списке Object IDs выделите элемент ID_RECORD_APPEND, а в раскрывающемся списке Class name выберите *CDBView* — класс, который служит для просмотра базы данных. Именно для него потребуется добавить некоторый код.

Далее в списке Message выделите элемент COMMAND и нажмите кнопку Add Function, в результате чего на экране появится диалоговое окно Add Member Function, в котором уже предложено некоторое имя (рис. 21.12), базирующееся на идентификаторе кнопки (присваивайте всем идентификаторам содержательные имена!).

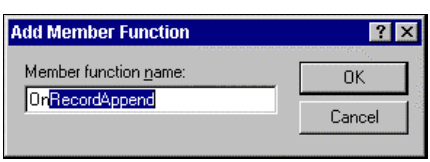



**Рис. 21.12.** Если идентификатору присвоено содержательное имя, то можно просто нажать кнопку OK

Нажмите кнопку OK, и диалоговое окно MFC ClassWizard примет вид, показанный на рис. 21.13. В этом окне нажмите кнопку Edit Code (Редактировать код), чтобы сразу перейти к шаблону созданной функции, куда потребуется добавить код, приведенный ниже.

```
void CDBView::OnRecordAppend()
{
```

```
  // Сначала проверяем, не была ли открыта база данных
  // в режиме только для чтения. Если это так,
  // то выводим предупреждение и выходим из функции
  if(m_pSet->CanAppend() == 0)
    MessageBox("Можно только просматривать записи",
               "Ошибка добавления записи",
               MB_OK | MB_ICONERROR);
  else
  {
    // Перемещаемся на первую запись
    m_pSet->MoveFirst();
    // Создаем пустую запись,
    // в которую пользователь будет вводить значения
    m_pSet->AddNew();
    // Устанавливаем флаг перехода в режим добавления записи
    m_bAdd = TRUE;
    // Обновляем поля формы
    UpdateData(FALSE);
  }
}
```

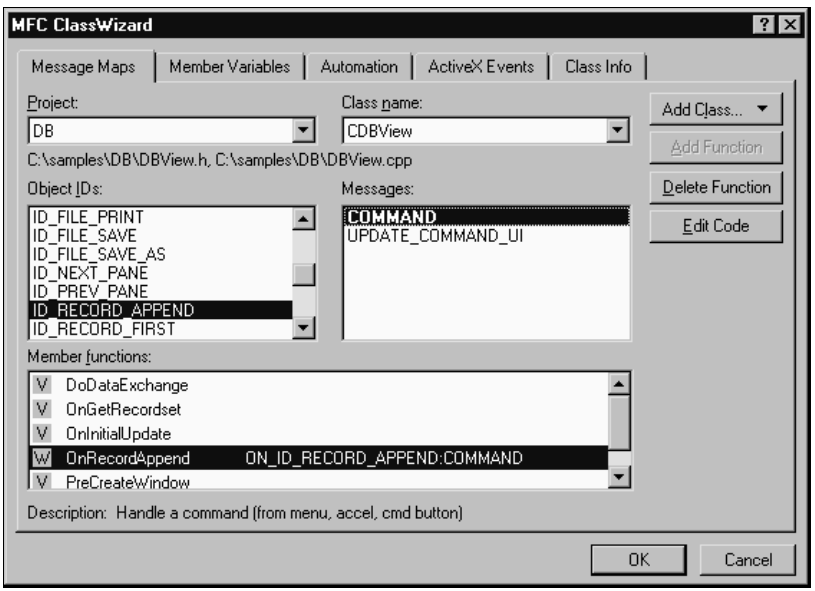



**Рис. 21.13.** Диалоговое окно MFC ClassWizard после проведенных действий

Рассмотрим, как работает функция *OnRecordAppend*. Прежде всего, она проверяет, можно ли добавлять данные в базу данных. Сделать это можно с помощью функции *CanAppend* класса *CRecordset*, которая была подробно опи-

сана в главе 20. Если добавлять записи нельзя, то необходимо вывести сообщение об ошибке и завершить работу, что и сделано в нашем примере.

Далее нужно подготовить элементы управления в форме для ввода новых значений. Для этого мы перемещаемся в начало набора записей (функция *MoveFirst*), вызываем специальную функцию *AddNew*, подготавливающую результирующий набор к приему новой записи, и устанавливаем флаг, что это новые данные. Осталось только очистить поля элементов управления, что и делает функция *UpdateData*(FALSE).

Итак, элементы управления в форме и результирующий набор готовы к приему данных. Теперь необходимо решить вопрос, каким образом и где передать введенные в форме значения в результирующий набор. Самым простым представляется способ добавления специальной кнопки, нажатие на которую вызовет специальный обработчик, где введенные значения перепишутся в результирующий набор. Второй способ не требует создания дополнительных элементов управления и для фиксации значений использует переопределенную функцию перемещения по записям результирующего набора *Move* (как получить для нее заготовку, вы уже должны знать).

```
BOOL CDBView::OnMove(UINT nIDMoveCommand)
{
  // Проверяем, в каком режиме — добавления или перемещения —
  // мы находимся
  if(m_bAdd)
  {
    // Читаем данные из элементов управления в форме
    // непосредственно в результирующий набор
    if(!UpdateData())
      return FALSE;
    // Переписываем новые данные в базу данных,
    // позаботившись об обработке ошибок посредством механизма
    // обработки исключений
    try
    {
      m_pSet->Update();
    }
    catch(CDBException e)
    {
      AfxMessageBox(e.m_strError);
      return FALSE;
    }
    // Поскольку используется результирующий набор типа
    // "мгновенный список", перечитываем информацию из базы данных
    m_pSet->Requery();
    // Обновляем поля формы
    UpdateData(FALSE);
```

```
    // "Выходим" из режима добавления
    m_bAdd = FALSE;
  }
  else
    // Переходим к следующей записи в результирующем наборе
    return CRecordView::OnMove(nIDMoveCommand);
}
```

Кроме того, необходимо определить переменную *m_bAdd*:

```
class CDBView : public CRecordView
{
...
private:
  BOOL m_bAdd;
};

CDBView::CDBView() : CRecordView(CDBView::IDD)
{
...
  m_bAdd  = FALSE;
}
```

Рассмотрим логику работы функции *OnMove*. Если мы попали в нее после ввода новой записи (установлен флаг *m_bAdd*), то переписываем введенные значения в базу данных. В противном случае просто переходим к следующей записи. Нас, естественно, интересует режим добавления.

Прежде всего необходимо прочитать данные из элементов управления формы. Обратите внимание, что переменные класса *CDBView* определены так, что мы это делаем непосредственно в результирующий набор, используя функции обмена полями данных, например, для поля *IDC_PROD*.

```
DDX_FieldText(pDX, IDC_PROD, m_pSet->m_ProductName, m_pSet);
```

Для отслеживания возможных ошибок мы воспользовались механизмом обработки исключений, реализованном в библиотеке MFC.

**Примечание**

Для тех, кто заинтересуется этим механизмом, могу порекомендовать книгу А. Мешкова и Ю. Тихомирова "Visual C++ и MFC", в которой данный вопрос изложен достаточно подробно.

Если никакой ошибки не произошло, то перечитаем информацию из базы данных, чтобы работать с самым последним набором. Теперь осталось только обновить элементы управления в форме. Помните, в функции *OnRecordAppend* мы перед входом в режим добавления переместились на первую запись ре-

зультирующего набора. Поэтому необходимо вызвать функцию *UpdateData*(FALSE), чтобы отобразить ее в элементах управления формы. И, наконец, не забываем сбросить флаг *m_bAdd*, чтобы избежать ошибок.

Вернемся к табл. 21.1, где представлены поля таблицы Products. Видим, что у столбца ProductID установлен тип данных AutoNumber. Это означает, что система управления базами данных (в нашем случае Access) сама позаботится о присвоении значения этому столбцу. Попробуйте ввести какое-либо значение, меньшее 77, в поле "Идентификатор". Результат, который вы получите, представлен на рис. 21.14.




**Рис. 21.14.** Попытка вставить повторяющееся значение в столбец, для которого установлен тип AutoNumber

Чтобы решить эту проблему, достаточно просто поменять переменную, с которой связан элемент управления IDC_PRODID, и тем самым переложить задачу на Access. Вот как это должно выглядеть (рис. 21.15).

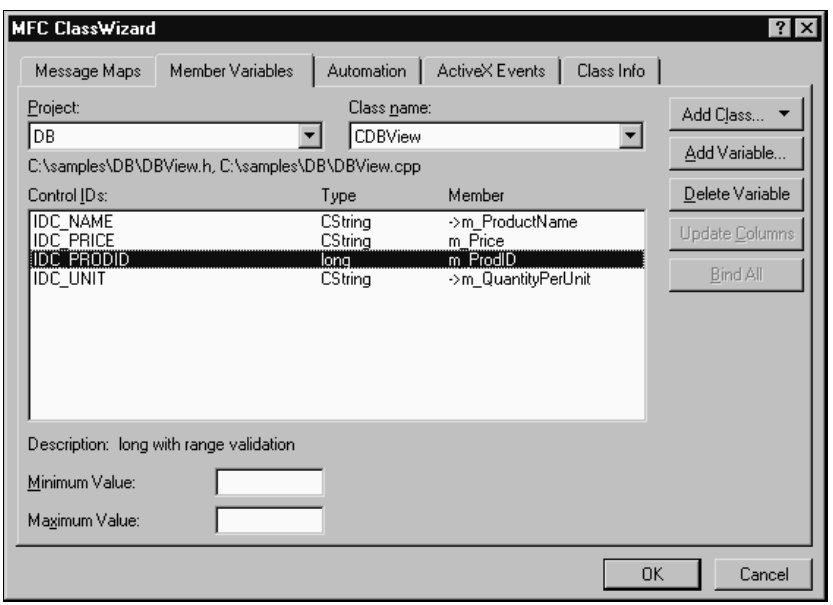



**Рис. 21.15.** Для элемента управления IDC_PRODID необходимо определить другую переменную

Кроме этого, в код требуется также добавить одну строку, чтобы можно было видеть информацию об идентификаторе.

```
void CDBView::DoDataExchange(CDataExchange* pDX)
{
...
  // Для отображения значений поля ProductID таблицы
  // необходимо выполнить присваивание
  m_ProdID = m_pSet->m_ProductID;
  // Осуществляем обмен данными с элементами управления в форме
  CRecordView::DoDataExchange(pDX);
  //{{AFX_DATA_MAP(CDBView)
  DDX_Text(pDX, IDC_PRICE, m_Price);
  DDX_FieldText(pDX, IDC_NAME, m_pSet->m_ProductName, m_pSet);
  DDX_FieldText(pDX, IDC_UNIT, m_pSet->m_QuantityPerUnit, m_pSet);
  DDX_Text(pDX, IDC_PRODID, m_ProdID);
  //}}AFX_DATA_MAP
...
}
```

Вот, собственно, и все, что надо выполнить. Однако, чтобы не возникало искушения вводить что-либо в поле идентификатора IDC_PRODID, уберем его с экрана на время вставки новой записи. Для этого немного модифицируем код в двух функциях *OnRecordAppend* и *OnMove*, как показано ниже.

```
void CDBView::OnRecordAppend()
{
  // Сначала проверяем, не была ли открыта база данных
  // в режиме только для чтения. Если это так,
  // то выводим предупреждение и выходим из функции
  if(m_pSet->CanAppend() == 0)
    MessageBox("Можно только просматривать записи",
               "Ошибка добавления записи",
               MB_OK | MB_ICONERROR);
  else
  {
    // Скрываем элементы управления на время ввода новой записи
    GetDlgItem(IDC_PRODST)->ShowWindow(SW_HIDE);
    GetDlgItem(IDC_PRODID)->ShowWindow(SW_HIDE);
    // Устанавливаем фокус на поле ввода названия
    GetDlgItem(IDC_NAME)->SetFocus();
    // Перемещаемся на первую запись
    m_pSet->MoveFirst();
    // Создаем пустую запись,
    // в которую пользователь будет вводить значения
    m_pSet->AddNew();
```

```
    // Устанавливаем флаг перехода в режим добавления записи
    m_bAdd = TRUE;
    // Обновляем поля формы
    UpdateData(FALSE);
  }
}
BOOL CDBView::OnMove(UINT nIDMoveCommand)
{
  // Проверяем, в каком режиме — добавления или перемещения -
  // мы находимся
  if(m_bAdd)
  {
    // Читаем данные из элементов управления в форме
    // непосредственно в результирующий набор
    if(!UpdateData())
      return FALSE;
    // Переписываем новые данные в базу данных,
    // позаботившись об обработке ошибок посредством механизма
    // обработки исключений
    try
    {
      m_pSet->Update();
    }
    catch(CDBException e)
    {
      AfxMessageBox(e.m_strError);
      return FALSE;
    }
    // Поскольку используется результирующий набор типа
    // "мгновенный список", перечитываем информацию из базы данных
    m_pSet->Requery();
    // Обновляем поля формы
    UpdateData(FALSE);
    // "Выходим" из режима добавления
    m_bAdd = FALSE;
    // После завершения ввода снова отображаем элементы управления
    GetDlgItem(IDC_PRODST)->ShowWindow(SW_SHOW);
    GetDlgItem(IDC_PRODID)->ShowWindow(SW_SHOW);
  }
  else
    // Переходим к следующей записи в результирующем наборе
    return CRecordView::OnMove(nIDMoveCommand);
}
```

После внесения этих изменений скомпилируйте и запустите приложение. После нажатия кнопки New Record и ввода информации о новом продукте основное окно будет иметь вид, аналогичный представленному на рис. 21.16.

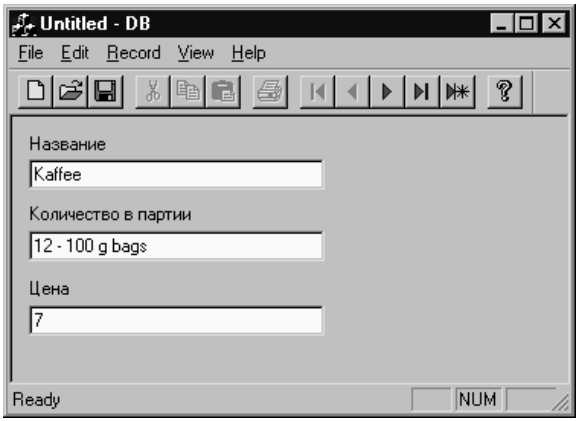



**Рис. 21.16.** Так теперь выглядит основное окно приложения в режиме ввода новой записи в базу данных

**Примечание**

Не забудьте изменить идентификатор статического поля "Идентификатор" на IDC_PRODST, иначе во время компиляции получите сообщение об ошибке.

Итак, теперь мы умеем обновлять и добавлять записи в базу данных. Следующим шагом является удаление записей.

## Удаление записей из таблицы

Как обычно, начнем с того, что добавим в наше приложение кнопку на панели инструментов, нажатие на которую позволит перейти в режим удаления. Чтобы не придумывать чего-либо непонятного, воспользуемся стандартным способом удаления, принятым во всех приложениях Windows (рис. 21.17).

Создайте функцию *OnRecordDelete*, воспользовавшись стандартной, описанной ранее, процедурой добавления новой функции, результат вызова которой представлен на рис. 21.18:

```
void CDBView::OnRecordDelete()
{
  // Последняя возможность сохранить запись
  if(MessageBox("Вы действительно хотите удалить эту запись?",
                "Удаление записи",
                MB_OKCANCEL|MB_ICONQUESTION|MB_DEFBUTTON2) == IDOK)
  {
    // Удаляем запись
    try
    {
      m_pSet->Delete();
    }
    catch(CDBException e)
    {
```

```
      // Если что-то произошло — сообщаем об этом
      AfxMessageBox(e.m_strError);
      return;
    }
    // Поскольку используется результирующий набор типа
    // "мгновенный список", перечитываем информацию из базы данных
    m_pSet->Requery();
    // После перечитывания данных из базы данных
    // переходим к следующей записи
    m_pSet->MoveNext();
    // Если удаляем запись из конца,
    // то устанавливаем указатель на последнюю запись
    if(m_pSet->IsEOF())
      m_pSet->MoveLast();
    // Если таблица стала пустой, то очищаем поля
    if(m_pSet->IsBOF ())
      m_pSet->SetFieldNull(NULL);
  }
  // Обновляем поля формы
  UpdateData(FALSE);
}
```

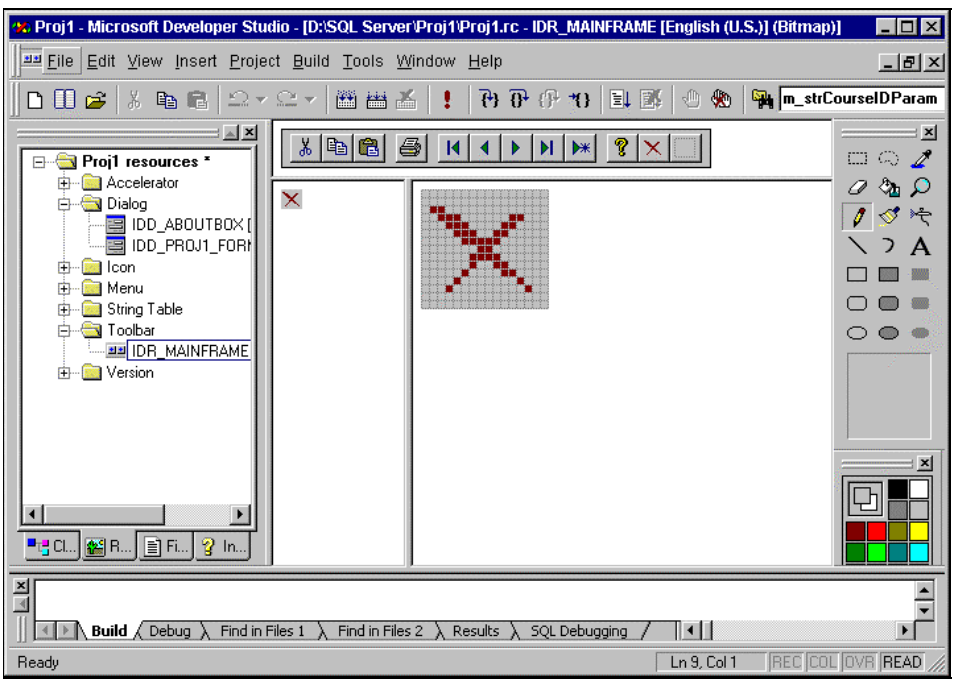



**Рис. 21.17.** Добавляем на панель инструментов новую кнопку

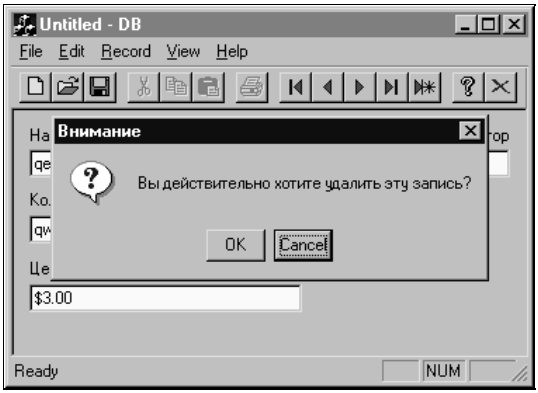



**Рис. 21.18.** После отказа от продукта запись о нем нужно удалить из базы данных

Как видите, принцип здесь тот же, что и при добавлении записи. Правила хорошего тона требуют, чтобы перед удалением пользователю был задан вопрос, хорошо ли он подумал и не случайно ли нажал кнопку удаления. Если запись действительно надо удалить, то смело нажимайте кнопку OK, после чего будет вызвана функция *Delete* класса *CRecordset*, которая проведет корректное удаление записи из базы данных. Чтобы обновить набор, с которым мы работаем, необходимо вызвать функцию *Requery*, иначе мы будем продолжать работать с записью, которой в базе данных уже нет. Этот шаг был бы не нужен, если бы мы работали с динамическим набором (dynaset) — как вы помните, в этом случае результирующий набор обновляется одновременно с обновлением базы данных. Новыми при удалении являются следующие два действия — проверка на выход за начало и конец записей. Действительно, если мы переместились (вызов функции *MoveNext*) за последнюю запись (проверка `if (m_pSet->IsEOF())`), то непонятно, на что в этом случае будет указывать указатель. Поэтому его нужно установить на последнюю запись результирующего набора (вызов `m_pSet->MoveLast()`). Аналогичные действия необходимо предпринять для проверки на выход за "левую границу", т. е. за первую запись (вызов `if (m_pSet->IsBOF())`). Эта ситуация возникает тогда, когда из таблицы удаляется последняя строка и требуется очистить поля единственной пустой строки (вызов `m_pSet->SetFieldNull(NULL)`).

Как и в случае обновления и добавления новой записи, здесь также всю основную работу делает библиотека MFC, а точнее, ее класс *CRecordset*.

Но пойдем дальше и рассмотрим еще некоторые возможности работы с базой данных.

## Сортировка записей

Используя классы библиотеки MFC, организовать сортировку записей очень просто. Если обратиться к главе 20, где описаны компоненты классов, то ответ на вопрос: "Как организовать сортировку записей таблицы базы дан-

ных?" становится тривиальным. Поэтому перейду сразу к примеру. Для демонстрации этой возможности создадим специальные элементы меню, внешний вид которых представлен на рис. 21.19.

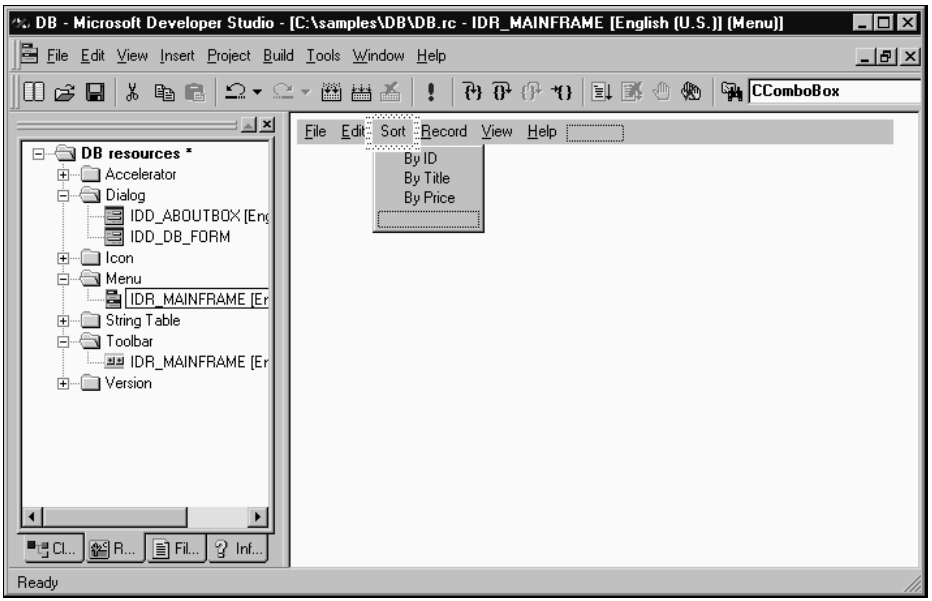



**Рис. 21.19.** Элементы меню, позволяющие задавать различные критерии сортировки

Как видите, мы использовали для организации сортировки три различных критерия: по идентификатору продукта, по названию и цене. После того как элементы меню созданы и им присвоены соответствующие идентификаторы (пример одного такого идентификатора приведен на рис. 21.20), необходимо создать обработчик соответствующих команд.

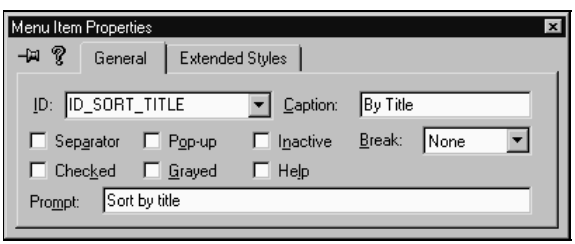



**Рис. 21.20.** Окно свойств элементов меню

Для этого можно воспользоваться ClassWizard, но тогда придется для каждой команды создавать свой собственный обработчик, а можно использовать специальный тип обработчика, который позволяет "реагировать" на команды из некоторого диапазона. По этому пути мы и пойдем. Для облегчения работы начнем все-таки с помощью ClassWizard и создадим обработчик для

какой-нибудь одной команды, например, ID_BY_ID. Способ, которым это делается, ничем не отличается от рассмотренного нами для обработчиков команд кнопок панели инструментов. Поскольку в дальнейшем мы будем модифицировать этот обработчик, чтобы он "мог работать" со всеми командами сортировки, дадим ему обобщенное имя — OnSort.

Теперь необходимо внести некоторые изменения в код, созданный ClassWizard, согласно приведенному ниже тексту:

```
BEGIN_MESSAGE_MAP(CDBView, CRecordView)
 //{{AFX_MSG_MAP(CDBView)
 ON_COMMAND(ID_RECORD_APPEND, OnRecordAppend)
 ON_COMMAND(ID_RECORD_DELETE, OnRecordDelete)
 ON_COMMAND_RANGE(ID_SORT_ID, ID_SORT_PRICE, OnSort)
 ON_UPDATE_COMMAND_UI_RANGE(ID_SORT_ID, ID_SORT_PRICE, OnUpdateSort)
 //}}AFX_MSG_MAP
END_MESSAGE_MAP()
class CDBView : public CRecordView
{
...
// Generated message map functions
protected:
  // Переменная, позволяющая отслеживать текущий режим сортировки
  UINT m_nSort;
...
};
```

Теперь необходимо добавить код в обработчики сообщений:

```
// Обработчик команд ID_SORT_ID — ID_SORT_PRICE (один на все)
void CDBView::OnSort()
{
  // Получаем идентификатор выбранной команды
  m_nSort = LOWORD(GetCurrentMessage()->wParam);
  // Организуем установку соответствующего критерия сортировки
  switch(m_nSort)
  {
    // Сортировка по идентификатору
    case ID_SORT_ID:
      m_pSet->m_strSort = "ProductID";
      break;
    // Сортировка по названию
    case ID_SORT_TITLE:
      m_pSet->m_strSort = "ProductName";
      break;
    // Сортировка по цене
    case ID_SORT_PRICE:
```

```
        m_pSet->m_strSort = "UnitPrice";
        break;
    }
    // Чтобы учесть новый порядок сортировки,
    // необходимо выполнить запрос к базе данных
    m_pSet->Requery();
    // Перемещаемся к первой записи результирующего набора
    m_pSet->MoveFirst();
    // Обновляем содержимое элементов управления в форме
    UpdateData(FALSE);
}
// Обработчик команд обновления
// ID_SORT_ID — ID_SORT_PRICE (один на все)
void CDBView::OnUpdateSort(CCmdUI* pCmdUI)
{
    // Устанавливаем или убираем галочку
    // у соответствующего элемента меню
    pCmdUI->SetCheck(pCmdUI->m_nID == m_nSort);
}
```

Все на удивление просто, и приведенных комментариев должно быть вполне достаточно, чтобы разобраться с тем, каким образом реализуются различные режимы сортировки. Скомпилируйте и запустите приложение. Результат его работы при выборе сортировки по названию представлен на рис. 21.21.

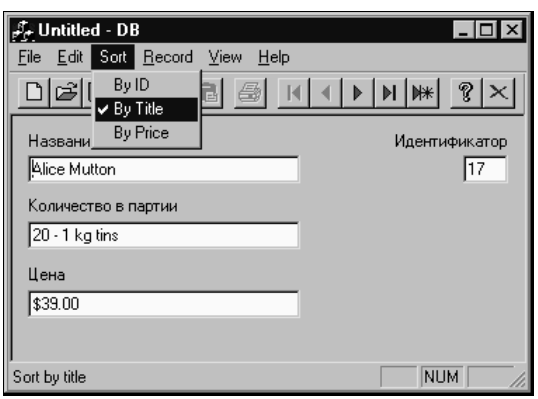



**Рис. 21.21.** Продукты отсортированы по названию

Если вы решите обойтись без помощи ClassWizard, то не забудьте также добавить необходимые строки в определение класса *CDBView*:

```
class CDBView : public CRecordView
{
...
// Generated message map functions
protected:
```

```
  // Переменная, позволяющая отслеживать текущий режим сортировки
  UINT m_nSort;
  //{{AFX_MSG(CDBView)
  afx_msg void OnRecordAppend();
  afx_msg void OnRecordDelete();
  afx_msg void OnSort();
  afx_msg void OnUpdateSort(CCmdUI* pCmdUI);
  //}}AFX_MSG
...
};
```

Последний режим, который мы рассмотрим в этой главе, относится к работе с компонентной переменной *m_srtFilter* класса *CRecordset*.

## Поиск информации в базе данных

Вопросам организации наиболее эффективного поиска посвящены целые книги. Поэтому выбору наиболее эффективного режима поиска мы не будем уделять внимание. При использовании ODBC существует простой способ, который, может быть, не является самым быстрым и эффективным, но в то же время является абсолютно надежным.

Очевидно, что, как обычно, необходимо потратить некоторое время на модификацию меню и/или панели инструментов, прежде чем переходить непосредственно к поиску. Я больше не буду останавливаться на этом вопросе и только приведу внешний вид кнопки, которую я добавил (рис. 21.22).

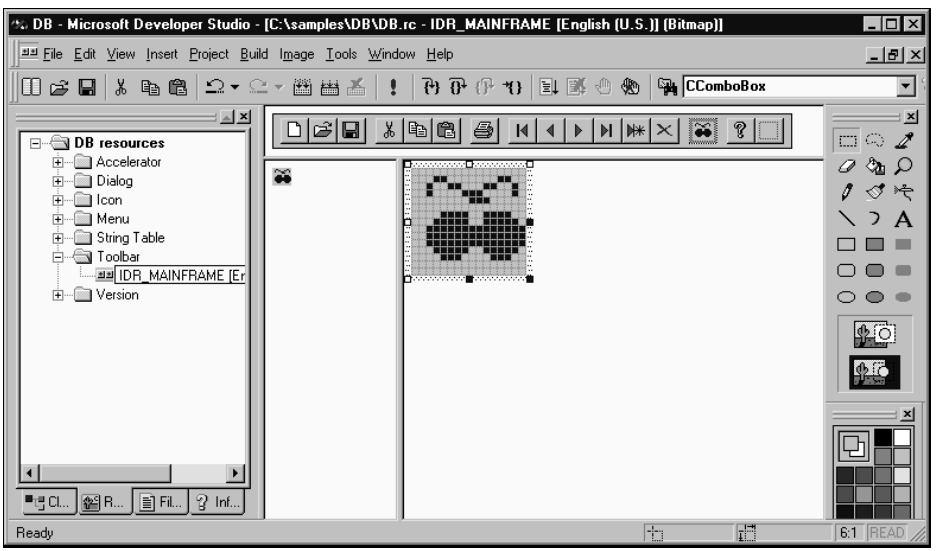



**Рис. 21.22.** Кнопка, которая выглядит подобно паре очков, предназначена для запуска поиска записи в базе данных

Для того чтобы запросить у пользователя, что именно нужно найти, необходимо создать небольшое диалоговое окно. Вернитесь в окно Resource View. Щелкните правой кнопкой мыши на папке Dialog и выберите из контекстного меню Insert Dialog (Вставить диалоговое окно). Visual C++ представит новое окно диалога. Необходимо изменить некоторые его свойства. Прежде всего нужно присвоить этому диалоговому окну более содержательное имя. Щелкните правой кнопкой мыши на элементе IDD_DIALOG1 в Resource View и выберите Properties из контекстного меню. Вы увидите диалоговое окно Dialog Properties, аналогичное показанному на рис. 21.23.

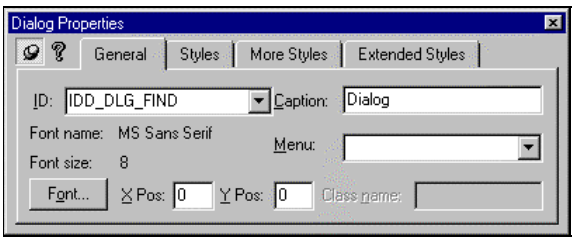



**Рис. 21.23.** Свойства блока диалога

Наберите IDD_DLG_FIND в поле ID. Далее следует дать этому диалоговому окну более содержательный заголовок. Наберите "Поиск информации в базе данных" в поле ввода Caption (Заголовок), а затем нажмите кнопку Close, расположенную в правом верхнем углу окна, чтобы завершить ввод изменений.

Теперь нам необходимо добавить один элемент управления Static Text и связанный с ним элемент управления Edit Box, чтобы завершить создание блока диалога. Его окончательный вид показан на рис. 21.24.

Последняя часть работы для этого диалогового окна — установить свойства элемента управления Edit Box. Щелкните правой кнопкой мыши на этом элементе управления и выберите Properties из контекстного меню. Измените свойство ID в диалоговом окне Edit Properties (Свойства текстового поля) на IDC_RECORD_FIND. Нажмите кнопку Close, расположенную в правом верхнем углу окна. Теперь при нажатой клавише <Ctrl> дважды щелкните на элементе управления Edit Box. При этом вы увидите блок диалога Adding a Class (Добавить класс), аналогичный показанному на рис. 21.25.

Причина появления этого блока диалога заключается в том, что каждый диалог в вашей программе должен быть с чем-то связан. Так же, как окно About, имеющееся в любом приложении, каждое диалоговое окно должно иметь свой собственный класс. Нажмите кнопку OK, чтобы создать новый класс для этого диалогового окна. Visual C++ отобразит диалоговое окно New Class, в котором нужно набрать имя класса в поле Name. Вы можете использовать любое имя класса, какое хотите, а я выбрал *CFindPrice*. Нажмите кнопку OK, чтобы завершить создание нового класса, после чего еще раз нажмите OK, чтобы закрыть диалоговое окно MFC ClassWizard. Затем

при нажатой клавише <Ctrl> еще раз дважды щелкните на элементе управления Edit Box, появится диалоговое окно Add Member Variable, которое вы и ожидали увидеть. Наберите m_FindPrice в поле Member variable name. Убедитесь, что поле Category содержит Value, а поле Variable Type — CString. Нажмите кнопку OK, чтобы добавить компонентную переменную.

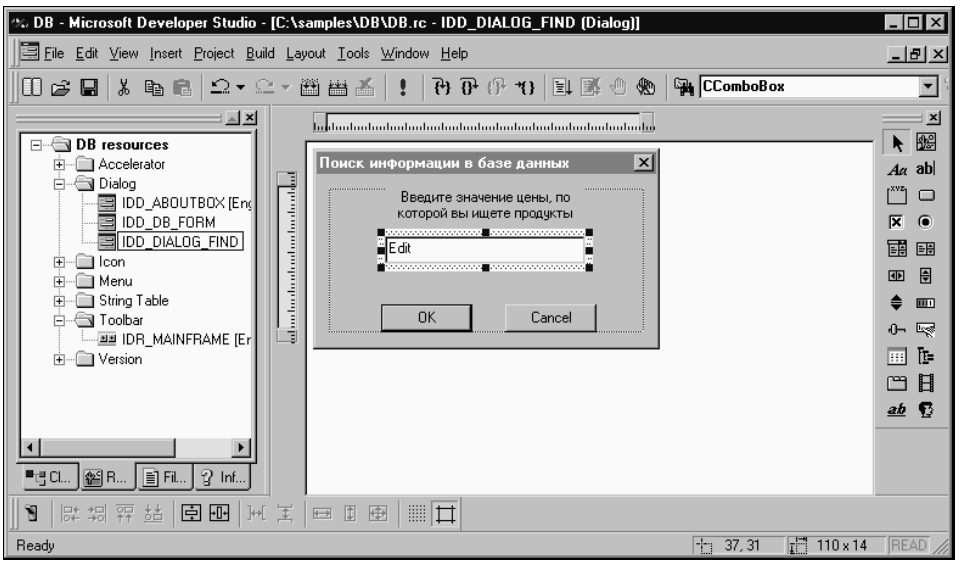



**Рис. 21.24.** Диалоговое окно "Поиск информации в базе данных" позволит пользователю осуществлять поиск в базе данных

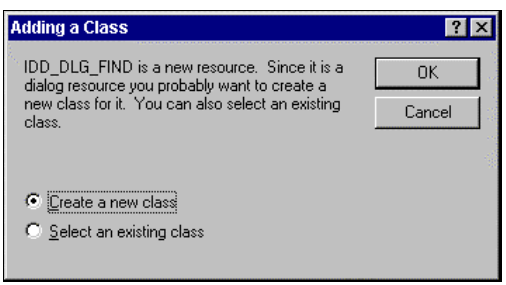



**Рис. 21.25.** Диалоговое окно Adding a Class

Осталось создать оболочку и саму функцию *OnRecordFind()*:

```
void CDBView::OnRecordFind()
{
  // Создаем экземпляр диалогового окна
  CFindPrice dlgFind;
  // Нашли соответствующую запись?
  BOOL bFound = FALSE;
```

```
  // Расположение закладки
  CDBVariant varBookmark;
  // Отображаем диалоговое окно и определяем,
  // какую кнопку пользователю нажимать для выхода
  if(dlgFind.DoModal() == IDOK)
  {
    // Сохраняем текущую позицию, если закладки поддерживаются
    if(m_pSet->CanBookmark())
      m_pSet->GetBookmark(varBookmark);
    // Идем к началу
    m_pSet->MoveFirst();
    // Приводим введенное значение к определенному формату
    if(dlgFind.m_FindPrice.Find('.') == -1)
      dlgFind.m_FindPrice += ".0000";
    else
      dlgFind.m_FindPrice += "00";
    // Ищем цену, введенную пользователем
    while(!m_pSet->IsEOF() ^ bFound)
    {
      // Проверяем, равны ли значения
      if(m_pSet->m_UnitPrice.CompareNoCase(dlgFind.m_FindPrice)==0)
        bFound = TRUE;
      // Переходим к следующей записи
      else
        m_pSet->MoveNext();
    }
    if(bFound)
    {
      // Отображаем данные о новой записи, если она найдена
      UpdateData(FALSE);
    } else
    {
      // Отображаем сообщение об ошибке, если запись не найдена
      MessageBox("Запись не найдена",
                 "Ошибка",
                 MB_OK | MB_ICONERROR);
      // Восстанавливаем текущую позицию,
      // если база данных поддерживает закладки
      if(m_pSet->CanBookmark())
        m_pSet->SetBookmark(varBookmark);
      else
        // В противном случае переходим к первой записи
        m_pSet->MoveFirst();
    }
  }
}
```

Первое, на что следует обратить внимание в этом коде — мы запоминаем текущую позицию в базе данных, используя для этого закладки. Они работают практически так же, как закладки, которые мы используем для книги, чтобы отметить место, где читаем. Поэтому, используя функцию *CanBookmark()*, нужно прежде всего убедиться в том, что система управления базами данных поддерживает закладки. Если это так, то можно применять функцию *GetBookmark()*, чтобы отметить текущую позицию, и функцию *SetBookmark ()*, чтобы восстанавливать ее.

Обратите внимание, что предложенная процедура поиска очень проста. Если мы обнаружили запись, которую затребовал пользователь, мы устанавливаем булевскую переменную на значение "истина", в противном случае идем к следующей записи. Этот процесс продолжается до тех пор, пока мы либо не найдем первую запись, соответствующую пользовательскому критерию, либо не закончим запрос. Если вы устанавливаете аналогичную процедуру поиска, убедитесь, что проверяете оба момента.

Если требуемая запись не будет обнаружена, программа выведет сообщение об ошибке. Она также старается возвратить пользователя к предыдущей записи, если система управления базами данных поддерживает закладки. В противном случае программа возвращает пользователя к первой записи в запросе. Крайне важно установить указатель на какую-нибудь запись, так как в противном случае система управления базами данных, вероятно, перейдет к обработке исключения, сообщая, что запись потеряна.

**Примечание**

После окончания поиска следует в обязательном порядке переустановить указатель записи, чтобы запрос находился на точно известной записи.

Если мы найдем запись, то должны отобразить информацию в форме. Это делает оставшаяся часть кода, приведенного в этом разделе.

Однако, прежде чем компилировать и выполнять наше приложение, необходимо внести еще одно дополнение — в начале листинга файла <dbview.cpp> расположены директивы #include. Чтобы можно было работать с диалоговым окном FindPrice, необходимо включить в модуль его заголовочный файл. Добавьте новую директиву #include, аналогичную показанной здесь:

```
// Для поддержки блока диалога FindPrice,
// в начало файла DBView.CPP необходимо
// самостоятельно включить строку
#include "FindPrice.h"
```

Теперь, когда есть все фрагменты, скомпилируйте и выполните программу. В нашей маленькой таблице процедура поиска выполняется очень быстро, но будет замедляться по мере увеличения размера таблицы. Если вы планируете работать с огромными базами данных, вам потребуется более тщатель-

но подойти к разработке процедуры поиска. Однако и рассмотренная процедура прекрасно работает с таблицей, содержащей около 10 000 записей. Конечно, все это зависит от сложности связей таблицы и числа полей, которые она содержит. Как выглядит созданный блок диалога в действии, показано на рис. 21.26, а результат выполнения поиска — на рис. 21.27.

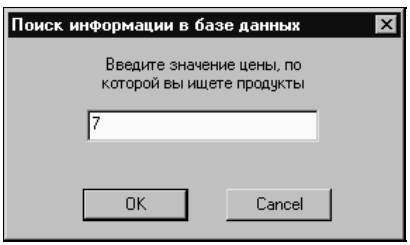



**Рис. 21.26.** Блок диалога FindPrice в действии

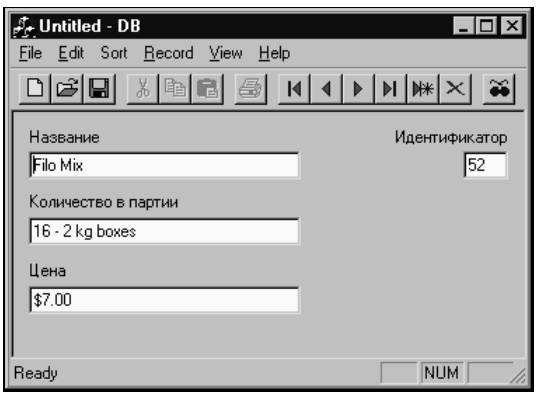



**Рис. 21.27.** Результат выполнения поиска в таблице базы данных

Если искомой записи в таблице нет, то появится соответствующее сообщение об ошибке (рис. 21.28).

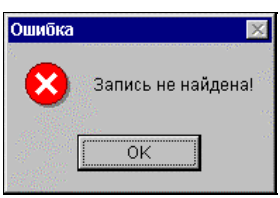



**Рис. 21.28.** Запись в таблице не найдена

На первый взгляд все работает хорошо и так, как ожидалось. Однако у реализованного способа поиска есть существенный недостаток (я не имею в виду ограниченность критерия поиска одним полем таблицы). Попробуйте несколько раз запросить поиск продуктов по цене $7. И что вы получите в результате? Правильно, каждый раз наша поисковая система будет находить один единственный продукт, который располагается первым в результирующем наборе. Задачу усовершенствования используемого алгоритма поиска я оставляю вам на самостоятельную проработку.



# Часть VIII

## Приложения

**Приложение 1.** Основы языка программирования C++

**Приложение 2.** Основные типы сообщений Windows

**Приложение 3.** Описание сопроводительной дискеты

# ПРИЛОЖЕНИЕ 1

# Основы языка программирования C++

Знакомство с языком программирования C++ хотелось бы начать с цитаты из книги разработчика языка Бьерна Страуструпа:

"Язык программирования служит двум связанным между собой целям: он дает программисту аппарат для задания действий, которые должны быть выполнены, и формирует концепции, которыми пользуется программист, размышляя от том, что делать...

Язык поддерживает стиль программирования, если он предоставляет соответствующие средства и делает удобным (т. е. легким, безопасным и эффективным) использование этого стиля... Например, можно писать структурированные программы на языке Fortran и объектно-ориентированные программы на C, но это неизменно будет трудно (во втором случае — особенно), поскольку они не поддерживают эти парадигмы.

Язык программирования C++ был разработан так, чтобы в дополнение к традиционной технологии программирования на C поддерживать абстракцию данных и объектно-ориентированное программирование при определяющей роли следующих пяти ограничений, первое из которых выражает чисто эстетический и логический идеалы, следующие два — идеал минимальности и, наконец, два последних могут быть охарактеризованы как "то, что вам не нужно, скорее всего не причинит ущерба":

- Все средства должны быть искусно и элегантно встроены в язык.
- Должна присутствовать возможность использования этих средств в комбинации друг с другом.
- Число паразитных и предназначенных для специальных целей средств должно быть минимальным.
- Реализация каждого средства не должна приводить к существенным накладным расходам в программах, если этого можно избежать.
- Программисту необходимо знать только ту часть языка, которая непосредственно касается программ данного класса.

Однако сказанное не означает, что программистам навязывается определенный стиль программирования."

После знакомства с теми общими положениями, которые хотел вложить в C++ его создатель, перейдем к знакомству с собственно языком.

# Дополнительные сведения

## Комментарии

Символ комментария //, действующий до конца строки, является теперь составной частью языка, а не только расширением Microsoft, как это было для языка C.

## Ключевые слова

Ключевые слова, характерные только для C++, приведены в табл. П1.1.

***Таблица П1.1.*** *Ключевые слова языка C++*

| | | | |
|---|---|---|---|
| bad_cast | explicitfalse | protected | try |
| bad_typeid | finally | public | type_info |
| bool | friend | reinterpret_cast | typeid |
| catch | inline | signed | typename |
| class | mutable | static_cast | using |
| const | namespace | template | virtual |
| const_cast | new | this | volatile |
| delete | operator | throw | xalloc |
| dynamic_cast | private | true | |
| except | | | |

## Константы

Как вы помните, для определения константы в языке C использовалась директива препроцессора **#define**, что было достаточно удобно. Отрицательной стороной такого определения является то, что при этом нельзя указать тип, которому принадлежит константа. В C++ введено новое ключевое слово **const**, позволяющее создавать константы. При этом явным образом указывается тип, который компилятор в дальнейшем может проверить.

```
const float PI = 3.1415926;
```

Константную переменную нельзя модифицировать, и поэтому попытка изменить ее вызывает ошибку во время компиляции. Константой можно объявить и указатель. При этом изменять нельзя только значение самого указателя, а объект (переменную), на который он указывает, менять можно:

```
int  nValue;
const int *ptrVar = &nValue;
*ptrVar += 7; // Такое действие допустимо
ptrVar++;     // А вот так нельзя изменять значение самого указателя
```

Ключевое слово **const** можно применить и к функции:

```
int anyFunc(...) const;
...
int anyFunc(...) const
(
...
)
```

В этом случае функция считается "только для чтения", и нельзя модифицировать объект, для которого она вызывается.

**Примечание**

Ключевое слово **const** необходимо указывать и при объявлении и при определении функции.

## Блочные объявления

В C++ допускаются объявления внутри блоков и после программных операторов, что позволяет определять (объявлять) объект в том месте программы, где он используется первый раз — во многих случаях это улучшает читабельность программы.

```
void myFunc()
{
  int nFirstVar;
...
  nFirstVar++;
  // Другие операторы функции
...
  float fSecondVar = 7.0;
...
  // Остальные операторы функции
}
```

Кроме того, можно определить и инициализировать переменную непосредственно внутри формального описания управляющей структуры:

```
for(int iCount = 0; iCount < MAX_COUNT; iCount++) {...}
```

**Примечание**

Определенная таким образом переменная *не входит* в блок управляющей структуры и поэтому действует до конца всего блока, в котором используется эта структура. Поэтому следующая конструкция недопустима:

```
for(int iCount = 0; iCount < MAX_COUNT1; iCount++)
{
...
}
for(int iCount = 0; iCount < MAX_COUNT2; iCount++)
{
...
}
```

## Ссылки

Для того чтобы функция могла вернуть более одного значения, в языке C необходимо было передать ей в качестве параметра указатель на те переменные, значения которых требуется получить. Зачастую это служило источником достаточно большого количества ошибок, особенно у начинающих программистов, когда вместо адреса в функцию передавалось значение переменной. В C++ можно определить *ссылку* на переменную или объект класса. Ссылка содержит адрес объекта, но используется так, как будто она представляет сам объект. Для объявления ссылки используется оператор **&**:

```
double dVar;          // Определяет переменную типа double
double &rdVar = dVar; // Определяет ссылку типа double
```

Такая переменная при определении должна быть обязательно проинициализирована.

Наиболее очевидным применением ссылок является обеспечение передачи функции адреса вместо самой переменной (иногда это называется передачей параметра по ссылке), что позволяет избежать копирования параметров, передаваемых функции в качестве аргументов, в стек:

```
struct BigStruct{
  int array[1000];
}stObject;
void func1(BigStruct bs);  // В качестве параметра передается
                           // значение
void func2(BigStruct *bs); // В качестве параметра передается
                           // указатель
void func3(BigStruct &bs); // В качестве параметра передается ссылка
```

Вызов функции func1 связан с копированием в стек всех 4000 байт (для версии Visual C++ 6.0) структуры BigStruct. Ситуация меняется в двух других случаях, когда передается только адрес структуры. Они различаются тем, что

в последнем случае мы работаем с этим адресом, как будто это непосредственно переменная, и дополнительных действий по снятию адреса не требуется.

## Имена перечислений, структур и объединений

В C++ имя перечисления, структуры или объединения является именем типа. Это упрощает нотацию, поскольку не нужно использовать ключевые слова **enum**, **struct** и **union** перед именем соответствующего типа. Таким образом, в C++ выражения вида

```
struct Fruits{
// Компоненты структуры
};
Fruits stApple; // Определение переменной типа Fruits
```

являются допустимыми.

## Распределение памяти

В C++ встроены специальные операторы для управления памятью — **new** и **delete**. Они чрезвычайно удобны для динамического создания и удаления переменных, массивов и объектов классов и заменяют функции malloc, alloc и free, используемые в C.

Определены три способа использования оператора **new**:

```
new t_name;             // Выделяется память под объект типа t_name
new t_name(initValue); // Выделяется память под объект типа
                        // t_name с одновременной инициализацией
new type_name[size];    // Выделяется память под size объектов
                        // типа t_name
```

Для первого способа существует эквивалентная запись:

```
new (type_name);
```

Этот оператор запрашивает у системы память, необходимую для размещения объекта, и возвращает на нее указатель.

Для удаления объекта, созданного с помощью оператора **new,** необходимо выполнить оператор **delete**, который может использоваться двумя способами:

```
delete vObject;    // Освободить память, занятую объектом
delete [] vObject; // Освободить память, занятую массивом объектов
```

**Примечание**

Применение оператора **delete** к неинициализированному указателю является ошибкой, за которой должен следить сам программист.

## Встраиваемые функции

Спецификатор **inline** предназначен для использования перед объявлением функции, чтобы компилятор помещал ее код непосредственно в место вызова:

```
// Код этой функции будет подставлен в каждом месте вызова
inline void swap(float &a, float &b);
```

При таком объявлении функции в *каждом* месте вызова появится ее локальная копия. В языке C этого же эффекта можно было добиться при помощи директивы препроцессора **#define**, неправильное использование которой зачастую служит источником ошибок.

**Примечание**

Использование спецификатора **inline** не больше чем просьба к компилятору, которую он может удовлетворить или нет. Кроме того, стандарт оговаривает случаи, когда эта просьба никогда не будет удовлетворена:

- если это рекурсивная функция;
- если имеется вызов из определенной ранее встроенной функции;
- если в функции имеются циклы.

В C++ есть еще один способ встраивания функций, но его мы рассмотрим после того, как познакомимся с классами.

## Перегрузка функций

В C каждая функция должна иметь уникальное имя. В C++ тип функции так же важен, как и ее имя. *Перегрузка* позволяет использовать одно и то же имя для реализации разных функций. Это означает, что можно иметь несколько функций с одним и тем же именем, но с различным числом и типами аргументов:

```
int sort(char **str);       // Сортирует массив строк
int sort(int*, int);        // Сортирует массив целых чисел
int sort(double*, double); // Сортирует массив вещественных чисел
```

Поскольку число и типы аргументов в каждой функции отличаются друг от друга, то C++ без труда автоматически выбирает нужную функцию.

Перегрузка функций позволяет программисту маскировать различия между типами аргументов. Она также является одной из причин замены макросов **#define** на встраиваемые функции. Макросы не имеют типа и поэтому не могут быть перегружены, тогда как встраиваемые функции позволяют реализовать этот механизм.

## Задание параметров функции по умолчанию

В дополнение к возможностям, имеющимся в C, в языке C++ появилась еще одна — теперь при объявлении или определении можно задавать значения некоторых параметров функции по умолчанию:

```
float Multi(float fFirst, float fSecond,
            float fThird = 1, float gFourth = 1)
{
  return (fFirst * fSecond * fThird * gFourth);
}
```

При вызове этой функции такие параметры можно не указывать:

```
...
float fResult;
fResult = Multi(2, 3, 4, 5); // Произведение четырех чисел:
                             // fResult = 120
fResult = Multi(2, 3, 4);    // Произведение трех чисел:
                             // fResult = 24
fResult = Multi(2, 3);       // Произведение двух чисел:
                             // fResult = 6
...
```

Эта возможность чрезвычайно удобна в том случае, когда при вызовах функции часто используются одни и те же значения некоторых параметров.

Здесь следует учитывать одно ограничение: *присваивание значений, заданных по умолчанию, должно выполняться справа налево*. Поэтому следующая запись недопустима:

```
double func(double dArg1 = 7, double dArg2); // Так нельзя
```

Если необходимо указать значение по умолчанию для dArg1, то и dArg2, в данном случае, тоже нужно присвоить значение по умолчанию.

## Операции

В C++ определены следующие три дополнительные операции:

| | |
|---|---|
| :: | Уточнение области действия |
| .* | Прямое обращение к компоненту класса |
| ->* | Косвенное обращение к компоненту класса |

Для устранения конфликта имен используется новая операция уточнения области действия — **::**. В языке C, если локальная и глобальная переменные имели одинаковые имена, то доступ к глобальной переменной был закрыт. В C++ доступ к ней возможен путем применения операции **::**.

```
int iVariable; // Глобальная переменная
void myFunc()
{
  int iVariable; // Локальная переменная
...
  ::iVariable++; // Увеличиваем значение глобальной переменной
...
  iVariable--;   // Уменьшаем значение локальной переменной
...
}
```

Эта операция может применяться также к статическим (**static**) компонентам структур и классов.

Операция прямого обращения к компоненту класса **.*** устанавливает указатели так, что в результате выполнения операции они указывают на компоненты данного класса:

```
(ptrToObject.*memberFunc)(7); // Вызывает метод memberFunc для
                        // объекта, на который указывает ptrToObject
```

Операция косвенного обращения к компоненту класса **->*** устанавливает указатели так, что в результате выполнения операции они указывают на указатели компонентов данного класса:

```
(ptr2Object->*ptr2MemberFunc)(7);// Вызывает метод ptr2MemberFunc
                                 // для объекта, на который
                                 // указывает ptr2Object
```

## Библиотеки потоков

В язык C++ включены дополнительные предопределенные потоки. Для терминального и файлового ввода/вывода предусмотрены три стандартных потока: **cin**, **cout** и **cerr**. Эта возможность позволяет легко настроить операции ввода/вывода в соответствии с требованиями приложения:

```
...
float sm = 0;
cout << "сантиметры = "; // Выводим строку в поток cout
cin >> sm;               // Получаем значение из потока cin
cout << sm << "соответствуют " << sm / 2.54 << "дюймам";
...
```

В результате выполнения последней строки приведенного фрагмента на экран будет выведена, например, следующая строка:

```
17.78 sm соответствуют 7 дюймам
```

После того как мы познакомились с усовершенствованиями, внесенными в язык C++, можно переходить к фундаментальным новшествам этого мощнейшего и очень удобного языка программирования.

Самое существенное улучшение по сравнению с языком C касается концепции объектно-ориентированного программирования. Это

- более четкий контроль структуры программы с целью ее декомпозиции;
- возможность создания новых типов данных со своими собственными специализированными операторами;
- средства для создания повторно используемого кода.

Теперь перейдем к рассмотрению применительно к языку C++ трех ключевых идей объектно-ориентированного программирования: инкапсуляции, наследования и полиморфизма и теснейшим образом связанного с ними понятия класса.

# Классы

Классы представляют собой первичный механизм, позволяющий реализовать скрытие данных, абстракцию данных и наследование — те самые свойства, которые делают C++ объектно-ориентированным языком.

Классы C++ предусматривают создание расширенной системы предопределенных типов. Каждый тип класса представляет уникальное множество объектов и операций над ними, а также операций, используемых для создания, манипулирования и уничтожения таких объектов. Могут быть объявлены производные классы, наследующие компоненты одного или нескольких базовых (порождающих) классов.

Но обо всем по порядку.

## Инкапсуляция

Для определения классов используются три ключевых слова: **struct**, **union** и **class**. Каждый класс, определенный посредством одного из этих ключевых слов, включает в себя функции — называемые *методами* или *компонентными функциями* (*member function*), и данные — называемые *элементами данных* (*class members*). Так же, как и для структур и объединений, каждому классу присваивается некоторое имя (лучше содержательное), которое становится идентификатором нового типа данных и может использоваться для объявления *объектов* (или *экземпляров*) этого типа (рис. П1.1).

Описание класса начинается с ключевого слова <ключ_класса>, в качестве которого можно использовать **struct**, **union**, **class** и непосредственно за которым следует имя класса — <имя_класса>:




**Рис. П1.1.** Инкапсуляция элементов и методов класса

```
ключ_класса имя_класса <: базовый_список>
{
// Здесь объявляются как методы, так и элементы класса
<Компоненты_класса>
};
```

Необязательный *базовый_список* содержит базовый класс (или классы), из которого *имя_класса* заимствует элементы и методы. Необязательный список компонентов объявляет элементы и методы класса и, при необходимости, спецификаторы доступа.

После того как новый уникальный тип данных объявлен, можно определять объекты данного типа (класса), а также объекты, являющиеся производными от него:

```
class Sample{
  unsigned uData;      // элемент данных
  unsigned readData(); // метод (компонентная функция)
...
};
Sample  clInst1,              // Переменная типа Sample
       *pclInst2 = &clInst1,  // Указатель на Sample
        arInst[7];            // Массив элементов типа Sample
```

Допускаются также неполные объявления класса:

```
class Simple;
```

которые разрешают некоторые ссылки к именам классов до того, как классы будут полностью определены.

Обращаться к компонентам класса можно, указав имя объекта — имя элемента или метода, и разделив эти имена операцией "." (точка), или, при использовании указателя — операцией "->":

```
clInst1.uData = 7;
unsigned uRes = pcInst2->readData();
```

**Примечание**

Несмотря на то, что приведенные выражения правильны, при компиляции будет выдано сообщение об ошибке, поскольку в соответствии с принципом скры-

тия данных, установленным по умолчанию, доступ и к элементу данных, и к методу невозможен.

Чуть раньше я уже говорил о том, что в C++ возможно встраивание функций посредством спецификатора **inline**, который является не больше, чем "просьбой" к компилятору, которую тот может как удовлетворить, так и проигнорировать.

Для методов класса возможно неявное выражение такой просьбы:

```
class MyClass{
  ...  // Объявления переменных и методов
  float  fVar;
  float* func(); {return fVar;} // inline по умолчанию
};
```

что эквивалентно следующей записи:

```
inline float* MyClass::func(); {return fVar;}
```

Очевидно, что для встраиваемых функций нельзя говорить об их адресе.

В отличие от обычных функций языка C (да и C++ тоже), *нестатические* методы классов в C++ имеют одну неявную локальную переменную. Эта переменная даже имеет специальное имя — **this**. И хотя на практике она используется достаточно редко, однако в некоторых случаях наличие этой переменной бывает полезно, в чем мы с вами убедимся при знакомстве с библиотекой MFC. Фактически же **this** — это указатель на текущий объект класса.

Вы, наверное, обратили внимание на то, что в последнем абзаце я употребил слово "нестатического", которое требует некоторых пояснений.

При объявлении данных и методов класса может быть использован спецификатор класса памяти **static**. Вспомните, что языке C этот спецификатор ограничивает область видимости переменной или функции одним модулем — налицо попытка введения некоторого ограничения доступа. В C++ компоненты класса, перед которыми стоит это ключевое слово, называются *статическими* и обладают свойствами, отличными от свойств "обычных" элементов. Для наглядности рассмотрим небольшой пример (рис. П1.2):

```
class MyClass{
  int nVarUnStatic;      // обычная переменная
  static int nVarStatic; // статическая переменная
  // объявления других переменных и методов
};
MyClass clInstance1, *pclInstance2;
```

Таким образом, если для нестатических (обычных) элементов для каждого объекта класса создается отдельная копия, и обращение, например, к элементу nVarUnStatic осуществляется по обычным правилам:

```
clInstance1.nVarUnStatic++;     // правильно
pclInstance1->nVarUnStatic++;   // правильно
```

то для статического элемента можно просто задать имя класса совместно с модификатором области действия:

```
MyClass::nVarStatic++;          // правильно
```

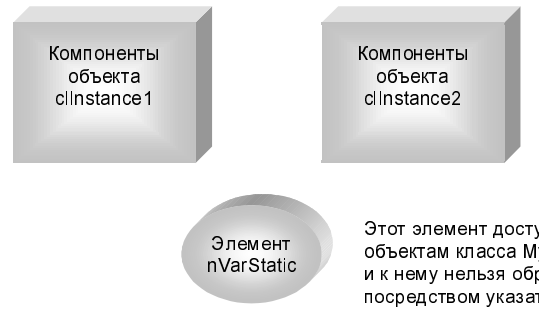



**Рис. П1.2.** Распределение памяти для статических и нестатических переменных

Однако это еще не все. В отличие от нестатических элементов, память под которые выделяется при создании объекта класса, для статических должно быть отдельное определение, отвечающее за распределение памяти и инициализацию, например, такое:

```
static MyClass::nVarStatic;
```

Основное назначение статических элементов состоит в том, чтобы отслеживать данные, общие для всех объектов класса, а также для уменьшения числа видимых глобальных имен.

По аналогии со статическими элементами данных, статические методы не ассоциируются с каким-либо конкретным объектом и, соответственно, не имеют указателя **this**. Такие методы вызываются просто по имени класса, например, так:

```
MyClass::func();
```

Поскольку статический метод не имеет указателя **this**, из него нельзя обращаться к элементам данных, не задавая в явном виде конкретный объект. В то же время допустимо непосредственное обращение к статическим компонентам класса. Осталось сказать еще о двух ограничениях:

- нельзя определять нестатический метод с тем же именем и тем же набором параметров, что и у статического метода класса;
- статический метод не может быть объявлен как виртуальный (о виртуальных методах мы поговорим чуть позже).

**Примечание**

Для элементов данных и методов класса недопустимо использование каких-либо других спецификаторов класса памяти, кроме **static**.

В языке C++ разграничение доступа получило свое логическое продолжение. Рассмотрим этот вопрос более подробно.

## Разграничение доступа (скрытие данных и методов)

Определив структуру, объединение или класс, можно создавать объекты этого класса (структуры, объединения) и манипулировать ими, используя методы. При этом некоторые данные и методы, инкапсулированные в одном классе, можно сделать недоступными вне реализации класса, а другие могут быть доступны из любого места программы.

Элементы структур, объединений и классов получают атрибуты доступа либо по умолчанию, либо при использовании какого-либо из трех спецификаторов доступа: **public**, **protected** или **private**, которые имеют следующий смысл:

| | |
|---|---|
| **public** | Компонент доступен для любой функции |
| **protected** | Компонент может быть использован методами или "друзьями" класса, в котором он объявлен, а также производных от него, но только в объектах производного класса |
| **private** | Компонент может быть использован только методами или "друзьями" класса, в котором он объявлен |

Определены следующие значения доступа по умолчанию для компонентов классов, структур и объединений:

- Класс, определенный с помощью ключевого слова **struct** — это класс, в котором все элементы по умолчанию являются общедоступными (**public**), и имеется возможность изменять этот уровень доступа.
- Класс, определенный с помощью ключевого слова **union** — это класс, в котором все элементы по умолчанию являются общедоступными (**public**), и этот уровень доступа не может быть изменен.
- Класс, определенный с помощью ключевого слова **class** — это класс, в котором все элементы по умолчанию являются частными (**private**), и имеется возможность изменять уровень доступа.

Спецификатор доступа, заданный явно или по умолчанию, остается действительным для всех последующих объявлений компонентов, пока не встретится какой-либо другой спецификатор доступа.

```
class MyClass{
  ...                  // Объявления переменных и методов — по умолчанию
private
```

```
protected:            // Изменяем атрибуты доступа
  float  fVar;
public:               // Еще раз изменяем атрибуты доступа
  float* func(); {return fVar;}
};
```

Спецификаторы доступа могут быть перечислены и сгруппированы в любой удобной для вас последовательности.

Теперь вам стало понятно, почему в примере на стр. 580 компилятор выдаст сообщение об ошибке — доступ к частным компонентам класса (что установлено по умолчанию) запрещен. Доступ к элементу данных uData может быть организован, например, так:

```
class Sample{
// элемент данных
  unsigned uData;
public:                  // изменяем права доступа
// методы (компонентные функции)
  unsigned writeData(unsigned uArg);
  unsigned readData();
...
} clInst1, *pcInst2;
```

Тогда

```
clInst1.uData = 7;        // по-прежнему ошибка!
unsigned uPar = 7;
pcInst2->writeData(uPar); // теперь правильно
```

Еще раз повторю. Методы и данные, определенные после ключевого слова **public**, представляют собой интерфейс класса — они единственные доступны вне класса. Компоненты, определенные после остальных спецификаторов доступа, относятся к внутренней реализации и вне класса недоступны. Различие между компонентами класса, описанными после ключевых слов **protected** и **private**, сказываются только при наследовании, которое мы рассмотрим несколько позже.

Рекомендуется всегда *явно* определять права доступа к компонентам класса.

Иногда ограничение на доступ к частным и защищенным компонентам класса только посредством методов класса оказывается неоправданно жестким. Бывают случаи, когда необходимо разрешить такой доступ.

## Друзья классов

В языке C++ предусмотрена возможность обеспечить доступ к частным и защищенным компонентам извне класса при помощи ключевого слова

**friend**, которое говорит о том, что следующая за ним функция является "другом" данного класса и имеет полный доступ ко всем его компонентам, не являясь в то же время методом этого класса. Такие дружественные функции не зависят от положения в классе и спецификаторов доступа и ничем другим от обычных методов не отличаются.

```
class MyClass
{
private:
  int nVar;
  // Дружественная функция
  friend void friendFunc(MyClass &clArg, int nArg);
public:
  // Метод класса
  void memberFunc(int nArg);
};
// Определения функций
void friend friendFunc(MyClass &clArg, int nArg)
                {clArg.nVar = nArg;}
void memberFunc(int nArg) {nVar = nArg;}
...
// Создаем объект класса
MyClass clObj;
// Обращаемся к частному компоненту класса
friendFunc(clObj, 7);
memberFunc(7);
...
```

Дружественные функции класса дают программисту сбалансированный подход к принципу инкапсуляции данных: если методы доступа влекут за собой слишком большие дополнительные издержки, то иногда проще дать некоторой внешней функции права непосредственного доступа к компонентам класса. Более того, одна и та же функция может быть дружественной для нескольких классов.

В качестве дружественной функции может выступать и некоторый метод другого класса:

```
class MyClass1
{
...
  // Метод memberFuncMyClass2 класса MyClass2 объявляется дружественным
  // классу MyClass1
  friend void MyClass2::memberFuncMyClass2(...);
...
};
```

По аналогии с функциями и методами можно объявить дружественным весь класс. При этом все методы такого дружественного класса смогут обращаться ко всем компонентам класса:

```
class MyClass
{
...
  // Класс MyFriend объявляется дружественным классу MyClass
  friend class MyFriend;
...
};
```

И еще два замечания:

1. "Дружба" классов не транзитивна. Другими словами, если класс MyClass1 друг класса MyClass2, а MyClass2 друг MyClass3, то это не означает, что MyClass1 друг MyClass3. Однако "дружба" наследуется, но об этом более подробно после знакомства с наследованием.
2. "Дружба" классов односторонняя, т. е. если класс MyFriend объявлен другом MyClass, то это не означает, что MyClass имеет доступ к частным и защищенным компонентам класса MyFriend. Таким образом, права для доступа к классу может предоставить только автор класса, а не программист, написавший функцию или некоторый другой класс.

Пойдем дальше. Как правило, перед использованием объект должен быть инициализирован. В C этот процесс выполнялся либо автоматически (для глобальных и статических элементов), либо путем определения специальной функции, которая вызывается сразу после создания объекта, до его использования. Аналогичный подход можно использовать и в C++. Однако в нем предусмотрен более гибкий способ.

## Конструкторы и деструкторы

Среди всех методов любого класса выделяются два, определяющие способы, какими создаются, инициализируются, копируются и разрушаются объекты класса. Речь идет о *конструкторах (constructor)* и *деструкторах (destructor)*, которые наряду с характеристиками обычных методов обладают и некоторыми уникальными свойствами:

- Конструктор всегда имеет то же имя, что и сам класс; это же относится и к деструктору, которому, однако, предшествует символ "~" (тильда).
- Не имеют объявлений значений возврата (даже **void**).
- Не могут быть унаследованы, хотя производный класс может вызывать конструкторы и деструкторы базового класса.
- Конструкторы, как и большинство функций C++, могут иметь аргументы по умолчанию или использовать списки инициализации элементов.

- Деструкторы могут иметь атрибут **virtual**, а конструкторы — нет.
- Нельзя работать с адресами конструкторов и деструкторов.
- Если конструкторы или деструкторы не были определены явно, они генерируются компилятором.
- Конструктор нельзя вызывать как обычную функцию; вызов деструктора возможем с полностью заданным именем.
- При определении и разрушении объектов вызов конструкторов и деструкторов осуществляется автоматически.
- Конструкторы и деструкторы при необходимости выделения объекту памяти могут выполнять неявные вызовы операций **new** и **delete**.

Прежде чем переходить к более подробному знакомству с этими специальными методами классов, рассмотрим правила, по которым происходит создание/уничтожение объектов:

- Конструкторы и деструкторы автоматически запускаются всякий раз при создании и разрушении объектов.
- Глобальные и локальные статические объекты создаются в начале выполнения программы — до того, как управление попадает в main (или WinMain). Уничтожение таких объектов происходит в момент завершения программы в результате возврата из функции main (WinMain).
- Локальные объекты создаются, когда программа встречает их определение, и уничтожаются при выходе из функции.
- Конструктор объекта, память которому выделяется в области Heap (Куча) (посредством операции **new**), вызывается автоматически при вызове **new**; такой объект разрушается при явной передаче объекта в оператор **delete**.

**Примечание**

Для объектов, память под которые выделяется вызовом malloc (или аналогичной функции), конструктор не вызывается вообще, поскольку в этом случае функция просто не знает, какой конструктор надо запустить (эта функция не принимает никакой информации о типе).

Рассмотрим простой пример, для которого воспользуемся классической функцией main, чтобы избежать дополнительных действий, необходимых для Windows-приложений:

```
// Простой пример,
// демонстрирующий последовательность вызовов конструкторов
   и деструкторов
#include <stdio.h>
#include <string.h>
// Объявляем класс
class Sample{
```

```
  private:  // К этим данным доступ возможен только посредством
              методов класса
    char *strName;
  public:  // К этим данным доступ возможен из любого места программы
    // Определяем простой конструктор
    Sample(char *str){
      strName = new char[strlen(str) + 1];
      strcpy(strName, str);
      printf("Entry in constructor for %s\n", strName);
    }
    // Определяем деструктор
    ~Sample(){
      printf("Entry in destructor for %s\n", strName);
      delete strName;
      strName = 0;
    }
};
// Определяем первый глобальный объект
Sample global1("global #1");
// "Глупая" функция, иллюстрирующая создание автоматического объекта
void SillyFunc(){
  // Определяем объект, локальный для данной функции
  Sample fnAuto("automatic function");
  printf("Function SillyFunc()\n");
}
int main(int argc, char* argv[])
{
  // Определяем первый локальный объект
  Sample auto1("automatic #1");
  printf("Begin main()\n");
  SillyFunc();
  // Определяем второй локальный объект
  Sample auto2("automatic #2");
  printf("Continue main()\n");
  return 0;
}
// Определяем второй глобальный объект
Sample global2("global #2");
```

Результат выполнения программы представлен на рис. П1.3, из которого ясно видна последовательность вызовов конструкторов и деструкторов для глобальных и автоматических объектов.

Пойдем дальше. Конструкторы подчиняются тем же правилам перегрузки, что и любые другие функции. Один и тот же класс может иметь несколько конструкторов, если каждый из них имеет свой собственный список аргу-

ментов. При определении объекта будет запущен тот конструктор, с которым совпадает заданный список аргументов.

**Примечание**

Единственным ограничением на аргументы конструктора является то, что в качестве них нельзя использовать объект того же класса. Вместо этого можно использовать ссылку на объект.

Некоторые конструкторы играют особую роль.

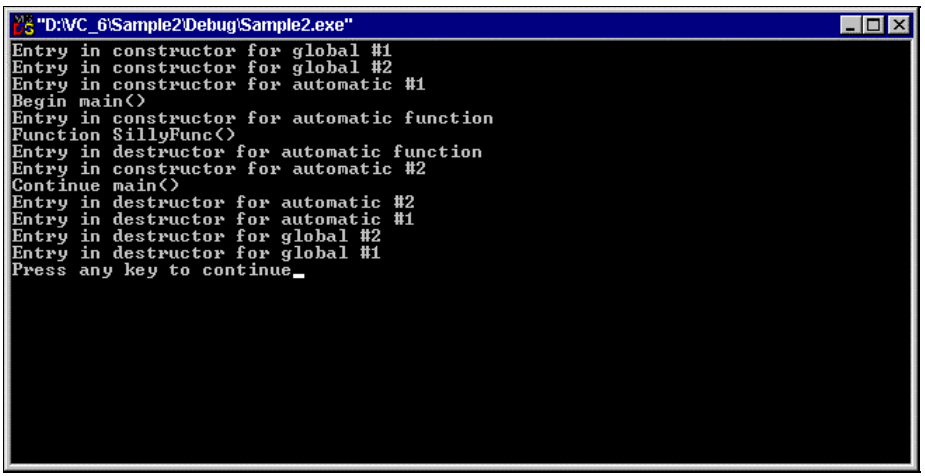



**Рис. П1.3.** Результат выполнения программы Sample

## Конструктор по умолчанию

Конструктор по умолчанию не имеет аргументов и используется, когда объект создается без инициализации его данных:

```
class MyClass{
public:
  MyClass(){}
...
// Остальные данные и методы
};
```

Без такого конструктора объекты не могли бы быть определены без задания инициализирующих значений. Если конструктор по умолчанию не объявлен явно, то компилятор автоматически назначает его. Однако, как и все остальное, лучше делать это самому. Он вызывается при таких определениях, как

```
MyClass clObj;
```

Как и любая другая функция, конструкторы могут иметь аргументы со значениями, заданными по умолчанию. При этом следует избегать неоднозначностей. Например, в случае определения класса

```
class MyClass{
public:
  MyClass();
  MyClass(int nArg = 0);
...
// Остальные данные и методы
};
```

может возникнуть неопределенность при вызове конструктора

```
MyClass clObj1(7); // можно — однозначно вызывается
                   // MyClass::MyClass(int)
MyClass clObj2;    // нельзя; неоднозначно задано, какой конструктор
                   // следует вызывать — MyClass::MyClass(int)
                   // или MyClass::MyClass(int nArg = 0)
```

## Конструктор копирования

Аргументом конструктора может быть и сам объект класса. Конструктор в этом случае будет иметь следующий прототип:

```
MyClass::MyClass(MyClass &);
```

Такие конструкторы запускаются при копировании данных "старого" объекта во вновь создаваемый:

```
MyClass clObj1(7);
MyClass clObj1 = clObj2;
```

Если вы не определили для класса конструктор копирования, то компилятор сделает это за вас. Однако, как обычно, лучше это делать самому.

## Несколько слов о деструкторах

В отличие от конструкторов, деструктор класса не имеет аргументов и не может быть перегружен. Как было наглядно продемонстрировано в примере на рис. П1.3, деструкторы вызываются строго в обратной последовательности вызова соответствующих конструкторов. Они вызываются автоматически при выходе объекта из блока, в котором были определены. Единственным исключением из этого общего правила является случай, когда объект создается динамически из "кучи", путем вызова оператора **new**. В этом случае для корректного удаления объекта необходимо явно выполнить оператор **delete**, который и вызовет необходимый деструктор.

Таким образом, можно подвести предварительные итоги. Понятие класса C++ устанавливает четко определенный интерфейс, который помогает разрабатывать, выполнять, поддерживать и сопровождать программы. Концепция класса органично связана с идеей абстракции данных, при которой данные не связываются ни с каким физическим воплощением, а скорее определяется в терминах методов (функций), выполняемых над ними. С этой точки зрения данные и методы рассматриваются как равные, независимые партнеры.

## Наследование

Следующим ключевым моментом объектно-ориентированного программирования, нашедшим свое воплощение в C++, является возможность создания иерархической структуры классов, которая базируется на наследовании — некоторый класс может передавать свои компоненты другому классу. При этом первый класс называется *базовым*, а второй — *производным* классом (или подклассом). Имя базового класса указывается непосредственно после имени производного, до открывающей фигурной скобки:

```
class BaseClass
{
 // Компоненты базового класса
};
class SubClass : public BaseClass
{
 // Компоненты производного класса
};
```

В данном случае производный класс SubClass наследует все компоненты базового, а также содержит любые, определенные непосредственно в нем самом компоненты.

Обратите внимание на ключевое слово **public** перед именем базового класса, которое определяет тип наследования — в данном случае общедоступный. Помимо такого способа можно также задавать частный (**private**) тип наследования. Разные типы влияют на доступность компонентов базового класса в производном. Чтобы разобраться в этом вопросе, рассмотрим пример:

```
class BaseClass
{
public:
  // Общедоступные компоненты базового класса
  int nPublBase;
  void funcBase();
protected:
  // Защищенные компоненты базового класса
  int nProtBase;
```

```
private:
  // Частные компоненты базового класса
  int nPrivBase;
};
// class SubClass : public BaseClass // public-наследование
class SubClass : private BaseClass   // private-наследование
{
public:
  // Общедоступные компоненты производного класса
  int nPublSub;
  void funcSub();
protected:
  // Защищенные компоненты производного класса
  int nProtSub;
private:
  // Частные компоненты производного класса
  int nPrivSub;
};
void main(int argc, char* argv[])
{
  BaseClass clBase;
  SubClass  clSub;
  clBase.nPublBase = 7;
  clBase.funcBase();
// clSub.nPublBase = 7; // Ошибка, т. к. для класса SubClass даже
// clSub.funcBase();    // public-компоненты базового класса
                        // являются private-компонентами
  clSub.nPublSub = 77;
  clSub.funcSub();
}
void BaseClass::funcBase()
{
  // Имеется доступ ко всем компонентам "своего" класса BaseClass
  nPublBase = 1;
  nProtBase = 2;
  nPrivBase = 3;
}
void SubClass::funcSub()
{
  // Имеется доступ только к public и protected-компонентам
  // базового класса BaseClass, которые для производного класса
  // являются private-компонентами и доступны только для методов класса
  nPublBase = 1;
  nProtBase = 2;
// nPrivBase = 3; // Ошибка; здесь нет доступа к private-компонентам
```

```
  // Имеется доступ ко всем компонентам "своего" класса SubClass
  nPublSub  = 4;
  nProtSub  = 5;
  nPrivSub  = 6;
  funcBase();
}
```

Что интересного в этом примере? Прежде всего, обратите внимание, что компонент базового класса, наследуемый как **public**, сохраняет тот же самый тип доступа.

Несколько по-другому дело обстоит при типе наследования **private**. Как вы конечно же помните, частный компонент класса доступен только другим компонентам и друзьям класса. Поэтому вполне логичным выглядит то, что такие частные компоненты не доступны в производном классе. Ведь в противном случае программист имел бы возможность получить доступ к таким компонентам, реализовав в производном классе необходимые методы, а это противоречит самой идеологии инкапсуляции.

Однако часто необходимо из производного класса обеспечить доступ к некоторым компонентам базового. Чтобы не пришлось объявлять такие компоненты общедоступными (**public**), т. е. полностью "раскрывать" их, в языке C++ и введена специальная категория доступности — защищенная (**protected**). Компоненты, которые имеют такой уровень доступа, заблокированы для всех частей программы, за исключением компонентов самого класса и производного от него.

Осталось только привести сводку правил наследования доступа:

| Тип наследования доступа | Доступность в базовом классе | Доступность компонентов базового класса в производном |
|---|---|---|
| public | public<br>protected<br>private | public<br>protected<br>private |
| private | public<br>protected<br>private | private<br>private<br>недоступны |

По умолчанию установлен тип наследования доступа — **private**.

Чтобы закончить с вопросом правил наследования доступа, осталось сказать, что в производном классе можно изменять доступность отдельных компонентов базового класса. Например, если в объявление производного класса SubClass из предыдущего примера внести следующее дополнение

```
class SubClass : BaseClass   // private-наследование по умолчанию
{
```

```
public:
  // Общедоступные компоненты производного класса
  BaseClass::nPublBase; // Конкретно объявляем как public
  int nPublSub;
  void funcSub();
...
};
```

то из любого места программы становится возможен доступ к этой компоненте базового класса:

```
clSub.nPublBase = 7;
```

Производный класс сам может служить базовым, т. е. от него можно образовывать новые классы. В этом случае полученный класс будет включать в себя элементы всех его базовых классов. От одного базового класса можно образовывать несколько производных. Таким образом, возможна древовидная структура иерархии классов (рис. П1.4).

Схема размещения в памяти объекта простого производного класса представлена на рис. П1.5.

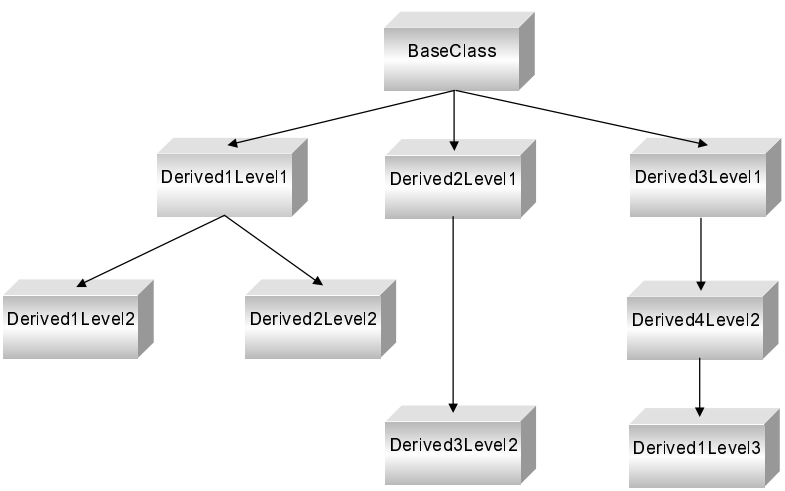



**Рис. П1.4.** Древовидная иерархия классов

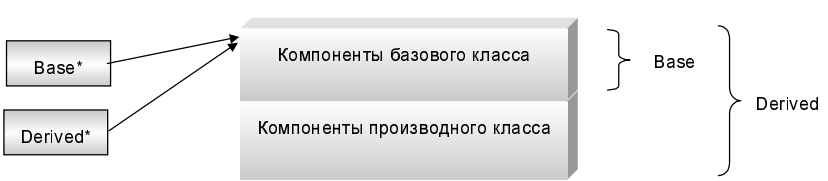



**Рис. П1.5.** Схема размещения в памяти объекта простого производного класса

Во всех рассмотренных примерах производный класс образовывался от одного базового класса — так называемое *единичное* наследование. Однако механизм наследования в C++ более развит и включает наряду с единичным и *множественное* наследование, при котором производный класс образуется сразу от нескольких базовых, что позволяет создавать достаточно сложные иерархии классов (рис. П1.6):

```
class Base1
{
 // Компоненты первого базового класса
};
class Base2
{
 // Компоненты второго базового класса
};
class Derived : public Base1, private Base2
{
 // Компоненты производного класса
};
```

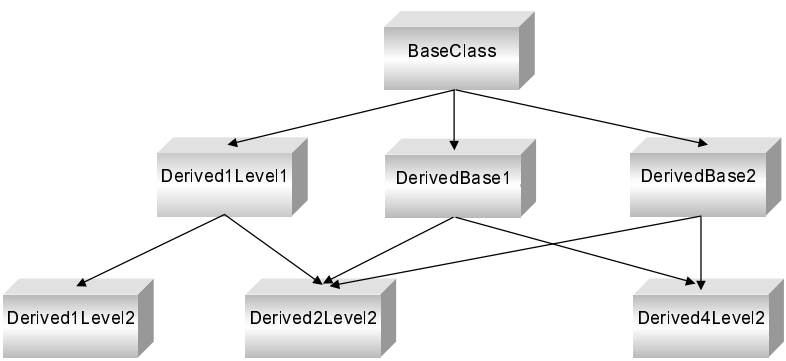



**Рис. П1.6.** Графовая иерархия классов

В объявлении класса может быть записано любое количество базовых классов, перечисленных через запятую. Порядок перечисления базовых классов произвольный. Он влияет только на последовательность вызова конструкторов, которая является следующей:

1. Сначала вызываются конструкторы всех виртуальных базовых классов; если их несколько, то конструкторы вызываются в порядке их наследования (виртуальные классы мы рассмотрим ниже).
2. Затем вызываются конструкторы невиртуальных базовых классов в порядке их наследования.
3. И, наконец, вызываются конструкторы всех компонентных классов.

Как обычно, последовательность вызова деструкторов является обратной относительно последовательности вызовов конструкторов.

Поскольку множественное наследование является более сложной конструкцией по сравнению с единичным, то вполне очевидно, что при этом возникают и некоторые проблемы.

В случае множественного наследования вполне вероятно, что один и тот же класс может унаследовать несколько объектов с одним и тем же именем:

```
class Base1
{
  int nObj;
 // Остальные компоненты первого базового класса
};
class Base2
{
  int nObj;
 // Остальные компоненты второго базового класса
};
class Derived : public Base1, public Base2
{
 // Компоненты производного класса
} clDer;
```

В данном случае мы имеем два объекта clDer.nObj с одним и тем же именем, а значит, обращаться к ним просто по имени нельзя. В этом случае выход заключается в использовании оператора разрешения контекста и полных имен объектов:

```
clDer.BaseClass1::nObj++;
clDer.BaseClass2::nObj++;
```

Аналогично решается проблема одинаковых имен и при использовании наследования через промежуточный класс:

```
class BaseClass
{
  int nBaseObj;
};
class FirstBaseClass : public BaseClass
{
  int nFirstObj;
};
class SecondBaseClass : public BaseClass
{
  int nSecondObj;
};
```

```
class DerivedClass : public FirstBaseClass, public FirstBaseClass
{
  long nDerivedObj;
  // Остальные компоненты производного класса
} clDer;
...
clDer.FirstBaseClass::nObj++;
clDer.SecondBaseClass::nObj++;
```

При наследовании через промежуточный класс возникает другая проблема — при приведении типа указателя одного класса к другому может возникнуть неоднозначность:

```
DerivedClass clDerived;
DerivedClass *ptrDer = &clDerived;
BaseClass *ptrBase;
ptrBase = (Base *)ptrDer; //Ошибка — неоднозначность приведения типа
```

Проблема решается выполнением явного приведения типа к классу, для которого неоднозначностей не возникает:

```
ptrBase = (Base *)(FirstBase *)ptrDer; // Теперь правильно
```

Размещение объектов производных классов в памяти в случае простого множественного наследования представлено на рис. П1.7.

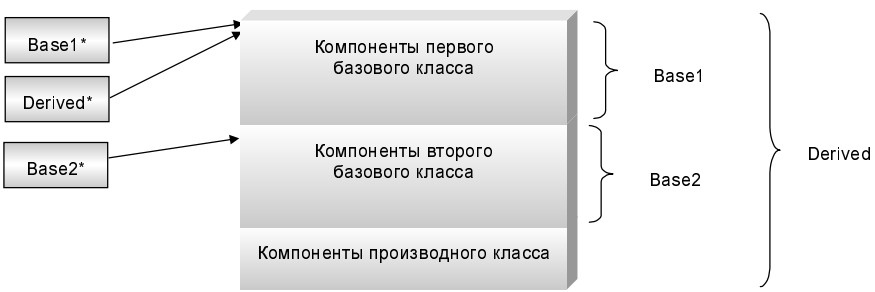



**Рис. П1.7.** Размещение объекта в памяти при простом множественном наследовании

Обратите внимание, что если мы передадим Derived* функции, ожидающей Base1, то все будет в порядке — неоднозначностей не возникает. Однако при вызове функции, ожидающей поступления Base2, возникнет проблема, и необходимо выполнять явное приведение типа.

Давайте теперь посмотрим, что же получается в случае наследования через промежуточный класс. Проблема, которая возникает в этом случае, достаточно наглядно продемонстрирована на рис. П1.8 — это дублирование компонентов "самого" базового класса BaseClass.

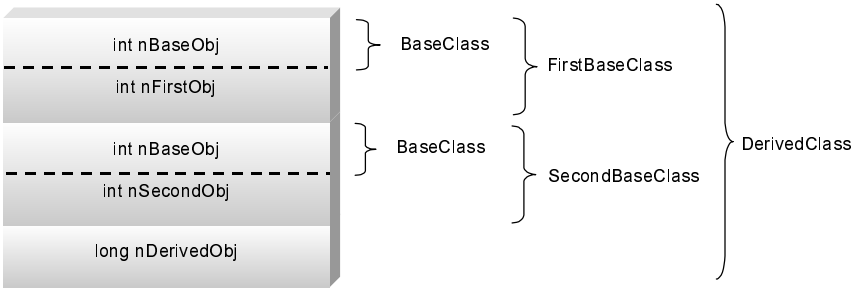



**Рис. П1.8.** Схема размещения объекта при наследовании через промежуточный класс

Впрочем, некоторые программисты могут и не посчитать это проблемой и сознательно использовать обе (или больше) копии. Однако в большинстве случаев такое положение все-таки нежелательно. Чтобы в таких случаях позволить наследование только одной копии BaseClass, в язык C++ включено ключевое слово **virtual**, с помощью которого реализуется так называемое *виртуальное наследование*:

```
class BaseClass
{
  int nBaseObj;
};
class FirstBaseClass : virtual public BaseClass {
 int nFirstObj;
};
class SecondBaseClass : virtual public BaseClass {
 int nSecondObj;
};
class DerivedClass : public FirstBaseClass, public FirstBaseClass
{
  long nDerivedObj;
  // Остальные компоненты производного класса
} clDer;
```

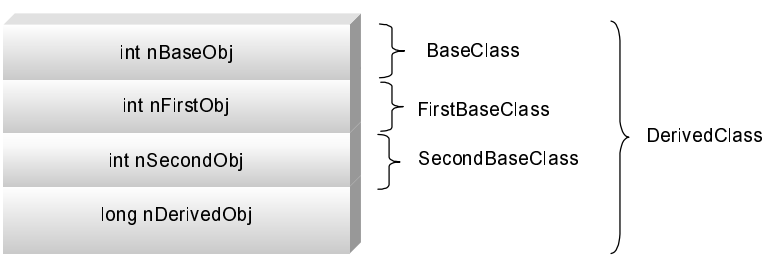



**Рис. П1.9.** Схема размещения объекта при виртуальном множественном наследовании

Такое дополнение (ключевым словом **virtual**) изменяет схему размещения объекта производного класса *DerivedClass*, как это показано на рис. П1.9.

Теперь существует только одна копия BaseClass и нет никакой неоднозначности и избыточности.

# Виртуальные функции — полиморфизм

Прямым результатом наследования явилось включение в язык виртуальных функций. Чтобы разобраться, о чем идет речь, рассмотрим пример:

```
class BaseClass
{
public:
  void Method();
};
class DrivedClass : public BaseClass
{
public:
  void Method();
};
void SimpleFunc(BaseClass *ptrBase)
{
  ptrBase->Method();
}
```

А теперь попробуйте ответить на вопрос: "К какому методу Method относится вызов в функции SimpleFunc?" Правильно, к методу базового класса. И для того, чтобы вызвать метод производного класса, необходимо использование операции разрешения контекста.

Что же делать, чтобы иметь возможность вызывать оба метода? Ответ на этот вопрос дает полиморфизм — использование одного и того же вызова для ссылки на разные методы в зависимости от типа объекта. Для поддержки полиморфизма язык программирования должен поддерживать *механизм позднего связывания*, при котором решение о том, какой именно метод с одним и тем же именем должен быть вызван, принимается в зависимости от типа объекта уже *во время выполнения программы*.

Обычная практика выбора вызываемого метода на этапе компиляции называется *ранним связыванием*.

В C++ имеется ключевое слово **virtual**, использование которого перед именем метода предполагает, что все вызовы этого метода (и всех методов с этим именем в общих производных классах) должны выполняться посредством механизма позднего связывания. Небольшое изменение в предыдущем примере снимает вопрос о неоднозначности вызываемого метода:

```
class BaseClass
{
public:
  virtual void Method(); // Метод объявлен виртуальным
};
class DrivedClass : public BaseClass
{
public:
  void Method();
};
void SimpleFunc(BaseClass *ptrBase)
{
  ptrBase->Method();
}
```

Теперь соответствующий метод будет вызываться в зависимости от того, указатель на какой объект передается в функцию SimpleFunc.

**Примечание**

- Если метод объявлен виртуальным, он не может быть перегружен.
- Виртуальный метод, чаще называемый "виртуальная функция", не может быть статическим.

В каких же случаях следует использовать виртуальные функции, если учитывать, что механизм позднего связывания требует дополнительных расходов времени на вызов метода? Общее правило следующее: если существует вероятность того, что в некотором производном классе будет использован метод, перекрывающий метод базового класса и которому нужен доступ к компонентам этого базового класса, то он должен быть виртуальным.

Может быть объявлен и *виртуальный деструктор*, что позволяет использовать механизм позднего связывания при определении того, какой деструктор следует вызывать для удаления объекта.

С виртуальными функциями тесно связано понятие *абстрактного класса* — такого, который содержит как минимум одну чистую виртуальную функцию. *Чистая виртуальная функция* — это функция, определенная со спецификатором **= 0** и являющаяся такой, для которой нет и не планируется какая-либо реализация. Ее объявление имеет вид:

```
virtual void Method() = 0; // Чистая виртуальная функция
```

Такие функции должны быть определены в производном классе, иначе и он будет абстрактным.

Абстрактные классы чрезвычайно полезны для организации иерархической структуры классов, когда базовый класс определяет некоторое общее пове-

дение, а некоторые частные особенности переносятся в конкретные производные классы.

**Примечание**

Нельзя создавать объекты абстрактного класса — он может быть использован только в качестве базового для построения других классов. Однако указатель на абстрактный класс создавать можно.

# Перегрузка операций

Так же, как и для функций, язык C++ позволяет переопределить действие большинства операций так, чтобы при использовании с объектами конкретного класса они выполняли заданные действия. Такая перегрузка желательна с точки зрения непротиворечивости использования. Классическим примером, приводимым практически во всех учебниках по C++, является выполнение арифметических операций над классами, представляющими комплексные числа. В C++ механизм перегрузки операций реализован путем определения функции с ключевым словом **operator**, стоящим перед перегружаемой операцией.

Переопределить операцию для объектов класса можно используя либо соответствующий метод, либо дружественную функцию класса. В случае переопределения операции с помощью метода класса, последний в качестве неявного параметра принимает ключевое слово **this**, являющееся указателем на данный объект класса. Поэтому если переопределяется бинарная операция, то переопределяющий ее метод класса должен принимать только один параметр, а если унарная, то метод класса вообще не имеет параметров.

Если операция переопределяется при помощи дружественной функции, то она должна принимать соответственно два и один параметр.

При переопределении операций необходимо учитывать некоторые имеющиеся в языке ограничения:

- нельзя изменять количество параметров операции;
- нельзя изменить приоритет операций;
- нельзя определить новые операции;
- нельзя использовать параметры по умолчанию;
- нельзя переопределять операции ., .*, ->*, ::, ?:, а также символы, обрабатываемые препроцессором;
- синтаксис не позволяет различать операции пре- и пост-инкремента и декремента.

Рассмотрим небольшой класс *CPoint* из библиотеки классов MFC:

```
class CPoint : public tagPOINT
{
public:
// Constructors
  CPoint();
  CPoint(int initX, int initY);
  CPoint(POINT initPt);
  CPoint(SIZE initSize);
  CPoint(DWORD dwPoint);
// Operations
...
  BOOL operator==(POINT point) const;
  BOOL operator!=(POINT point) const;
  void operator+=(POINT point);
  void operator-=(POINT point);
// Operators returning CPoint values
  CPoint operator-() const;
  CPoint operator+(POINT point) const;
...
};
BOOL CPoint::operator+=(POINT point) const
{
  return (x += point.x; y += point.y;);
}
```

Использование перегруженных операторов тривиально:

```
CPoint ptObj1(1,2);
CPoint ptObj2(3,4);
CPoint ptObj3 = ptObj1 + ptObj2; // В ptObj3 теперь значение (4, 6)
```

В заключение рассмотрения перегрузки операций хочется сказать, что при неаккуратном использовании этот механизм может привести к получению очень сложного для восприятия кода. В то же время при правильном написании перегруженные операции могут как резко улучшить читабельность программы, так и повысить вероятность повторного применения существующих алгоритмов.

# Шаблоны

Метаморфозы, происходящие в мире программирования, просто удивительны. До сих пор мы много говорили о достоинствах инкапсуляции — сведении в единый объект данных и методов для работы с ними. Теперь речь пойдет о *шаблонах* (*templates*). Понятие шаблона в C++ обеспечивает то, что

в литературе называют *параметризованным типом*, и имеет сходство с соответствующими средствами языков Clu и Ada. Одним из основных достоинств шаблона является то, что они позволяют отделить метод от данных (!). Они настолько значительно изменяют внешнюю сторону программирования, что по праву могут считаться одной из самых важных особенностей C++. Они предназначены для создания функций и классов, отличающихся друг от друга только типом обрабатываемых ими данных. Такие функции и классы часто называют соответственно *параметризованными* или *родовыми функциями* (*generic functions*) и *параметризованными* или *родовыми классами* (*generic classes*). Для их определения в язык C++ включено ключевое слово **template**.

Наше знакомство с шаблонами начнем с параметризованных классов. Общий синтаксис объявления шаблона имеет следующий вид:

```
template<class tType>  // Дальше пойдет шаблон
class className
{
// Компоненты класса
};
```

Здесь префикс **template**<class tType> говорит о том, что описывается параметризованный класс, имеющий параметром тип шаблона tType. При необходимости можно определить несколько параметризованных типов данных, разделяя их запятой. В пределах определения класса имя tType можно использовать в любом месте.

После того как параметризованный класс объявлен, можно создавать его конкретную реализацию:

```
className<dataType> clObj;
```

Тип данных, который заменяет собой переменную tType и с которыми фактически оперирует класс, задается параметром dataType, например

```
className<float *> clObj;
```

В результате такого определения *на этапе компиляции* на основании заданного типа будет автоматически сгенерирован соответствующий объект. Таким образом, использование шаблонов не влечет за собой никаких дополнительных временных издержек по сравнению с явно задаваемыми типами.

Методы параметризованного класса являются параметризованными функциями, и когда они определяются вне базового класса, параметр типа шаблона должен быть указан явно, например:

```
template<class tType> void className<tType>::memberFunc()
{
// Тело функции
}
```

Замечательно здесь то, что за формирование версии каждой параметризованной функции для каждого типа аргумента отвечает компилятор, а не программист.

Можно определить параметризованные функции, не являющиеся методами какого-либо класса. Общий синтаксис такого объявления имеет вид:

```
template<class tType> retType funcName(parametrList);
```

Во всем остальном такие функции ничем не отличаются от обычных.

При рассмотрении шаблонов используются и другие термины, которые можно встретить в литературе по C++. Конкретную версию функции, создаваемую компилятором, называют *порожденной функцией* (*generated function*). Процесс построения такой функции называется *конкретизацией* (*instantiating*).

Ранее мы рассматривали различные структуры данных. При этом каждая из них использовала единственный конкретный тип данных, с которым она могла работать. Теперь, после знакомства с шаблонами C++, легко создать классы контейнеров, которые будут работать с широким кругом типов данных — как встроенных в язык, так и создаваемых программистом. Большую пользу в этом процессе оказывает библиотека MFC, в которую входят специальные классы для хранения групп объектов:

- массивы — класс *CArray* и его варианты;
- списки — класс *CList* и его варианты;
- ассоциативные списки (словари) — класс *CMap* и его варианты.

Их использование исключает необходимость в самостоятельной разработке и длительной отладке, что позволяет уделить больше времени прикладной части приложения.

Для тех, кто заинтересовался шаблонами, могу порекомендовать прекрасную книгу Герберта Шилдта "Теория и практика C++".

# Обработка исключений

*Обработка исключений* обеспечивает передачу управления и информации в произвольную точку вызова, где была затребована обработка исключения данного типа. Могут возникать и обрабатываться исключения любого типа.

Средства обработки исключений нужны, чтобы дать возможность программе, где возникает исключение, которое она не может обработать, передать эту обработку некоторой другой программе, из которой данная программа явно или неявно была вызвана.

Описываемая ниже модель обработки исключений предназначена главным образом для обработки ошибок. Другие варианты ее использования, такие

как завершение цикла или дополнительные пути "нормального" завершения функций, хотя и возможны, однако считаются второстепенными.

Реализации обработки исключений полагаются редкими по сравнению с реализацией функций, а возникновение исключений — возникающими редко по сравнению с вызовами функций. Цель обработки исключений — создание программ вне устойчивых к сбоям подсистем без необходимости обращать внимание на обработку исключений в каждой функции. Другими словами, во многих случаях нет необходимости делать каждую функцию устойчивой к сбоям. Гораздо проще (и лучше), чтобы лишь некоторые функции были ответственны за обработку всех ошибок, возникающих в них и в вызываемых ими функциях.

Язык C++ сам обеспечивает обработку исключений (*exception handling*), причем исключений явных, т. е. таких, возможное возникновение которых должен предусмотреть сам разработчик. При этом проверка необходимости обработки и возбуждение исключения сохраняются, т. е. сохраняется проверка возникновения ошибок, а сама обработка производится в другом месте. Обработка исключений языка C++ является *обработкой с завершением*, т. е. у обработчика нет возможности возобновить исполнение с точки возникновения исключения. При возникновении исключения в обработчик передается произвольное количество информации с контролем типов.

Для лучшего понимания того, как на практике реализуется обработка исключений, приведем следующую синтаксическую сводку:

```
блок-с-контролем:
  try составной_оператор список_обработчиков
список_обработчиков:
  обработчик список_обработчиков_opt
обработчик:
  catch (объявление_исключения) составной_оператор
объявление_исключения:
  список_спецификаторов_типа описатель
  список_спецификаторов_типа абстрактный_описатель
  список_спецификаторов_типа
  ...
выражение_возбуждения_исключения:
  throw выражение_opt
```

Хотя приведенная сводка вполне прозрачна для понимания, дадим краткий комментарий. `<Составной_оператор>` после ключевого слова **try** содержит последовательность операторов, включая вызовы функций, которые проверяются на возникновение исключения. Такой блок еще называют *защищенный блок* (*guarded block*). По сути, именно в рамках этого составного оператора может встретиться `<выражение_возбуждения_исключения>`, где необяза-

тельное <выражение> после ключевого слова **throw** как раз и определяет тип и информацию, описывающую само исключение.

**Примечание**

Возбуждение исключения может происходить и как бы вне составного оператора, следующего после ключевого слова **try**, т. к. отсутствие в функции этого составного оператора еще не значит, что данная функция не выполняется в некотором блоке с контролем, который может быть определен и в библиотеке, что исключает его явное наличие в написанном разработчиком коде программы.

Ключевое слово **throw** без последующего <выражения> может использоваться только в обработчиках исключений для организации обработки текущего исключения в следующем по иерархии вызова блоке с контролем. Заключенное в круглые скобки <Объявление_исключения> после ключевого слова **catch** определяет тип данных, описывающих обрабатываемое исключение. Причем тип данных может быть любым допустимым в языке типом, включая класс. Если в качестве <объявление_исключения> используется многоточие (...), обработчик берет на себя обработку исключений любого типа и, что очень важно, в том числе обработку структурных исключений. Последняя возможность облегчает совместное использование обоих видов обработки исключений. Такой обработчик должен быть последним в списке обработчиков своего блока с контролем.

Обработчики исключения выполняются только при возникновении исключения, заданного типом <объявление_исключения>, и если это исключение не было обработано ранее (об этом ниже). Если же исключения нет, код обработчика не выполняется. Запрещен также переход в обработчик с использованием оператора **goto** или метки **case** оператора **switch**. Однако такой запрет компилятором может не контролироваться, в частности, Microsoft Visual C++ 6.0 этого не делает. Приложение, использующее такой метод выполнения, просто будет аварийно завершено.

Сразу после выполнения оператора **throw** весь код до того места в программе, где будет обрабатываться исключение, игнорируется, т. е. он выполняться не будет. Это один из ключевых принципов, на которых базируется вся обработка исключений, и о нем всегда необходимо помнить при разработке устойчивых к сбоям программ.

Несколько слов следует сказать о том, что же такое блок с контролем. Обычно термин "блок" используется для указания на последовательность операторов, заключенную в фигурные скобки. Напротив, блок с контролем представляет собой составной оператор, следующий за ключевым словом **try**, и список обработчиков. Причем между составным оператором и списком обработчиков не должно быть ни одного другого оператора. Вот простейший пример.

```
int x;
...
```

```
try
{
  if (x < 0)
    throw 0;
  else if (x > MAX_VALUE)
    throw "Out of scope";
}
catch (int)
{
  ...
}
catch (char *error_message)
{
  puts(error_message);
  throw;          // Передаем обработку во внешний блок с контролем,
                  // который в данном примере не представлен
}
catch (...)       // В данном примере этот код
{                 // никогда не будет выполняться, т. к. реализованы
  ...             // все необходимые типизированные обработчики
}
```

В заключение этого раздела осталось пояснить происхождение всех трех ключевых слов, используемых в синтаксисе обработки исключения. Первое слово **try** (проба, испытание) у разработчиков не вызывало сомнений — его значение как нельзя лучше обозначает соответствующие действия. Слово **catch** (перехватывать) вполне подходит, однако мы все время говорим об обработке (handle), нежели о перехвате. Термин handle был признан нецелесообразным из-за его частого использования в программах для IBM-совместимых персональных компьютеров и компьютеров Macintosh. При переходе на C++ не раз приходилось менять имена переменным, таким как new, из-за того, что это слово является ключевым в языке. Термин **throw** (бросать) был использован вместо терминов signal или raise, т. к. стандартная C-библиотека уже имеет функции с этими именами.

## Исключения C++

Основное достоинство обработки исключений заключается не только в том, что она дает возможность обрабатывать самые различные ситуации, которые могут возникать в программе. Сам механизм обработки, основанный на свертывании стека, без которого просто невозможна правильная передача управления, обеспечивает корректное уничтожение переменных и объектов, для которых была зарезервирована память и другие ресурсы. Более того, именно язык программирования C++ в полной мере обеспечивает такое корректное уничтожение. Достигается это путем автоматического вызова

деструкторов в процессе свертывания стека для всех объектов, созданных до возбуждения исключения.

При возникновении исключения управление передается на ближайший обработчик соответствующего типа, причем ближайшим является тот, в чей блок с контролем исполнение попало в последнюю очередь. Если в этом блоке обработчик не найден, проверке подвергается следующий по иерархии вызова блок с контролем. Такой поиск осуществляется до тех пор, пока не будет найден подходящий обработчик.

Естественно, что может возникнуть ситуация, когда соответствующий обработчик так и не будет найден, т. к. он попросту отсутствует. Что же происходит в этом случае, и является ли это ошибкой? Ответ на первый вопрос таков: до того, как обработчик получает управление, на этапе свертки стека возбуждается еще одно исключение и вызывается функция **terminate**. Считать ли такую ситуацию ошибочной, зависит от того, какого рода исключение возникло. Если все равно приложение должно быть завершено, т. к. дальнейшее выполнение невозможно, то именно такое завершение вполне можно считать оправданным. Если же разработчик не позаботился о соответствующей обработке, то естественно это будет ошибкой, причем скорее всего логической.

Как только приложение определилось с реакцией на исключение (обработка или завершение), но непосредственно перед тем, как эта реакция получает управление, происходит свертка стека. Во-первых, для всех объектов, расположенных в нем и имеющих деструкторы, последние вызываются. Если объект был построен частично, то деструкторы вызываются только для его полностью построенных подобъектов. Если исключение возникло в конструкторе на элементе автоматического массива — разрушаться будут лишь уже построенные элементы этого массива. Во-вторых, освобождается и сам стек. Обязательный вызов деструкторов позволяет при возникновении исключений не только корректно уничтожать объекты, но и выполнять другие необходимые действия, например, освобождение ресурсов, закрытие файлов и т. п.

Для лучшего понимания того, как определяется блок с контролем, а также соответствующий обработчик исключения, приведем три возможных варианта.

```
void inner()
{
  throw 0;     // Исключение в блоке_с_контролем,
               // определенным в функции outer
}
void outer()
{
  try
```

```
  {
    inner();
  }
  catch (int) // Обработчик исключения,
              // возбуждаемого в функции inner
  {
  }
}
```

Наличие в функции inner возбуждения исключения может привести к тому, что оно не будет иметь обработчика, т. к. при использовании этой функции в разных контекстах может случиться так, что программист просто забудет предоставить в каждом случае соответствующий обработчик. Для того чтобы как-то проконтролировать возникновение таких ситуаций, можно либо переопределить функцию завершения (**terminate**), либо в функции, находящейся на самом верху иерархии вызовов, например, **main**, определить блок с контролем, содержащим обработчик, способный обрабатывать исключения любого типа (**catch** (...)).

Следующий пример иллюстрирует вариант вложенных блоков с контролем. Наличие блока с контролем в функции inner не очевидно для функции outer.

```
void inner(int mode)
{
// Оба исключения возбуждаются в текущем блоке с контролем,
// однако их обработчики определены в различных блоках (и функциях)
  try
  {
    if (mode != 0)
      throw "Exception!";
    else
      throw 0;
  }
  catch (char *)  // Обработчик исключения,
                  // возбуждаемого в этой же функции
  {
  }
}
void outer()
{
  try
  {
    inner(0);
  }
```

```
  catch (int)      // Обработчик исключения,
                   // возбуждаемого в функции inner
  {
  }
}
```

Последний пример иллюстрирует вложенные блоки с контролем, реализованные в одной функции.

```
void outer(int mode)
{
  try
  {
    try
    {
      if (mode != 0)
        throw "Exception!";
      else
        throw 0;
    }
    catch (char *)
    {
    }
  }
  catch (int)
  {
  }
}
```

Свертка стека никак не сказывается на объекте, определяющем тип исключения. Достигается это тем, что обработчик получает не сам объект, а его копию, которая реализуется, возможно, с помощью конструктора копирования, при этом осуществляется копирование объекта из одного места стека в другое.

```
try
{
  CExClass e_try("Exception", 15);
  throw e;
}
catch (CExClass& e_exept) // Ошибки не будет — ссылка e_exept
{                         // ссылается не на объект e_try
}
```

Так как операнд **throw** трактуется в точности так же, как параметр в вызове функции или операнд в операторе **return**, а уничтожение самого операнда

происходит после соответствующего копирования, обработчик исключения получает объект (копию), который располагается в текущем для обработчика месте стека. Интересно, что объекты, описывающие исключения, реализованные в библиотеке MFC, создаются не в стеке, как в предыдущем примере, а в свободной памяти. Естественно, что в этом случае требуется явное удаление таких объектов.

```
try
{
  CException *e = new CException;
  throw e;
}
catch (CException *e)
{
  delete e;
}
```

Такой подход не годится для обработки исключений, возникающих в процессе управления свободной памятью, т. к. в этих ситуациях некорректно использовать для размещения временного объекта обычные средства размещения в свободной памяти — ее и так уже возможно не хватает. Кроме того, подключение общего механизма работы со свободной памятью было бы неэффективным и сделало бы обработку исключений чувствительной к ошибкам работы со свободной памятью. Таким образом, реализация может использовать заранее выделенный пул памяти исключительно для механизма обработки исключений. Что касается последнего момента, то для исключений, связанных с отсутствием свободной памяти, библиотека MFC обращается к заранее созданным объектам, описывающим такого рода исключения.

Следующий пример демонстрирует порядок вызовов деструкторов для различных объектов. Для демонстрации порядка вызова деструктора во время сворачивания стека используются два класса: *CExcept*, который является классом описания исключения, и *CTheObject*.

```
#include "afxwin.h"
class CTheObject
{
public:
  CTheObject();
  ~CTheObject();
};
class CExcept
{
public:
  CExcept();
```

```
  CExcept(CExcept& e);
  ~CExcept();
};
CTheObject::CTheObject()
{
  TRACE("\tConstruct: CTheObject\n");
}
CTheObject::~CTheObject()
{
  TRACE("\tDestruct: CTheObject\n");
}
CExcept::CExcept()
{
  TRACE("\tConstruct: CExcept\n");
}
CExcept::CExcept(CExcept&)
{
  TRACE("\tCopy Construct: CExcept\n");
}
CExcept::~CExcept()
{
  TRACE("\tDestruct: CExcept\n");
}
void ThrowFunction()
{
  TRACE("Start >>>> ThrowFunction\n");
  CTheObject o;
  CExcept e;
  throw e;
  TRACE("End >>>> ThrowFunction\n");
}
int WINAPI WinMain(
    HINSTANCE hInstance,
    HINSTANCE hPrevInstance,
    LPSTR lpCmdLine,
    int nCmdShow)
{
  TRACE("Start >>>> Main\n");
  try
  {
    TRACE("Begin -> try\n");
    ThrowFunction();
    TRACE("End -> try\n");
  }
```

```
  catch (CExcept& e)
  {
    TRACE("Begin -> catch\n");
    TRACE("End -> catch\n");
  }
  TRACE("End >>>> Main\n");
  return 0;
}
```

Ниже приведено содержимое окна отладчика (номера строк и отступы добавлены для облегчения понимания):

```
 1. Start >>>> Main
 2. Begin -> try
 3. Start >>>> ThrowFunction
 4. Construct: CTheObject
 5. Construct: CExcept
 6. Copy Construct: CExcept
 7. Destruct: CExcept
 8. Destruct: CTheObject
 9. Begin -> catch
10. End -> catch
11. Destruct: CExcept
12. End >>>> Main
```

Как видите, результат согласуется с изложенным ранее материалом. Обратите внимание на то, что сворачивание стека начинается только после того, как завершается создание копии объекта описания исключения. Пожалуй, стоит также обратить внимание на то, что копия объекта класса *CExcept* уничтожается сразу по выходу из обработчика исключения. Кстати, при замене типа обработчика со ссылки на копию объекта (вместо ссылки — **catch** (CExcept e)) строки 6 и 11 будут продублированы, т. к. в этом случае объект описания должен создаваться (и уничтожаться) дважды: при создании копии и при создании фактического параметра.

## Типы исключений

В связи с тем, что язык C++ позволяет обрабатывать исключения любого типа, следует определить обработчики для каждого типа исключения. Либо обработчик может обрабатывать исключения того же типа, что объект описания, использованный при возбуждении ситуации, либо исключение может быть обработано обработчиком любых исключений. Кроме того, возможно использование ссылок в качестве указания типа обрабатываемого исключения. Учтите, что использование ссылок сокращает количество создаваемых копий при передаче объекта описания исключения в обработчик.

Для исключений, описанных при помощи классов, так же может быть использован обработчик базового класса объекта описания исключения. Наконец, вполне возможно использование указателей на объекты описания исключений. Однако в этом случае указание на объект, находящийся в стеке, приведет к тому, что обработчик получит указатель на уже уничтоженный объект, т. к. к моменту передачи управления обработчику сворачивание стека уже завершено.

Приведу сводку возможных обработчиков для всех указанных выше типов исключений. После возбуждения исключения оно может быть обработано следующими обработчиками:

- обработчик исключений любого типа (**catch** (...));
- обработчик исключений того же типа, что и сам объект описания исключения (т. к. обработчик получает копию объекта, модификаторы **const** и **volatile** игнорируются);
- обработчик исключений ссылки на тот же тип, что и сам объект описания исключения;
- обработчик исключений ссылки на тот же тип, что и сам объект описания исключения, с модификаторами **const** и **volatile**;
- обработчик исключений базового класса объекта описания исключения (т. к. обработчик получает копию объекта, модификаторы **const** и **volatile** игнорируются); обработчик не должен располагаться в списке обработчиков ранее того, который обрабатывает объекты описания исключения, порождаемые от базового класса;
- обработчик исключений ссылки на базовой класс объекта описания исключения;
- обработчик исключений ссылки на базовой класс объекта описания исключения с модификаторами **const** и **volatile**;
- обработчик исключений указателя на объект, который может быть получен при помощи стандартных преобразований указателей.

## Специальные функции

Комитет по разработке стандарта языка программирования C++ предполагает обработку двух групп ситуаций, которые могут возникнуть при возбуждении и обработке исключений.

Первая включает ситуацию, когда в функции со спецификацией исключений возбуждается исключение, не указанное в спецификации.

Вторая группа включает целый набор ситуаций, когда:

- механизм обработки исключений не находит реакции на возникшее исключение;

- механизм обработки исключений обнаружил, что разрушен стек;
- деструктор, вызванный в процессе свертки стека при возникновении исключения, сам пытается завершиться, возбуждая при этом исключение.

Рассмотрим эти группы по порядку.

При возникновении исключения, не указанного в спецификации исключений функции, механизм обработки исключений вызывает системную функцию **unexpected**, которая в соответствии с рекомендациями ANSI может быть заменена на пользовательскую функцию при помощи функции **set_unexpected**. Однако Microsoft Visual C++ 6.0 не поддерживает спецификацию исключений при описании функций (подробнее см. документацию по C++), и, соответственно, пользовательская функция вызываться никогда не будет.

**Примечание**

В ряде случаев стандарт языка программирования накладывает определенные требования на реализацию тех или иных положений. Однако часто некоторые моменты зависят именно от реализации. Например, размещение (выравнивание) полей структур и в C, и в C++ зависит от реализации. С другой стороны, конкретный компилятор может отходить от стандартов как в сторону расширения, так и в сторону усечения тех или иных возможностей. Именно поэтому принято говорить не просто о компиляторе языка C++, а, например, о Microsoft Visual C++, Borland C++, Symantec C++ и т. п. Более того, хорошо еще добавлять номер версии.

Это вполне оправдано, т. к. даже разработчики стандарта указывают, что:

"В идеале могут возникнуть только исключения, упомянутые в спецификации исключений... Почему бы не достичь идеала, заставив каждую функцию иметь спецификацию исключений и отвергать любую программу, где возникнет неожиданное исключение? Предотвращение подобных ошибок во время компиляции легло бы тяжким бременем на программиста, а потому он стал бы обходить и обманывать контроль... Кроме того, если функция изменилась и возбуждает новое для нее исключение, все функции, прямо или косвенно вызывающие ее, должны быть изменены и перекомпилированы".

Возникновение одной из ситуаций второй группы приводит к вызову функции **terminate**, которая может быть заменена на пользовательскую при помощи функции **set_terminate**. Функция **terminate** просто вызывает функцию **abort** системной библиотеки. Пользовательская же функция может выполнить какие-либо необходимые действия. Вообще по завершению выполнения таких действий рекомендуется вызвать функцию **exit** для завершения текущего потока, т. к. в отличие от функции **abort** функция **exit** все-таки производит некоторые полезные действия, например, закрывает открытые файлы. Однако, если этого не сделать, по завершению пользовательской функции будет в обязательном порядке вызвана функция **abort**, т. к. никаких попыток возобновить или продолжить работу не делается.

Функция **set_terminate** возвращает указатель на последнюю установленную функцию завершения, причем для каждого потока определяется своя функция завершения. Это может быть полезно, но следует помнить, что, заменив функцию завершения в одном потоке, не следует ожидать той же реакции и для другого потока.

В общем случае для того, чтобы избежать возникновения ситуации отсутствия обработчика исключения, следует просто добавить обработчик всех исключений в конец списка обработчиков.

## Порядок обработки исключений

Обработчики исключений проверяются на применимость в порядке появления в списке обработчиков. Ошибкой является расположение обработчика исключения базового класса перед обработчиком производного от него класса, поскольку при таком порядке обработчик на производный класс никогда не получит управления. То же самое касается и обработчика всех исключений (**catch** (...)). Он должен быть последним в списке, иначе ни один обработчик, указанный после него, также не получит управления. Интересно, что преобразование типов, например, от **int** к **float**, не производится, так что следующий код приведет к возникновению ситуации по отсутствию обработчика:

```
try
{
  throw 25;
}
catch (float)
{
}
```

Альтернативным подходом к обработке исключений, описываемых при помощи объектов классов, может быть использование механизма виртуальных функций:

```
class CError
{
  ...
  virtual Evaluate();
};
class CMathError : public CError
{
  ...
  virtual Evaluate();
};
```

```
class CDataError : public CError
{
  ...
  virtual Evaluate();
};
try
{
  ...
}
catch (CError& e)
{
  e.Evaluate();
}
```

Тип обработчика должен быть ссылкой, иначе в обработчик будет передана только часть объекта (в данном примере относящаяся к классу CError). Если же тип обработчика является указателем, то объект должен быть статическим или располагаться в свободной памяти, и его необходимо будет после использования удалить.

## Рекомендации по использованию

Способ обработки исключений можно разделить на две группы. К первой относятся случаи, когда код возбуждения исключения находится в той же функции, что и обработчик. Ко второй — когда в **try**-блоке происходит вызов некоторой функции, в которой и происходит возбуждение исключения. Два этих случая требуют различного подхода к управлению объектами, создаваемыми с момента входа в защищенный блок и до возбуждения исключения. Конечно, речь идет прежде всего об объектах, динамически создаваемых в свободной памяти, а также о действиях над ресурсами, которые следует выполнить по завершению работы, например, закрыть открытые файлы.

В первом случае рекомендуется инициализировать переменные значениями, характеризующими начальное состояние. Лучше всего на эту роль подходит значение NULL. По завершению обработки исключения все объекты, указатели на которые не равны NULL, должны быть освобождены. Такой подход существенно упрощает программирование, избавляя код от громоздких конструкций:

```
void Function()
{
  LPVOID lpBuffer = NULL;
  FILE *fp_1 = NULL, *fp_2 = NULL;
  try
```

```
  {
    lpBuffer = malloc(1000);
    if (lpBuffer)
      throw "Нет памяти\n";
    if ((fp_1 = fopen("FILE_1.DAT", "r")) == NULL)
      throw "Невозможно открыть файл FILE_1.DAT\n";
    if ((fp_2 = fopen("FILE_2.DAT", "w")) == NULL)
      throw "Невозможно открыть файл FILE_2.DAT\n";
    if (!fread(lpBuffer, 1, 1000, fp_1))
      throw "Ошибка чтения из файла FILE_1.DAT\n";
    if (!fwrite(lpBuffer, 1, 1000, fp_2))
      throw "Ошибка записи в файл FILE_2.DAT\n";
  }
  catch (char *error)
  {
    puts(error);
  }
  if (lpBuffer != NULL)
    free(lpBuffer);
  if (fp_1 != NULL)
    fclose(fp_1);
  if (fp_2 != NULL)
    fclose(fp_2);
}
```

Все деинициализирующие действия выполняются в одном месте. В противном случае, начиная с ошибки открытия первого же файла, потребовалось бы перед каждым возбуждением исключения производить все больше и больше такого рода действий — сначала освобождение памяти, далее освобождение памяти и закрытие первого файла, освобождение памяти, закрытие первого файла и закрытие второго файла и т. д. Кроме того, после выполнения обработчика все равно бы потребовалось определить, какие из деинициализирующих действий выполнены, а какие нет. (Альтернативой последнему может служить завершение функции в конце обработчика.)

К сожалению, в случае, когда возбуждение исключения и обработчик находятся в различных функциях, перед собственно возбуждением исключения потребуется освобождение локальных ресурсов этой функции. Тут уже ничего не поделаешь. Кстати, частным случаем является возбуждение исключения в конструкторе класса, что часто применяется в классах библиотеки MFC, например, классах графических ресурсов.

Вообще при невозможности правильного создания объекта класса, когда память под него уже выделена, но дополнительная инициализация завершается неудачно, не существует простого способа удаления самого объекта во время выполнения конструктора. Чтобы как-то сигнализировать о неудач-

ном выполнении конструктора, можно предусмотреть специальный член класса, либо воспользоваться механизмом исключений. Однако наиболее приемлемый способ — создать "пустой" конструктор и инициализирующую функцию (например, Create), неудача выполнения которой потребует простого вызова деструктора напрямую, вне каких-либо методов, включая конструктор.

**Примечание**

Тому, кто хочет более подробно познакомиться с обработкой исключений, могу порекомендовать книгу А. Мешкова и Ю. Тихомирова "Visual C++ и MFC. Том II".

# ПРИЛОЖЕНИЕ 2

# Основные типы 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 количество сообщений характерно для одного типа окна. Другие сообщения играют особую роль в совместном использовании данных или 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 файле <winuser.h> следующим образом:

```
#define WM_COMMAND 0x0111
```

или

```
#define WM_PAINT   0x000F
```

Префикс WM_ говорит о том, что речь идет об "оконном сообщении" (от Window Message). Эти символические имена обозначают числовые значения. 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 Listbox и т. д. Помимо этого, вы можете создавать и собственные сообщения.

Чтобы как-то систематизировать такое огромное количество сообщений, я разделил их на следующие категории:

| | |
|---|---|
| Аппаратные | Входные данные от мыши и клавиатуры |
| Организация окна | Уведомление, требование действия, запрос |
| Организация интерфейса пользователя | Меню, указатель мыши, линейка прокрутки, блоки диалога, MDI |
| Завершение работы | Закрытие прикладной программы или системы |
| Частные | Элементы управления: кнопка, редактор и т. д. |

| | |
|---|---|
| Уведомление о системном ресурсе | Изменение цвета, шрифтов, буферизация печати, режимы работы устройства |
| Обмен данными | Буфер обмена и динамический обмен данными |
| Внутрисистемные | Недокументированные сообщения |

# Аппаратные сообщения

Оконная процедура получает сообщения, инициируемые тремя частями аппаратного обеспечения: клавиатурой, мышью и системным таймером. Все они генерируют аппаратные прерывания. Поскольку диспетчеризация Windows не действует в режиме прерываний, аппаратные события должны быть буферизованы. Это обеспечивает обработку событий в порядке их поступления. Например, когда вы нажимаете клавишу <K>, прерывание уведомляет систему о том, что приготовлены данные от клавиатуры. Драйвер клавиатуры отыскивает эти данные и создает элемент в первичной очереди, откуда система их передает в очередь приложения, а точнее, конкретного потока. Когда наступает его очередь, сообщения, содержащие информацию о клавише <K>, направляются в соответствующую оконную процедуру. Сообщения от мыши и системного таймера обрабатываются аналогично. В табл. П2.1—П2.4 представлены основные сообщения, которые формируются в ответ на работу аппаратного обеспечения.

***Таблица П2.1.*** *Сообщения от мыши в клиентской области окна*

| | |
|---|---|
| WM_LBUTTONDBLCLK | Двойное нажатие левой кнопки |
| WM_LBUTTONDOWN | Нажатие левой кнопки |
| WM_LBUTTONUP | Отпускание левой кнопки |
| WM_MBUTTONDBLCLK | Двойное нажатие средней кнопки |
| WM_MBUTTONDOWN | Нажатие средней кнопки |
| WM_MBUTTONUP | Отпускание средней кнопки |
| WM_MOUSEMOVE | Перемещение мыши |
| WM_RBUTTONDBLCLK | Двойное нажатие правой кнопки |
| WM_RBUTTONDOWN | Нажатие правой кнопки |
| WM_RBUTTONUP | Отпускание правой кнопки |

***Таблица П2.2.*** *Сообщения от мыши в неклиентской области окна*

| | |
|---|---|
| WM_NCLBUTTONDBLCLK | Двойное нажатие левой кнопки |
| WM_NCLBUTTONDOWN | Нажатие левой кнопки |

***Таблица П2.2*** *(окончание)*

| | |
|---|---|
| WM_NCLBUTTONUP | Отпускание левой кнопки |
| WM_NCMBUTTONDBLCLK | Двойное нажатие средней кнопки |
| WM_NCMBUTTONDOWN | Нажатие средней кнопки |
| WM_NCMBUTTONUP | Отпускание средней кнопки |
| WM_NCMOUSEMOVE | Перемещение мыши |
| WM_NCRBUTTONDBLCLK | Двойное нажатие правой кнопки |
| WM_NCRBUTTONDOWN | Нажатие правой кнопки |
| WM_NCRBUTTONUP | Отпускание правой кнопки |

***Таблица П2.3.*** *Сообщения от клавиатуры*

| | |
|---|---|
| WM_CHAR | Ввод символа |
| WM_DEADCHAR | Диакритический знак (умляут, ударение и т. д.) |
| WM_KEYDOWN | Была нажата клавиша |
| WM_KEYUP | Была отпущена клавиша |
| WM_SYSCHAR | Ввод специального символа |
| WM_SYSDEADCHAR | Специальный диакритический знак |
| WM_SYSKEYDOWN | Была нажата специальная клавиша |
| WM_SYSKEYUP | Была отпущена специальная клавиша |

***Таблица П2.4.*** *Сообщения от таймера*

| | |
|---|---|
| WM_TIMER | Истекло заданное время |

Сообщения от таймера помещаются в очередь так же, как и сообщения от мыши и клавиатуры (т. е. таймеры в системе Windows не являются точными).

Сообщение WM_MOUSEMOVE заслуживает особого упоминания, т. к. обрабатывается особым образом, чтобы избежать переполнения очереди аппаратных событий. Дело в том, что при быстром перемещении мыши могут произойти сотни прерываний. Во избежание переполнения очереди система Windows хранит только одно сообщение о перемещении мыши в каждый момент времени. Когда приходит новое сообщение, Windows проверяет, есть ли уже подобное. Если оно есть, то в нем обновляется информация о положении. Новое сообщение помещается в очередь сообщений только в том случае, если сообщения о перемещении мыши еще не было.

Сообщения от клавиатуры делятся на два основных типа: обычные и системные. Обычно системные сообщения от клавиатуры (начинающиеся на WM_SYS...) игнорируются, т. к. оконная процедура, устанавливаемая по умолчанию, превращает их в соответствующую системную команду.

# Сообщения обслуживания окна

Здесь собраны наиболее часто используемые сообщения, и вам следует хорошенько разобраться с каждым из них. К сожалению, имена не окажут вам большой помощи. Начнем с того, что сообщения этой группы можно разбить на три части: уведомление, требование действия и запрос (табл. П2.5—П2.7).

***Таблица П2.5.*** *Оконные сообщения: уведомление*

| | |
|---|---|
| WM_ACTIVATE | Окно активно |
| WM_ACTIVATEAPP | Приложение активно |
| WM_CREATE | Окно было создано |
| WM_DESTROY | Окно было уничтожено |
| WM_ENABLE | Был разрешен ввод в окно |
| WM_KILLFOCUS | Окно потеряло фокус ввода от клавиатуры |
| WM_MOUSEACTIVATE | Окно станет активным после нажатия кнопки мыши |
| WM_MOVE | Окно было перемещено |
| WM_SETFOCUS | Окно получило фокус ввода от клавиатуры |
| WM_SIZE | Окно изменило размер |

***Таблица П2.6.*** *Оконные сообщения: требование действия*

| | |
|---|---|
| WM_CLOSE | Закрыть (уничтожить) окно |
| WM_ERASEBKGND | Удалить фон |
| WM_ICONERASEBKGND | Удалить фон окна, показываемого в виде пиктограммы |
| WM_NCACTIVATE | Изменить заголовок окна для показа активного состояния |
| WM_NCCREATE | Создать неклиентскую область окна |
| WM_NCDESTROY | Уничтожить неклиентскую область окна |
| WM_NCPAINT | Перерисовать неклиентскую область |

***Таблица П2.6*** *(окончание)*

| | |
|---|---|
| WM_PAINT | Перерисовать клиентскую область |
| WM_PAINTICON | Перерисовать клиентскую область окна, показываемого в виде пиктограммы |
| WM_SETREDRAW | Запретить перерисовку окна |
| WM_SETTEXT | Изменить текст окна |
| WM_SHOWWINDOW | Изменить режим отображения окна |

***Таблица П2.7.*** *Оконные сообщения: запросы*

| | |
|---|---|
| WM_GETMINMAXINFO | Запрос информации о минимальном/максимальном размере окна |
| WM_GETTEXT | Запрос текста заголовка окна, а для окон, не имеющих заголовка (например, для кнопок) — текста, размещенного в окне |
| WM_GETTEXTLINGTH | Запрос длины текста окна |
| WM_NCCALCSIZE | Запрос размеров клиентской области |
| WM_QUERYNEWPALETTE | Запрос существования новой палитры |
| WM_QUERYOPEN | Запрос возможности открытия окна, отображаемого в виде пиктограммы |

Сообщение-уведомление говорит оконной процедуре о том, что состояние окна было изменено. При этом оконная процедура вовсе не должна выполнять действие, обозначенное именем сообщения. Например, сообщение WM_MOVE не является запросом на некоторое перемещение. Наоборот, оно является "сообщением после действия" и напоминает, что окно уже было перемещено. Обратите внимание, что из сообщения не ясно, каким образом произошло перемещение. Это могло быть результатом действий пользователя, равно как и другой программы. В любом случае сообщения-уведомления представляют собой одностороннюю связь с оконной процедурой.

Сообщение-требование действия говорит само за себя, т. к. отсутствие необходимого действия вызывает ошибку в интерфейсе пользователя. Например, сообщение WM_PAINT посылается оконной процедуре, если окно необходимо перерисовать (обновить) из-за изменения данных, связанных с окном, или в результате активизации и перевода окна на передний план из-за другого окна. Если оконная процедура не перерисует окно, то его "видимое" содержимое не будет актуальным. Обычно устанавливаемая по умолчанию оконная процедура обеспечивает некоторые необходимые минимальные действия, однако зачастую следует перехватывать и обрабатывать такого

рода сообщения самому, как в случае с сообщением WM_PAINT. При этом нередко нужно воспроизвести все действия обработчика, устанавливаемого по умолчанию.

Сообщения-запросы требуют ответа. Они используются для двусторонней связи между Windows и вашей программой. Подобно сообщениям-требованиям действия, оконная процедура, устанавливаемая по умолчанию, даст приемлемый ответ в большинстве случаев. Сообщения-запросы следует перехватывать, если вы хотите изменить ответ по умолчанию. Например, сообщение WM_QUERYOPEN вы получите, когда пользователь попытается открыть минимизированное окно. Если вы хотите, чтобы программа выполнялась только тогда, когда окно отображается в виде пиктограммы, то нужно просто ответить FALSE вместо TRUE, установленного по умолчанию.

# Сообщения об организации интерфейса пользователя

Эта группа содержит сообщения для других объектов пользовательского интерфейса, включающих меню прикладной программы, указатель мыши, линейки прокрутки, блоки диалога и их элементы управления, а также сообщения, используемые для поддержки многодокументного интерфейса (MDI). Основные сообщения приводятся в табл. П2.8—П2.13.

***Таблица П2.8.*** *Сообщения от меню*

| | |
|---|---|
| WM_COMMAND | Был выбран пункт меню |
| WM_INITMENU | Уведомление о возможности изменить меню перед его активизацией |
| WM_INITMENUPOPUP | Уведомление о возможности изменить пункты всплывающего меню перед его отображением на экране |
| WM_MENUCHAR | Для выбора меню использована мнемоническая клавиша |
| WM_MENUSELECT | Пользователь просматривает меню |

***Таблица П2.9.*** *Системные команды*

| | |
|---|---|
| WM_SYSCOMMAND | Была выбрана системная команда |

***Таблица П2.10.*** *Сообщения от указателя мыши*

| | |
|---|---|
| WM_NCHITTEST | Передача в окно текущего положения указателя мыши |
| WM_SETCURSOR | Уведомление окну при перемещении мыши с возможностью изменить курсор в зависимости от положения указателя |

***Таблица П2.11.*** *Сообщения линейки прокрутки*

| | |
|---|---|
| WM_HSCROLL | Была нажата горизонтальная линейка прокрутки |
| WM_VSCROLL | Была нажата вертикальная линейка прокрутки |

***Таблица П2.12.*** *Сообщения блока диалога и его элементов управления*

| | |
|---|---|
| WM_COMMAND | Управляющая связь с блоком диалога |
| WM_COMPAREITEM | Посылается родителю с запросом на сравнение двух элементов для сортировки |
| WM_CTLCOLOR | Управляющий запрос на установку цветов |
| WM_DELETEITEM | Уведомление, направляемое элементам управления Listbox и Combobox,о том, что удален элемент |
| WM_DRAWITEM | Запрос родителю пользовательских элемента управления и меню на рисование |
| WM_GETDLGCODE | Запрос оконной процедуре, какие типы сообщений будут ею впоследствии обрабатываться |
| WM_GETFONT | Запрос дескриптора основного шрифта отображения |
| WM_INITDIALOG | Запрос на инициализацию диалогового окна |
| WM_MEASUREITEM | Запрос родителю элемента управления на обеспечение его размеров при прорисовке |
| WM_SETFONT | Требование к элементу управления: использовать данный шрифт |

***Таблица П2.13.*** *Сообщения многодокументного интерфейса (MDI)*

| | |
|---|---|
| WM_CHILDACTIVATE | Объявляет порождающему окну об активизации дочернего окна |
| WM_MDIACTIVATE | Объявляет дочернему окну MDI, что оно получает или теряет активизацию |
| WM_MDICASCADE | Требование оформить дочерние окна MDI в каскадной (ступенчатой) форме |
| WM_MDICREATE | Требование окну клиента MDI создать дочернее окно |
| WM_MDIGETACTIVE | Запрос окна клиента MDI о текущем активном дочернем окне |
| WM_MDIICONARRANGE | Требование упорядочить расположение дочерних окон MDI, отображаемых в виде пиктограмм |

***Таблица П2.13*** *(окончание)*

| | |
|---|---|
| WM_MDIMAXIMIZE | Требование максимизировать дочерние окна MDI |
| WM_MDINEXT | Требование активизировать следующее дочернее окно MDI |
| WM_MDIRESTORE | Требование восстановить дочернее окно MDI в его предыдущем состоянии |
| WM_MDISETMENU | Настраивает меню для текущего дочернего окна MDI |
| WM_MDITITLE | Требование оформить дочерние окна MDI в форме черепичного (без перекрытий) расположения |

# Сообщения о завершении

Эта группа сообщений самая малочисленная, но все они очень важны, т. к. используются для управления завершением приложений (табл. П2.14)

***Таблица П2.14.*** *Завершение работы приложения и системы*

| | |
|---|---|
| WM_QUIT | Требование на завершение приложения |
| WM_QUERYENDSESSION | Запрос готовности приложения к завершению работы всей системы |
| WM_ENDSESSION | Уведомление о результатах запроса на завершение работы системы |

# Частные сообщения

Очень просто определить собственные сообщения и использовать их для связи между различными окнами, которые вы создаете. При определении частного сообщения необходимо использовать область значений, начинающуюся с WM_USER, которая определена в <winuser.h> следующим образом:

```
#define WM_USER 0x0400
```

Например, если мы хотим работать с окнами, которые показывают числа, то могли бы контролировать появление чисел различных систем счисления с помощью следующих частных сообщений (PM_ — префикс "частного сообщения" от Private Message):

```
#define PM_DECIMAL WM_USER + 0
#define PM_BINARY  WM_USER + 1
#define PM_HEX     WM_USER + 2
#define PM_OCTAL   WM_USER + 3
```

Кроме того, есть несколько частных системных сообщений (табл. П2.15).

***Таблица П2.15.*** *Частные системные сообщения*

| | |
|---|---|
| WM_CANCELMODE | Требование системы отменить режим, например, захват мыши |
| WM_ENTERIDLE | Сообщает, что система находится в режиме ожидания |

# Информационные сообщения системных ресурсов

Имеется всего восемь сообщений с уведомлением об изменении некоторых системных ресурсов (табл. П2.16). Они посылаются окну верхнего уровня каждой программы при изменении системного ресурса. Например, когда добавляются или удаляются шрифты в системе, распространяется сообщение WM_FONTCHANGE. А когда пользователь изменяет системные цвета или системное время, отправляются сообщения WM_SYSCOLORCHANGE или WM_TIMECHANGE, соответственно. Типичной реакцией на часть такого рода сообщений является немедленная запись произведенных изменений, т. е. сохранение выполненной работы. Судите сами.

***Таблица П2.16.*** *Сообщения с уведомлением об изменениях системных ресурсов*

| | |
|---|---|
| WM_COMPACTING | Недостаточно системной памяти |
| WM_DEVMODECHANGE | Изменилась настройка принтера |
| WM_FONTCHANGE | Изменились установленные в системе шрифты |
| WM_PALETTECHANGED | Палитра аппаратных цветов изменена |
| WM_SPOOLERSTATUS | Задание было удалено из очереди буфера печати |
| WM_SYSCOLORCHANGE | Один или более системных цветов изменены |
| WM_TIMECHANGE | Изменилось системное время |
| WM_WININICHANGE | Изменился файл инициализации <win.ini> |

Не всякое изменение представляет интерес для каждой программы. Например, программа, использующая таймер, вероятно, будет проверять новое значение времени при получении сообщения WM_TIMECHANGE. Но если она не работает со шрифтами, то, скорей всего, сообщение WM_FONTCHANGE будет проигнорировано. Поскольку большинство программ

использует системные цвета, то они будут реагировать на сообщение WM_SYSCOLORCHANGE, организуя перерисовывание новыми цветами.

# Сообщения о совместном использовании данных

В Windows есть сообщения, которые посылаются при совместном использовании данных для поддержки механизмов работы с буфером обмена (clipboard) и динамического обмена данными (DDE), которые определены в файлах <winuser.h> и <dde.h>, соответственно. И хотя полное обсуждение этого вопроса выходит за рамки книги, мы приведем список сообщений этой группы (табл. П2.17—П2.18).

***Таблица П2.17.*** *Сообщения от буфера обмена*

| | |
|---|---|
| WM_ASKCBFORMATNAME | Запрашивает имя формата буфера обмена |
| WM_CNANGECBCHAIN | Уведомление об изменении цепочки просмотра |
| WM_DESTRIYCLIPBOARD | Содержимое буфера обмена уничтожено |
| WM_DRAWCLIPBOARD | Содержимое буфера обмена изменено |
| WM_HSCROLLCLIPBOARD | Горизонтальная прокрутка буфера обмена |
| WM_PAINTCLIPBOARD | Запрашивает рисование буфера обмена |
| WM_RENDERALLFORMATS | Запрашивает данные для всех ранее объявленных форматов буфера обмена |
| WM_RENDERFORMAT | Запрашивает данные для ранее объявленного формата буфера обмена |
| WM_SIZECLIPBOARD | Размер буфера обмена изменен |
| WM_VSCROLLCLIPBOARD | Вертикальная прокрутка буфера обмена |

***Таблица П2.18.*** *Сообщения DDE*

| | |
|---|---|
| WM_DDE_ACK | Подтверждение приема |
| WM_DDE_ADVISE | Запрос от пользователя DDE об установлении постоянного канала передачи данных |
| WM_DDE_DATA | Направляет элемент данных от DDE-сервера клиенту DDE |
| WM_DDE_EXECUTE | Дает указание DDE-серверу выполнить серию команд |
| WM_DDE_INITIATE | Вход в DDE-сервер |

***Таблица П2.18*** *(окончание)*

| | |
|---|---|
| WM_DDE_POKE | Запрос пользователя серверу об обновлении определенного элемента данных |
| WM_DDE_REQUEST | Однократный запрос пользователя на информацию |
| WM_DDE_TERMINATE | Выход из DDE-сервера |
| WM_DDE_UNADVISE | Завершить постоянный канал передачи данных, инициализированный сообщением WM_DDE_ADVISE |

# Внутрисистемные сообщения

Помимо перечисленных, имеется большая группа сообщений, не описанных ни в какой документации. Это внутренние системные сообщения, которые Windows использует в своих собственных целях. Эти сообщения передаются оконной процедуре, и если она их не обработает, то направляются обработчику сообщений, устанавливаемому по умолчанию.

В принципе ничто не мешает вам использовать эти сообщения в собственных целях. Но будьте осторожны! В будущих версиях Windows может измениться способ использования этих сообщений, а некоторые могут быть вообще удалены. Исходя из своего опыта, могу сказать, что недокументированные возможности интересно изучать, но опасно использовать в программных продуктах, предназначенных для широкого распространения.

# ПРИЛОЖЕНИЕ 3

# Описание сопроводительной дискеты

Сопроводительная дискета содержит самораспаковывающийся архив файлов с примерами различных приложений, как описанных, так и не описанных в книге.

## Установка примеров

Установку примеров на ваш компьютер можно произвести двумя способами:

1. Вставьте дискету в дисковод и запустите находящийся на ней файл Install.exe.
2. В появившемся блоке диалога укажите путь к папке, в которую необходимо установить примеры, и нажмите кнопку Extract.

Либо

1. Скопируйте файл Install.exe с дискеты в папку, в которую будут установлены примеры и запустите его.
2. В появившемся блоке диалога нажмите кнопку Extract.

## Использование примеров

Полные тексты программ приведены в одноименных каталогах.

- Для использования примеров, а также для их модификации или перекомпиляции требуется Visual C++ версии не ниже 5.0.

**Внимание!**

Несовместимость русскоязычной и американской (пан-европейской) версий Windows в поддержке национальных языков требует для использования данных примеров в русскоязычной версии Windows включить директивы "Russian Resources" в исходные файлы ресурсов всех приложений до компиляции. Изменения следует произвести вне системы разработки Visual C++. Добавляются 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 блоков диалога будет невозможно прочитать, а русский текст в файлах ресурсов после компиляции будет заменен символами вопроса (?) и подчеркивания (_).

- Для просмотра содержимого файлов необходимо пользоваться *только* средствами Windows, т. к. комментарии, добавленные в тексты программ автором, написаны на русском языке, естественно, в кодировке, отличной от той, что используется в MS-DOS.

# Список приложений

- Clock — отображение действующих часов. В этом примере использовано 90% всех возможностей графического вывода.
- DlgSmpl — приложение, главным окном которого является блок диалога.
- First — минимальное приложение на базе классов библиотеки MFC.
- Graph — простейший графический редактор, где также рассматриваются вопросы создания и работы с MDI.
- MDI — пример многодокументного приложения.
- Note'n'Draw (Notedraw) — MDI-приложение на базе классов библиотеки MFC, реализующих архитектуру "документ/представление" (document/view).
- Pro — пример, иллюстрирующий работу с наборами свойств и элементами управления.
- Styles — использование различных стилей при создании окон.
- Template — создание блока диалога непосредственно в памяти, т. е. без использования ресурсов.
- DB — демонстрация принципов разработки приложения для работы с базами данных.

Кроме того, на дискете имеются примеры, которые не нашли своего отражения на страницах книги, но могут оказаться полезными при разработке собственных приложений.

# Предметный указатель


## ::

::GetClassLong 76
::GetSystemMetrics 61
::GetWindowLong 76
::PeekMessage 84
::RegisterWindowMessage 92, 99
::SetClassLong 72, 75
::SetWindowLong 73, 76
::TimerProc 227

## A

AfxFormatString1 25
AfxFormatString2 26
AfxFreeLibrary 23
AfxGetInstanceHandle 22
AfxGetMainWnd 22
AfxGetResourceHandle 22
AfxLoadLibrary 23
AfxMessageBox 26
AfxRegisterClass 23
AfxRegisterWndClass 22
ASSERT 24
ASSERT_KINDOF 25
ASSERT_VALID 25

## B

BEGIN_MESSAGE_MAP 87
bitmaps 116
brushes 116

## C

CAnimateCtrl
  Play 364
  Seek 365
  Stop 365
CBitmap 121, 137
  CreateBitmap 138
  CreateBitmapIndirect 138
  CreateCompatibleBitmap 138
  CreateDiscardableBitmap 138
  LoadBitmap 138
CBrush 121, 135
  CreateHatchBrush 137
  CreatePatternBrush 137
  CreateSolidBrush 136
CCheckListBox
  Enable 356
  GetCheck 356
  GetCheckStyle 356
  IsEnabled 356
  OnGetCheckPosition 357
  SetCheck 356
  SetCheckStyle 356
CCmdTarget 92
CCmdUI
  ContinueRouting 98
  Enable 98
  m_nID 98
  m_nIndex 98
  m_pMenu 98
  m_pOther 98
  m_pSubMenu 98
  SetCheck 98
  SetRadio 98
  SetText 98
CColorDialog 318
  DoModal 318
  GetColor 318
  GetSavedCustomColors 319
CControlBar
  EnableDocking 210
CCreateContext 431
CCtrlView 461
  m_dwDefaultStyle 460
  m_strClass 460

CDatabase 512
CanUpdate 515
Close 514
ExecuteSQL 515
GetConnect 515
IsOpen 515
Open 513
OpenEx 513
CDataExchange
m_bSaveAndValidate 311
CDC 119, 154
BeginPath 173
BitBlt 176
CClientDC 120
Chord 172
CMetaFileDC 120
CPaintDC 119
CreateCompatibleDC 156
CreateDC 155
CWindowDC 120
DPtoLP 169
DrawFocusRect 172
DrawFrameControl 170
DrawIcon 170
DrawState 170
DrawText 181
Ellipse 172
EndPath 173
EnumObjects 158
ExcludeClipRect 169
ExcludeUpdateRgn 169
ExtFloodFill 178
ExtTextOut 180
FillRect 170
FillRgn 169
FillSolidRect 170
FrameRgn 169
GetBrushOrg 158
GetClipBox 169
GetCurrentBitmap 157
GetCurrentBrush 157
GetCurrentFont 157
GetCurrentPalette 157
GetCurrentPen 157
GetDeviceCaps 157
GetHalftoneBrush 160
GetOutputTextMetrics 183
GetPixel 176
GetTextColor 163
GetTextMetrics 183
IntersectClipRect 169
InvertRect 170
InvertRgn 169
LPtoDP 169
m_hAttribDC 154
m_hDC 154
PaintRgn 169
PatBlt 176
Pie 172
Polygon 172
PolyPolygon 172
PtVisible 169
Rectangle 172
RoundRect 172
SelectClipPath 175
SelectClipRgn 169
SelectObject 159
SelectPalette 160
SelectStockObject 159
SetBkMode 160
SetBrushOrg 158
SetMapMode 164
SetPixel 176
SetPolyFillMode 160
SetROP2 161
SetStretchBltMode 162
SetTextColor 163
SetViewportExt 167
SetViewportOrg 165
SetWindowExt 167
SetWindowOrg 165
StretchBlt 178
UpdateColors 160
CDialog 294
Create 298
CreateIndirect 299
DoModal 295
EndDialog 300
GotoDlgCtrl 301
InitModalIndirect 298
MapDialogRect 301
NextDlgCtrl 301
OnCancel 300
OnInitDialog 299
OnOK 300
OnSetFont 300
PrevDlgCtrl 301

CDocManager
m_templateList 428
CDocTemplate
AddDocument 419
CDocTemplate 419
CreateNewDocument 422
GetDocString 420
GetFirstDocPosition 423
GetNextDoc 423
LoadTemplate 419
MatchDocType 422
OpenDocumentFile 423
RemoveDocument 419
SetDefaultTitle 423
CDocument 442
AddView 442
CanCloseFrame 444
DeleteContents 444
GetDocTemplate 442
GetFirstViewPosition 442
GetNextView 442
OnNewDocument 444
OnOpenDocument 445
OnSaveDocument 445
RemoveView 442
UpdateAllViews 443
CEditView 462
dwStyleDefault 462
FindText 463
GetPrinterFont 462
GetSelectedText 463
LockBuffer 463
OnReplaceAll 464
OnReplaceSel 464
OnTextNotFound 464
PrintInsideRect 462
SerializeRaw 463
SetPrinterFont 462
UnlockBuffer 463
CFieldExchange
IsFieldType 534
SetFieldType 536
CFont 121, 127
CreateFont 133
CreateFontIndirect 128
CreatePointFont 135
CFrameWnd 58
ActivateFrame 431
CreateView 432
DockControlBar 210
EnableDocking 210
FloatControlBar 211
GetActiveDocument 432
GetActiveView 432
InitialUpdateFrame 432
LoadFrame 62
OnCreateClient 430
OnEndPrintPreview 498
OnSetPreviewMode 498
RecalcLayout 211
rectDefault 59
SetActiveView 432
ShowControlBar 215
CGdiObject 121
child window 51
CHOOSECOLOR 317
CListBox
AddString 351
CompareItem 355
DeleteString 351
DrawItem 355
FindString 352
FindStringExact 352
GetCount 353
GetHorizontalExtent 349
GetItemHeight 350
GetItemRect 350
GetSelItems 353
InitStorage 351
InsertString 351
ItemFromPoint 354
MeasureItem 355
ResetContent 352
SelectString 352
SetColumnWidth 350
SetHorizontalExtent 349
SetItemHeight 350
CListCtrl
Arrange 381
CreateDragImage 394
EditLabel 392
FindItem 389
GetCallbackMask 385
GetCheck 384
GetColumn 380
GetColumnOrderArray 394
GetColumnWidth 380
GetEditControl 392

CListCtrl *(прод.)*
GetFirstSelectedItemPosition 389
GetHeaderCtrl 380
GetImageList 383
GetItem 386
GetItemCount 386
GetItemData 387
GetItemPosition 388
GetItemState 387
GetNextItem 388
GetNextSelectedItem 389
GetSelectedCount 386
GetSelectionMark 386
GetStringWidth 378
GetWorkAreas 382
HitTest 390
InsertColumn 379
RedrawItems 381
SetCallbackMask 385
SetCheck 384
SetColumn 380
SetColumnOrderArray 394
SetColumnWidth 380
SetImageList 383
SetItem 386
SetItemCount 386
SetItemCountEx 384
SetItemData 387
SetItemPosition 388
SetItemState 387
SetSelectionMark 386
SetWorkAreas 382
SortItems 391
SubItemHitTest 390
Update 381
CMDIChildWnd
Create 271
GetMDIFrame 272
m_hMenuShared 271
MDIActivate 272
MDIDestroy 271
MDIMaximize 272
MDIRestore 272
CMDIFrameWnd
GetWindowMenuPopup 270
MDIActivate 269
MDICascade 270
MDIGetActive 269
MDIIconArrange 269
MDIMaximize 269
MDINext 270
MDIRestore 270
MDISetMenu 270
MDITile 270
CMenu 238
AppendMenu 239
CheckMenuItem 242
CheckMenuRadioItem 242
CreateMenu 238
CreatePopupMenu 238
DeleteMenu 241
DestroyMenu 239
DrawItem 257
EnableMenuItem 241
InsertMenu 240
LoadMenu 238
m_hMenu 238
MeasureItem 256
ModifyMenu 241
RemoveMenu 241
SetMenuItemBitmaps 242
TrackPopupMenu 243
CMultiDocTemplate
m_docList 424
m_hAccelTable 424
m_hMenuShared 424
m_nUntitledCount 424
CRgn 122, 143
CombineRgn 145
CreateEllipticRgn 144
CreatePolygonRgn 144
CreatePolyPolygonRgn 145
CreateRectRgn 144
CreateRectRgnIndirect 144
CreateRoundRectRgn 145
CompareFunc 391, 409
Confidence 422
CPalette 121
CPen 121, 139
CreatePen 139
CPrintInfo 485
CPrintPreviewState 497
CProgressCtrl
OffsetPos 367
SetPos 366
SetRange 366
SetRange32 366
SetStep 366
StepIt 366

CPropertyPage 339
OnApply 340
OnCancel 341
OnKillActive 340
OnOK 340
OnSetActive 339
OnWizardBack 342
OnWizardFinish 343
OnWizardNext 342
SetModified 339
CPropertySheet 325
AddPage 331
Construct 326
Create 328
DoModal 328
EnableStackedTabs 337
EndDialog 332
GetTabControl 332
RemovePage 331
SetWizardMode 342
CREATESTRUCT 66
CRecordset 518
AddNew 524
CanAppend 522
CanBookmark 522
Cancel 528
CancelUpdate 526
CanRestart 522
CanUpdate 523
Close 522
Delete 525
DoFieldExchange 530
Edit 525
GetBookmark 528
GetDefaultConnect 531
GetDefaultSQL 531
GetRecordCount 523
GetSQL 523
GetStatus 523
GetTableName 523
IsBOF 524
IsDeleted 524
IsEOF 524
IsFieldDirty 528
IsFieldNull 529
IsFieldNullable 529
IsOpen 524
m_nFields 516
m_nParams 516
m_pDatabase 517
m_strFilter 517
m_strSort 517
Move 526
MoveFirst 527
MoveLast 527
MoveNext 528
MovePrev 528
Open 518
Requery 529
SetBookmark 528
SetFieldDirty 529
SetFieldNull 529
SetParamNull 530
Update 525
CRecordView 532
IsOnFirstRecord 533
IsOnLastRecord 533
OnGetRecordset 532
OnMove 534
CScrollView 467
GetDeviceScrollPosition 469
GetDeviceScrollSizes 469
GetScrollPosition 469
GetTotalSize 469
ResizeParentToFit 470
ScrollToPosition 469
SetScaleToFitSize 469
SetScrollSizes 468
CSpinButtonCtrl
SetAccel 374
SetBase 374
SetRange 373
SetRange32 373
CSplitterWnd
Create 474
CreateStatic 475
CreateView 476
CStatusBar 221
CommandToIndex 224
Create 222
CreateEx 223
GetStatusBarCtrl 224
SetIndicators 223
SetPaneText 224
CToolBar
Create 203
CreateEx 205
GetButtonStyle 211

CToolBar *(прод.)*
GetItemRect 208
GetToolBarCtrl 212
LoadBitmap 207
LoadToolBar 207
SetButtonStyle 211
SetButtonText 209
SetSizes 208
CToolBarCtrl
GetButtonCount 214
CTreeCtrl
DeleteItem 403
Expand 406
GetChildItem 405
GetCount 400
GetImageList 399
GetIndent 399
GetItem 404
GetItemData 405
GetItemImage 404
GetItemRect 405
GetItemState 404
GetItemText 405
GetNextItem 407
GetNextSiblingItem 407
GetParentItem 407
GetPrevSiblingItem 407
GetVisibleCount 400
HitTest 408
InsertItem 400
ItemHasChildren 405
Select 406
SelectDropTarget 406
SelectItem 406
SetImageList 399
SetIndent 399
SetItem 404
SetItemData 405
SetItemImage 405
SetItemState 404
SetItemText 405
SortChildren 409
SortChildrenCB 409
CView 456
DoPreparePrinting 486
DoPrintPreview 497
GetDocument 456
OnActivateFrame 458
OnActivateView 458
OnBeginPrinting 487
OnDraw 459
OnEndPrinting 488
OnFilePrintPreview 496
OnInitialUpdate 457
OnPrepareDC 488
OnPreparePrinting 484
OnPrint 489, 491
OnScroll 459
OnScrollBy 459
OnUpdate 457
CWinApp 45
AddDocTemplate 426
CloseAllDocuments 427
CreatePrinterDC 483
ExitInstance 46
GetFirstDocTemplatePosition 427
GetNextDocTemplate 427
GetPrinterDeviceDefaults 482
InitInstance 45
m_pDocManager 428
OnFileNew 427
OnFileOpen 427
OnFilePrintSetup 480
OnIdle 150
OpenDocumentFile 427
SaveAllModified 427
CWnd 55
BringWindowToTop 275
Create 49
CreateEx 49
Default 99
DoDataExchange 310
DrawMenuBar 243
EnableWindow 81
FindWindow 275
GetClientRect 285
GetCurrentMessage 99
GetDlgItem 301
GetExStyle 78
GetForegroundWindow 275
GetSafeWnd 77
GetStyle 78
IsIconic 189, 275
IsWindowEnable 81
KillTimer 225
ModifyStyle 78
MoveWindow 74
OnInitMenu 247
OnInitMenuPopup 247
OnMenuSelect 248

CWnd *(прод.)*
OnPaint 119
OnSysCommand 246
PostMessage 99
PreCreateWindow 65
PreTranslateMessage 99
SendMessage 99
SetForegroundWindow 275
SetTimer 225
SetWindowPos 62, 68
ShowWindow 49
UpdateData 310

## D

DECLARE_DYNAMIC 23
DECLARE_DYNCREATE 24
DECLARE_MESSAGE_MAP 87
DECLARE_SERIAL 24
DLGITEMTEMPLATE 296
DLGTEMPLATE 295
DRAWITEMSTRUCT 259
Dynaset 509

## E, I

END_MESSAGE_MAP 87
IMPLEMENT_DYNAMIC 23
IMPLEMENT_DYNCREATE 24
IMPLEMENT_SERIAL 24

## L

LOGFONT 128
LV_COLUMN 379
LV_FINDINFO 389
LV_HITTESTINFO 390
LV_ITEM 377

## M

MEASUREITEMSTRUCT 260
MINMAXINFO 79

## O

ON_COMMAND_RANGE 91
ON_CONTROL_RANGE 91
ON_MESSAGE 92
ON_NOTIFY_RANGE 91
ON_REGISTERED_MESSAGE 92
ON_UPDATE_COMMAND_UI_RANGE 91
overlapped window 51

## P, R, S

popup window 51
PROPSHEETHEADER 334
Recordset 508
RECT 286
RGB 136
RUNTIME_CLASS 24
Snapshot 509

## T, V

Table 510
TBBUTTON 213
TEXTMETRIC 183
TRACE 25
TV_HITTESTINFO 408
TV_INSERTSTRUCT 401
TV_ITEM 402
TV_SORTCB 409
VERIFY 25

## А, Б, Д

Атрибуты меню 244, 245
Битовые массивы 116
Динамический набор 509

## И

Извещения
  NM_CLICK 91
  NM_DBLCLK 91
  NM_KILLFOCUS 91
  NM_OUTOFMEMORY 91
  NM_RCLICK 91
  NM_RDBLCLK 91
  NM_RETURN 91
  NM_SETFOCUS 91

## К

Карандаши 116
  PS_COSMETIC 141
  PS_GEOMETRIC 141
Кисти 116
Контекст
  класса 114
  общий 114
  устройства 118
  частный 114, 115
Контуры 116
Координаты блока диалога 301

## Л, М, Н

Логические палитры 116
Меню контекстное (плавающее ) 237
Моментальный снимок 509
Набор записей 508

## О

Область вывода
  логическая 164
  физическая 164
Окна
  всплывающие 51, 54
  вспомогательные 51
  дочерние 51, 55
  перекрывающиеся 51, 54

## П, Р

Параметры размера и положения 68
Регионы 116
Режим соединения линий
  PS_JOIN_BEVEL 141
  PS_JOIN_MITER 141
  PS_JOIN_ROUND 141
Режимы
  заливки контуров 117
  масштабирования 117
  отображения 117
  сжатия 117
Режимы закрашивания многоугольников
  ALTERNATE 161
  WINDING 161

## С

Системные графические объекты 159
Сообщения
  ON_UPDATE_COMMAND_UI 97
  UDN_DELTAPOS 370
  WM_CREATE 74, 93, 127, 276
  WM_CTLCOLOR 347
  WM_DRAWITEM 256
  WM_GETMINMAXINFO 79
  WM_INITDIALOG 299
  WM_LBUTTONDOWN 229
  WM_LBUTTONUP 89, 229
  WM_MEASUREITEM 256
  WM_MENUSELECT 248
  WM_MOUSEMOVE 229, 231
  WM_NOTIFY 90, 410
  WM_PAINT 119, 149, 284
  WM_RBUTTONDOWN 255
  WM_SETFONT 298
  WM_SYSCOMMAND 246
Стили окна 51

## Т, Ш

Таблица 510
Шаблон диалога
  массив данных 297
  массив заголовка 297, 298
  массив класса 297
  массив меню 297
Шрифты 116