Министерство образования Российской Федерации

Томский государственный университет
систем управления и радиоэлектроники

УДК 681.3.06

Боровской И.Г.

ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНЫХ СИСТЕМ

Учебное пособие
Издание второе, переработанное

Рекомендовано Сибирским региональным учебно-методическим центром
высшего профессионального образования для межвузовского
использования в качестве учебного пособия для студентов
технических и экономических специальностей
всех форм обучения

2012

Содержание

стр

Введение 10

Часть 1. Общие сведения о технологии программирования 12

1. Задачи технологии программирования 12

1.1. Базовые определения 12
1.2. Невозможность доказательства отсутствия программных ошибок 13
1.3. Надежность программной системы 14
1.4. Технология программирования как способ создания надежных программных систем 15
1.5. Этапы развития технологии программирования 16
1.6. Технология программирования и информатизация общества 17

2. Общие принципы разработки программных систем 19

2.1. Специфика разработки программных систем 19
2.2. Основные подходы при создании ПС 19
2.3. Жизненный цикл программной системы 20
2.4. Понятие качества программной системы 22
2.5. Обеспечение надежности – основной критерий разработки программных систем 23
2.5.1. Методы борьбы со сложностью 24
2.5.2. Обеспечение точности перевода 25
2.5.3. Преодоление барьера между пользователем и разработчиком 26
2.5.4. Контроль принимаемых решений 26

3. Архитектура программной системы 27

3.1. Понятие архитектуры программной системы 27
3.2. Основные классы архитектур программных систем 27
3.3. Архитектурные функции 29

4. ТЕСТИРОВАНИЕ И ОТЛАДКА ПРОГРАММНОЙ СИСТЕМЫ 31

4.1. Основные понятия 31
4.2. Принципы и виды отладки программной системы 31
4.3. Заповеди отладки программной системы 33
4.4. Автономная отладка программной системы 34
4.5. Комплексная отладка программной системы 37

Часть 2. Windows программирование на основе API 40

5. Основные характеристики Windows платформ 40

5.1. Краткая история Windows ... 40
5.2. Отличия и общие свойства Windows платформ ... 42
5.2.1. Общие свойства Windows платформ ... 42
5.2.2. Отличия Win32 платформ ... 42
5.2.3. Окна и сообщения ... 45
5.2.4. Приложения, потоки и окна ... 45
5.2.5. Классы окон ... 46
5.2.6. Венгерская нотация ... 47
5.2.7. Типы сообщений ... 49
5.2.8. Сообщения и многозадачность ... 51
5.2.9. Очереди сообщений ... 51
5.2.10. Процессы и потоки ... 52
5.2.11. Потоки и сообщения ... 52
5.2.12. Оконная функция – функция обратного вызова ... 53
5.2.13. Синхронные и асинхронные сообщения ... 53
5.2.14. Функции Windows ... 54
5.2.15. Другие API ... 58
6. Структура Windows приложений ... 60
6.1. Файлы заголовков ... 61
6.2. Глабальные переменные ... 61
6.3. Точка входа в приложение ... 61
6.4. Необходимые переменные ... 62
6.5. Регистрация класса окна ... 62
6.6. Создание главного окна ... 63
6.7. Отображение главного окна ... 64
6.8. Цикл обработки сообщений ... 65
6.9. Функция окна ... 65
7. Обработка сообщений в оконной функции ... 67
7.1. Создание окна WM_CREATE ... 67
7.2. Определение размера окна WM_SIZE ... 68
7.3. Отображение содержимого окна WM_PAINT ... 68
7.3.1. Случаи генерации сообщения WM_PAINT ... 69
7.3.2. Особенность сообщения WM_PAINT ... 70
7.3.3. Правила обработки WM_PAINT ... 70
7.3.4. Отрисовка вне WM_PAINT ... 71
7.3.5. Определение возможностей контекста устройства ... 71
7.3.6. Системные метрики ... 72
7.4. Определение расположения окна WM_MOVE ... 72
7.5. Использование оконных полос прокрутки ... 73
7.5.1. Диапазон и положение полос прокрутки ... 73

7.5.2. Сообщения полос прокрутки 75
7.6. Клавиатурный ввод 76
7.6.1. Фокус ввода и активное окно 77
7.6.2. Генерация клавиатурных сообщений 77
7.6.3. Аппаратные сообщения 78
7.6.3.1. Системные аппаратные сообщения 78
7.6.3.2. Несистемные аппаратные сообщения 79
7.6.3.3. Битовые поля параметра lParam 79
7.6.3.4. Виртуальные коды клавиш 80
7.6.4. Символьные сообщения 80
7.6.5. Обработка сообщения WM_CHAR 81
7.6.6. Определение состояния управляющих клавиш 81
7.6.7. Наборы символов 82
7.7. Системный таймер 83
7.7.1. Использование таймера. Первый способ 84
7.7.2. Использование таймера. Второй способ 85
7.8. Удаление окна, сообщение WM_DESTROY 86
8. Ресурсы приложения и их использование 88
8.1. Меню приложения 89
8.1.1. Виды меню 89
8.1.2. Возможные состояния пунктов меню 89
8.1.3. Сообщения от пунктов меню 90
8.1.4. Создание главного меню приложения 91
8.1.5. Функции для работы с меню 92
8.2. Стандартные элементы управления 93
8.2.1. Создание стандартных элементов управления 94
8.2.2. Разрушение элементов управления 95
8.2.3. Функции для работы с элементами управления 95
8.2.4. Сообщения дочерних окон 96
8.2.5. Сообщения родительского окна дочерним окнам 96
8.2.6. Расширенное управление дочерними окнами 97
8.2.7. Оконные процедуры элементов управления 97
8.2.8. Элемент управления кнопка 98
8.2.8.1. Стили кнопок 99
8.2.8.2. Сообщения от кнопок, получаемые родительским окном 99
8.2.8.3. Сообщения от родительского окна к кнопке 100
8.2.8.4. Кнопки-переключатели 100
8.2.8.5. Сообщение от переключателей 101
8.2.8.6. Сообщение от родительского окна к переключателям 101
8.2.9. Структура DRAWITEMSTRUCT 101

8.2.10. Стандартный элемент управления окно ввода 102
8.2.10.1. Стили окна редактирования 103
8.2.10.2. Сообщения от редактора к родительскому окну 103
8.2.10.3. Сообщения от родительского окна к редактору 104
8.2.11. Стандартный элемент управления статический текст 106
8.2.11.1. Стили элемент управления STATIC 106
8.2.11.2. Сообщения от статического элемента управления 106
8.2.11.3. Сообщения от родительского окна к STATIC 107
8.2.12. Стандартный элемент управления список 107
8.2.12.1. Стили элемента управления список 107
8.2.12.2. Сообщения от списка к родительскому окну 108
8.2.12.3. Сообщения от родительского окна к списку 108
8.2.13. Стандартный элемент управления список с вводом 110
8.2.13.1. Стили элемента управления combobox 110
8.2.13.2. Сообщения от combobox к родительскому окну 111
8.2.13.3. Сообщения от родительского окна к combobox 112
9. Создание и использование диалоговых окон 114
9.1. Этапы создания диалога 114
9.1.1. Создание шаблона диалога 114
9.1.2. Функция диалога 115
9.1.2.1. Сходства между диалоговой функцией и оконной процедурой 115
9.1.2.2. Различия между диалоговой функцией и оконной процедурой 115
9.2. Типы диалоговых панелей 116
9.3. Создание модального диалога 116
9.4. Закрытие модального диалога 117
9.5. Окна сообщений. 118
9.6. Немодальные диалоги 119
9.7. Диалоговые окна общего пользования 120
10. Управление файлами 123
10.1. Доступ к файловой системе 124
10.2. Потоковый ввод/вывод 124
10.3. Функции ядра Windows для работы с файлами 125
10.4. Специализированные функции для работы с файлами 128
11. Печать документов 129
11.1. Последовательность печати документа 129
11.2. Контекст устройства принтера 129
11.3. Диалог отмены печати 130
11.4. Запуск процесса печати 132

11.5. Печать страницы документа ... 132
11.6. Завершение печати документа ... 133
12. Процессы и потоки ... 134
12.1. Диспетчеризация потоков ... 135
12.2. Проблемы многопоточной технологии ... 136
12.3. Создание рабочего потока ... 137
12.4. Организация взаимодействия потоков ... 138
12.4.1. Рабочий поток – процесс ... 138
12.4.2. Процесс – рабочий поток ... 139
12.5. Общая схема взаимодействия потоков ... 140
13. Приложение "Тестер файлов" ... 143
13.1. Функция WinMain() ... 144
13.2. Функция главного окна ... 146
13.3. Вспомогательные функции ... 149
13.4. Функция рабочего потока ... 152
Часть 3. Windows программирование на основе MFC ... 156
14. Введение в Visual C++ ... 156
14.1. Интерфейс вызовов функций в Windows ... 157
14.2. Библиотеки динамической загрузки ... 157
14.3. Преимущества использования MFC ... 158
14.4. Основные компоненты MFC приложения ... 159
14.5. Архитектура MFC приложения ... 159
14.6. Наследование – главный принцип построения MFC приложений ... 160
14.7. Каркас приложения ... 161
14.8. Проект приложения ... 162
14.9. Типы мастеров проектов ... 162
14.10. Имена, используемые в MFC ... 163
15. Краткий обзор классов MFC ... 166
15.1. CObject – самый базовый класс библиотеки MFC ... 166
15.2. CCmdTarget – основа структуры приложения MFC ... 166
15.3. Классы CWinThread и CWinApp – подзадачи приложения ... 166
15.4. CDocument – документ приложения ... 167
15.5. CDocTemplate, CSingleDocTemplate и CMultiDocTemplate – шаблоны документов ... 167
15.6. CWnd – базовый класс окон ... 167
15.7. CFrameWnd – окно обрамляющей рамки ... 168
15.8. CView – окна просмотра ... 168

15.9. CDialog – диалоговые панели 169
15.10. Семейство классов - элементов управления..... 170
15.11. Управляющие панели..... 170
15.12. CMenu – класс меню приложений 171
15.13. Массивы, списки, словари..... 171
15.14. Файловая система..... 173
15.15. Контекст отображения..... 174
15.16. Объекты графического интерфейса..... 175
15.17. Исключения 175
15.18. Классы, не имеющие базового класса..... 177
16. Структура простейших MFC приложений..... 180
16.1. Приложение без главного окна 180
16.2. Приложение с главным окном 181
16.3. Обработка сообщений 183
16.3.1. Оконные сообщения..... 184
16.3.2. Сообщения от элементов управления..... 184
16.3.3. Командные сообщения 184
16.3.4. Таблица сообщений 185
16.3.5. Обработчики сообщений 186
17. Диалоговые панели в MFC приложениях 193
17.1. Этапы создания диалоговых панелей..... 193
17.2. Модальная диалоговая панель 194
17.2.1. Шаблон диалога..... 194
17.2.2. Класс диалоговой панели и его реализация 194
17.2.3. Отображение модальной диалоговой панели..... 200
17.3. Немодальная диалоговая панель..... 201
18. Поддержка MFC новых элементов управления 204
18.1. Элемент CRichEditCtrl 205
18.1.1. Создание элемента RichEdit 205
18.1.2. Загрузка текста в RichEdit 206
18.1.3. Форматирование текста..... 206
18.1.4. Форматирование параграфов 207
18.1.5. Сохранение в файле текста..... 207
18.2. Элемент управления CListCtrl 208
18.2.1. Предварительные замечания 208
18.2.2. Создание элемента ListCtrl..... 209
18.2.3. Инициализация списка 210
18.2.4. Выборка из списка..... 211
18.2.5. Пересортировка списка..... 213
18.3. Элемент CTreeCtrl..... 215

18.3.1. Предварительные замечания ... 215
18.3.2. Создание элемента ... 216
18.3.3. Инициализация элемента ... 217
18.3.4. Вставка объектов ... 218
18.3.5. Позиционирование ... 219
18.3.6. Динамическое изменение образа объекта ... 219
18.3.7. Динамический отклик на пользовательский выбор ... 220
18.3.8. Отслеживание нажатий кнопок мыши ... 220
18.3.9. Редактирование меток объектов ... 221
18.4. Элемент CSpinButtonCtrl ... 223
19. Архитектура Document-View ... 225
19.1. Пользовательский интерфейс ... 225
19.2. Документ и его представления ... 225
19.3. Создание документов и его отображений ... 226
19.4. Взаимодействие документа и его видов ... 228
19.5. SDI-приложение ... 228
19.5.1. Ресурсы приложения ... 229
19.5.2. Динамическая замена одного объекта вид другим ... 230
19.5.3. Расщепление главного окна приложения ... 233
19.6. MDI-приложение ... 234
19.6.1. Ресурсы приложения ... 235
19.6.2. Классы приложения ... 235
19.7. Дополнительные компоненты MFC приложений ... 239
19.7.1. Панель управления ... 239
19.7.2. Дополнительные возможности панели управления ... 240
19.7.3. Панель состояния ... 241
19.7.4. Индикаторы панели состояния ... 242
19.7.5. Дополнительные возможности панели состояния ... 242
20. DLL и MFC ... 245
20.1. Библиотеки динамической компоновки ... 245
20.2. Библиотеки импортирования ... 245
20.3. Согласование интерфейсов ... 246
20.4. Загрузка неявно подключаемой DLL ... 246
20.5. Динамическая загрузка и выгрузка DLL ... 247
20.5.1. Загрузка обычной DLL ... 247
20.5.2. Выгрузка DLL ... 248
20.5.3. Библиотеки ресурсов ... 249
20.6. Создание DLL ... 249
20.6.1. Загрузка MFC расширений динамических библиотек ... 250
20.6.2. Функция DllMain ... 250

20.6.3. Экспортирование функций из DLL 251
20.6.4. Файл определения модуля 252
20.6.5. Экспортирование классов 252
20.6.6. Статическая память DLL 253
20.7. Использование MFC в DLL 254
20.7.1. Обычные MFC DLL 254
20.7.2. Управление информацией о состоянии MFC 254
20.7.3. Динамические расширения MFC 255
20.7.4. Инициализация динамических расширений 255
20.7.5. Загрузка динамических расширений MFC 256
20.7.6. Экспортирование классов из динамических расширений 256
Вместо заключения – API, MFC, что же дальше? 258
Литература к первой части 259
Литература ко второй части 259
Литература к третьей части 259

Введение

Настоящее пособие состоит из трех частей. В первой части рассматриваются общие принципы технологии программирования. Здесь даются базовые понятия, рассматривается полный “цикл жизни” программы, а также освещаются основные вопросы, касающиеся создания надежных программных систем. Несмотря на некоторую “академичность” этой части, ее можно рассматривать как набор обоснованных правил и общих рекомендаций по созданию “хороших программ”, без привязки к конкретной операционной системе. Следуйте этим правилам, чтобы избежать возможных затруднений при разработке собственных программ.

Вторая часть посвящена технологии программирования применительно к операционной системе Windows. Здесь детально рассматриваются все аспекты взаимодействия операционной системы и пользовательских приложений. Особое значение уделяется вопросам, связанным с оптимальным использованием ресурсов операционной системе Windows. Освоив материал этой части, вы сможете создать Windows приложение любой сложности.

В третьей части излагается технология Windows программирования на базе MFC в среде Microsoft Visual C++.

В первую очередь заметим, что язык C++ обладает выдающейся гибкостью и позволяет делать буквально все, при этом заслуженно считается, что Visual C++ является самой подходящей средой для создания наиболее эффективных приложений. Такие широкие возможности часто вызывают неуверенность в собственных силах у неопытного разработчика и, как следствие, могут создать ситуацию, когда в исходный код закрадываются трудноуловимые ошибки, поиск которых требует длительного времени. Многие оппоненты C++, особенно приверженцы RAD систем, таких как Visual Basic или Delphi, отмечают это обстоятельство как главный недостаток C++. Действительно, изучение языка C++ требует большего времени, а программирование на нем – внимательности и аккуратности. Но наградой за это будет компактное и быстрое приложение, не уступающее по эффективности коду, разработанному на “чистом” ассемблере. В ответ можно услышать, что создание быстрого кода больше не требуется, поскольку неэффективность и медлительность программы компенсируется повышенной производительностью современных компьютеров. Но это весьма спорный аргумент, а вот тот факт, что существуют задачи программирования, которые могут быть решены только средствами C/C++ остается бесспорным. Не могут RAD системы использоваться для создания, например, сервисов операционной системы или драйверов виртуальных устройств. Только используя C++, вы получите полную свободу при решении *любой* задачи программирования.

Но почему следует использовать именно Visual C++ и MFC, если вам требуется создавать Windows приложения. Существуют же другие среды программирования, например, Borland C++ фирмы Inprase или Watcom C++ фирмы Watcom. Наиболее веским доводом в пользу Visual C++ является то, что корпорация Microsoft, является разработчиком как Visual C++, так и операционных систем линейки Windows, и, следовательно, имеет наилучшее представление о том, что и как функционирует внутри операционной системы. Кроме того, библиотека MFC является неотъемлемым системным компонентом объектно-ориентированной операционной системы Windows, она лицензирована перечисленными выше фирмами для использования в собственных программных продуктах. Будем считать это последним аргументом в пользу изучения именно Visual C++ и MFC.

Часть 1. Общие сведения о технологии программирования

1. Задачи технологии программирования

Прежде, чем мы приступим к вопросам, непосредственно касающихся технологии программирования вообще, и Windows программирования в частности, необходимо выяснить значения ряда основных терминов и понятий. Например, ввести понятие компьютерной программы как формализованное описание процесса обработки данных; уяснить значение термина программная система и многое другое.

1.1. Базовые определения

Итак, цель программирования состоит в описании процессов обработки некоторых *данных*.

Данные – это представление фактов и идей в формализованном виде, пригодном для передачи и переработке в некоем процессе, а информация – это смысл, который придается данным при их представлении [1].

Обработка данных – это выполнение систематической последовательности действий с данными. Данные представляются и хранятся на так называемых носителях данных.

Информационная среда – это совокупность носителей данных, используемых при какой-либо обработке данных.

Состояние информационной среды – это набор данных, содержащихся в какой-либо момент в информационной среде. Тогда понятие *процесса* можно определить как последовательность сменяющих друг друга состояний некоторой информационной среды. Описать процесс означает определить последовательность состояний заданной информационной среды.

Если необходимо, чтобы по заданному описанию требуемый процесс порождался автоматически на каком-либо компьютере, необходимо, чтобы это описание было формализованным. Такое описание называется *программой*.

Однако программа должна быть понятной и человеку, так как и при разработке программ, и при их использовании приходится выяснять, какой именно процесс она порождает. Поэтому программа составляется на удобном для человека формализованном *языке программирования*, с которого она автоматически переводится на язык соответствующего компьютера с помощью другой программы, называемой *транслятором*.

Разработчику, прежде чем составить программу на удобном для него языке программирования, приходится проделывать большую подготовительную работу по уточнению постановки задачи, выбору метода ее решения, выяснению специфики применения требуемой программы, определению общей организации разрабатываемой программы и многое другое.

Использование этой информации может существенно упростить задачу понимания программы человеком, поэтому весьма полезно ее как-то фиксировать в виде отдельных документов, зачастую не формализованных, а рассчитанных только для восприятия человеком.

Как правило, программы разрабатываются в расчете на то, что ими будут пользоваться люди, не только не участвующие в их разработке, но и вообще далекие от вопросов программирования. Поэтому таким пользователям для освоения программы требуется определенная дополнительная документация. Программа или логически связанная совокупность программ на носителях данных, снабженная программной документацией, называется *программной системой* (ПС).

Программа позволяет осуществлять некоторую автоматическую обработку данных на компьютере, тогда как программная документация позволяет понять, какие функции выполняет та или иная программа из состава ПС, как подготовить исходные данные и запустить требуемую программу в процесс ее выполнения, а также, что немало важно, как трактовать получаемые результаты. Программная документация несет и еще одну функцию, а именно, она помогает разобраться в самой программе, что необходимо, например, при ее модификации.

1.2. Невозможность доказательства отсутствия программных ошибок

Можно считать, что итоговым продуктом технологии программирования является программная система, содержащее программы, которые выполняют набор требуемых функций. Здесь под термином “программа” будем понимать *правильно работающую программу*, т.е. программу, которая не содержит ошибок. Сложность состоит в том, что понятие *ошибки* в программе трактуется неоднозначно. Согласно работам [2,3], будем считать, что в программе имеется ошибка, если программа не выполняет того, что разумно ожидать от нее пользователю. При этом “разумное ожидание пользователя” формируется на основании документации по применению этой программы. Отсюда следует, что понятие ошибки в программе является существенно *не формальным*. В ПС программы и документация взаимно увязаны, образуют некоторую целостность. Поэтому правильнее говорить об ошибке не в программе, а в ПС в целом.

Будем считать, что в программной системе имеется ошибка, если она не выполняет того, что разумно ожидать от нее пользователю. В частности, разновидностью ошибки в ПС является несогласованность между программами ПС и документацией по их применению. В работе [3] выделяется в отдельное понятие частный случай ошибки в ПС, когда программа не соответствует своей функциональной спецификации, т.е. описанию, составляе-

мому на этапе, который предшествует непосредственному программированию. Такая ошибка именуется *дефектом* программы. Однако выделение такой разновидности ошибки в отдельное понятие вряд ли оправданно, так как причиной ошибки может оказаться сама функциональная спецификация, а не уже готовая программа.

Итак, мы сталкиваемся с ситуацией, когда задание на ПС сформулировано неформально, а понятие ошибки в ПС не формализовано. Отсюда следует, что *в общем случае нельзя* доказать формальными методами, т.е. с привлечением строгого математического аппарата, правильность ПС.

Конечно, каждое ПС проходит этап опытного тестирования, когда некоторому набору исходных данных соответствует определенный ожидаемый результат. Однако, как указал Дейкстра [4], нельзя показать правильность ПС с помощью тестирования – тестирование может лишь выявить наличие в ПС ошибок, *но не их отсутствие*. Поэтому понятие правильной ПС неконструктивно в том смысле, что после окончания работы над созданием ПС мы не сможем достоверно убедиться, что достигли итоговой цели.

1.3. Надежность программной системы

В качестве альтернативы *правильного* ПС введем термин *надежное* ПС. Под надежностью ПС будем понимать его способность безотказно выполнять определенные функции при заданных условиях в течение заданного периода времени с достаточно большой вероятностью. При этом под отказом в ПС понимают проявление в нем ошибки [2]. Таким образом, надежное ПС не исключает наличия в нем ошибок. Важно лишь то, что эти ошибки при практическом применении данного ПС в заданных условиях проявлялись достаточно редко. Убедиться, что ПС обладает таким свойством можно при его опытном тестировании, а также при практическом применении. Таким образом, в реальности мы можем разрабатывать лишь *надежные*, но не *правильные* ПС.

ПС может обладать различной степенью надежности. Как определить эту степень? Так же, как в любой отрасли техники – степень надежности характеризуется вероятностью работы ПС без отказа в течение определенного периода времени [2]. Однако в силу специфических особенностей ПС определение этой вероятности наталкивается на ряд трудностей по сравнению с решением аналогичной задачи в технике.

При оценке степени надежности ПС следует также учитывать последствия каждого отказа. Некоторые ошибки в ПС могут вызывать лишь некоторые неудобства при его применении, тогда как другие ошибки могут иметь катастрофические последствия, например, угрожать полному разрушению информации на магнитном носителе. Поэтому для оценки надежности ПС иногда используют дополнительные показатели, учитывающие сте-

пень вреда для пользователя при проявлении конкретного отказа. Позже мы еще вернемся к этому вопросу.

1.4. Технология программирования как способ создания надежных программных систем

Под технологией программирования будем понимать совокупность производственных процессов, которая приводит к созданию требуемого ПС, а также описание этой совокупности процессов. Другими словами, под технологией программирования будем понимать определенную технологию разработки программных систем, включая сюда все процессы, начиная с момента зарождения идеи этого программного средства, не упуская из виду мер, касающихся создания необходимой программной документации. Каждый процесс этой совокупности базируется на использовании определенных средств и методов. В частности, при создании компьютерных программ можно говорить о компьютерной технологии программирования. В литературе можно найти и другие определения технологии программирования.

Нужно заметить, что существуют два других понятия, довольно тесно связанных с понятием технологии программирования – это *программная инженерия* и *методология программирования*.

Программная инженерия определяется как систематический подход к разработке, эксплуатации, сопровождению и, наконец, изъятию из обращения программных систем. Главное различие между технологией программирования и программной инженерией как дисциплинами для изучения заключается в способе рассмотрения и систематизации материала. В технологии программирования основной акцент делается на изучении процессов разработки ПС, тогда как в программной инженерии большее внимание уделяется различным методам и инструментальным средствам разработки ПС с точки зрения достижения определенных целей.

Не следует также путать технологию программирования с методологией программирования [3]. В технологии программирования все методы рассматриваются “сверху”, с точки зрения организации технологических процессов, а в методологии программирования – методы рассматриваются как бы “снизу”, т.е. с точки зрения основ их построения.

Возвращаясь к технологии программирования и имея в виду, что надежность является неотъемлемым атрибутом ПС, мы будем рассматривать технологию программирования как способ разработки надежных ПС. При этом будут рассматриваться все процессы разработки ПС, начиная с момента возникновения замысла о каком-либо ПС, когда в качестве конечного продукта технологии принимается надежное ПС, которое *может* содержать ошибки.

Такой взгляд на технологию программирования будет существенно влиять на организацию технологических процессов, на выбор в них методов и инструментальных средств.

1.5. Этапы развития технологии программирования

Кратко рассмотрим этапы развития программирования, чтобы лучше понять движущие силы и перспективы дальнейшей эволюции технологии программирования.

В 50-е годы мощность компьютеров первого поколения была невелика, а программирование для них велось преимущественно в машинном коде. Главным образом решались научно-технические задачи оборонного характера, при этом задание на программирование содержало, как правило, достаточно точную математическую постановку задачи. Например, расчет траектории полета ракеты с учетом множества сопутствующих факторов. Использовалась интуитивная технология программирования, когда сразу же приступали к составлению программы по исходному заданию, при этом часто само задание несколько раз изменялось, что сильно увеличивало время разработки программы. Минимальная документация оформлялась уже после того, как программа начинала работать. Тем не менее, именно в этот период родилась фундаментальная для технологии программирования концепция модульного программирования [5], ориентированная на преодоление трудностей программирования в машинном коде. Появились первые языки программирования высокого уровня.

В 60-е годы можно было наблюдать бурное развитие и широкое использование языков программирования высокого уровня, таких как АЛГОЛ 60, ФОРТРАН, КОБОЛ, значение которых в технологии программирования явно преувеличивалась. Надежда на то, что эти языки решат все проблемы, возникающие в процессе разработки *больших программ*, не оправдалась. В результате повышения мощности компьютеров и накопления опыта программирования на языках высокого уровня быстро росла сложность решаемых на компьютерах задач, в результате чего обнаружилась ограниченность языков, проигнорировавших модульную организацию программ. Кроме того, стало понято, что важно не только то, на каком языке мы программируем, но и то, как мы программируем [4]. Это было уже началом серьезных размышлений над методологией и технологией программирования. Появление в компьютерах второго поколения прерываний привело к развитию мультипрограммирования и созданию больших программных систем. Широко стала использоваться коллективная разработка, которая, однако, вскрыла ряд серьезных технологических проблем.

В 70-е годы получили широкое распространение информационные системы и базы данных. К середине 70-х годов стоимость хранения одного

бита информации на компьютерных носителях стала меньше, чем на традиционных носителях. Это резко повысило интерес к компьютерным системам хранения данных. Появляются языки программирования высокого уровня, например, Ада, С и другие, учитывающие требования технологии программирования в полной мере [6]. Началось интенсивное развитие технологии программирования, прежде всего, в следующих направлениях:

- обоснование и широкое внедрение нисходящей разработки и структурного программирования;
- развитие абстрактных типов данных и модульного программирования, в частности, возникновение идеи разделения спецификации и реализации модулей и использование модулей, скрывающих структуры данных;
- исследование проблем обеспечения надежности и мобильности ПС;
- создание методики управления коллективной разработкой ПС;
- появление инструментальных программных средств поддержки технологии программирования.

80-е годы характеризуются широким внедрением персональных компьютеров во все сферы человеческой деятельности и тем самым созданием обширного и разнообразного контингента пользователей ПС. Это привело к бурному развитию пользовательских интерфейсов и созданию четкой концепции качества ПС. Развиваются методы и языки спецификации ПС [7]. Выходит на передовые позиции *объектный подход* к разработке ПС [8]. Создаются различные инструментальные среды разработки и сопровождения ПС. Бурно развиваются концепции компьютерных сетей.

90-е годы знаменательны широким охватом всего человеческого общества международной компьютерной сетью, персональные компьютеры стали подключаться к ней как терминалы. Остро встала проблема защиты компьютерной информации и передаваемых по сети сообщений. Стали бурно развиваться компьютерная технология разработки ПС и связанные с ней формальные методы спецификации программ. Можно сказать, что в этот период начинается решающий этап полной информатизации и компьютеризации общества.

1.6. Технология программирования и информатизация общества

Технология программирования играла разную роль на разных этапах развития программирования. По мере повышения мощности компьютеров и развития средств и методологии программирования росла и сложность решаемых задач с применением компьютеров, что и привело к повышенному вниманию к технологии программирования. Резкое удешевление стоимости компьютеров и, в особенности, стоимости хранения информации на магнитных носителях привело к широкому внедрению компьютеров практически

во все сферы человеческой деятельности, что существенно изменило направленность технологии программирования. Человеческий фактор стал играть в ней решающую роль. Сформировалось достаточно ясное понятие качества ПС, причем предпочтение стало отдаваться не столько эффективности ПС, сколько удобству работы с ним для пользователей, конечно, при равной надежности. Широкое использование компьютерных сетей привело к интенсивному развитию распределенных вычислений; появилась возможность для дистанционного доступа к удаленной информации и для электронного способа обмена сообщениями между людьми. Компьютерная техника из средства решения отдельных задач все более превращается в средство информационного моделирования реального и мыслимого мира. Все это ставит перед технологией программирования новые и достаточно трудные задачи.

2. Общие принципы разработки программных систем

2.1. Специфика разработки программных систем

1. Прежде всего, напомним о существующем противоречии, когда с одной стороны, мы имеем неформальный характер требований к ПС в виде постановки задачи, а также неформальное понятие программной ошибки, а с другой – формализованный основной объект разработки в виде итоговой программы. Тем самым разработка ПС содержит этапы формализации, а переход от неформального к формальному остается существенно неформальным.

2. Разработка ПС носит творческий характер, когда на каждом шаге приходится делать какой-либо выбор и принимать какое-либо решение. Она не сводится к выполнению какой-либо последовательности регламентированных действий. Тем самым эта разработка ближе к процессу проектирования каких-либо сложных устройств, но никак не к их массовому производству.

3. Вполне понятно, что продукт разработки представляет собой некоторую совокупность текстов в виде *статических объектов*, тогда как смысл или семантика этих текстов выражается процессами обработки данных и действиями пользователей, запускающих эти процессы, т.е. является *динамическим*. Это предопределяет выбор разработчиком ряда специфичных приемов, методов и средств.

4. Продукт разработки имеет и другую специфическую особенность – программная система при своем использовании не расходуется и не расходует дополнительных ресурсов.

2.2. Основные подходы при создании ПС

К настоящему время сложилось пять основных подходов к организации процесса создания и использования ПС [9].

Водопадный подход. При таком подходе разработка ПС состоит из цепочки этапов. На каждом этапе создаются документы, используемые на последующем этапе. В исходном документе фиксируются требования к ПС. В конце этой цепочки создаются программы, включаемые в ПС.

Исследовательское программирование. Этот подход предполагает быструю реализацию рабочих версий программ ПС, выполняющих лишь в первом приближении требуемые функции. После экспериментального применения реализованных программ производится их модификация с целью сделать их более полезными для пользователей. Этот процесс повторяется до тех пор, пока ПС не будет достаточно приемлемо для пользователей. Можно сказать, что исследовательское программирование исходит из взгля-

да на программирование как на искусство. Оно применяется тогда, когда водопадный подход не может быть использован из-за того, что не удается точно сформулировать требования к ПС.

Прототипирование. Этот подход моделирует начальную фазу исследовательского программирования вплоть до создания рабочих версий программ, предназначенных для проведения экспериментов с целью установить требования к ПС. В дальнейшем должна последовать разработка ПС по установленным требованиям в рамках какого-либо другого подхода, например, водопадного.

Формальные преобразования. Этот подход включает разработку формальных спецификаций ПС и превращение их в программы путем корректных преобразований. На этом подходе базируется, например, компьютерная CASE-технология разработки ПС.

Сборочное программирование. Этот подход предполагает, что ПС конструируется из предопределенного набора компонент. Должна существовать некоторая библиотека таких компонент, каждая из которых может многократно использоваться в разных ПС. Процесс разработки ПС при данном подходе состоит скорее из сборки программ из компонент, чем из их программирования. В качестве примера реализации данного способа можно указать такие RAD системы, как Visual Basic и Delphi.

Из всех вышеперечисленных, водопадный подход представляет наибольший интерес, поскольку в этом подходе приходиться иметь дело с большинством процессов программной инженерии. Кроме того, в рамках этого подхода создается большинство больших программных систем. Именно этот подход рассматривается в качестве индустриального подхода разработки программного обеспечения.

2.3. Жизненный цикл программной системы

Под *жизненным циклом* ПС понимают весь период его разработки и эксплуатации, начиная от момента возникновения замысла ПС и кончая прекращением всех видов его использования [10]. Жизненный цикл охватывает довольно сложный процесс создания и использования ПС. Этот процесс может быть организован по-разному для разных классов ПС и в зависимости от особенностей коллектива разработчиков.

В рамках водопадного подхода выделяют три стадии жизненного цикла ПС (рис. а), а именно: разработка ПС, производство программных изделий[1] и, наконец, эксплуатация ПС.

Стадия *разработки* ПС включает следующие этапы, при этом заметим, что всем этапам сопутствуют процессы документирования ПС:

---

1 Программное изделие – это экземпляр или копия разработанного ПС.

- Этап внешнего описания ПС включает процессы, приводящие к созданию некоторого документа, который мы будем называть внешним описанием ПС. Этот документ является описанием поведения ПС с точки зрения внешнего по отношению к нему наблюдателя с фиксацией требований относительно его качества. Внешнее описание ПС начинается с анализа и определения требований к ПС со стороны заказчика, а также включает процессы спецификации этих требований.
- Конструирование ПС охватывает процессы разработки архитектуры ПС, разработки структур программ ПС и их детальную спецификацию.
- Кодирование ПС включает процессы создания текстов программ на языках программирование, их отладку посредством тестирования. Зачастую этапы конструирования и кодирования перекрываются. Это означает, что кодирование некоторых частей программной системы может быть начато еще до завершения этапа конструирования.
- На этапе аттестации ПС производится оценка качества ПС. Если эта оценка оказывается приемлемой для практического использования ПС, то разработка ПС считается законченной.

Рис. А. Стадии и этапы жизненного цикла ПС. 1 - этап внешнего описания; 2 - конструирование; 3 - кодирование; 4 - аттестация.

Стадия *производства программных изделий* – это процесс генерации или воспроизведения программ и программных документов ПС с целью их поставки пользователю для применения по назначению. Тогда под производством программных изделий будем понимать совокупность работ по обеспечению изготовления требуемого количества копий ПС в установленные сроки [11]. Вполне очевидно, что в жизненном цикле ПС эта стадия является вырожденной, так как она представляет рутинную работу, которая может быть выполнена автоматически и без ошибок. Этим она принципиально отличается от стадии производства технических устройств.

Стадия *эксплуатации* ПС охватывает процессы хранения, внедрения и сопровождения ПС. Она состоит из двух параллельно проходящих фаз – фазы применения ПС и фазы сопровождения ПС.

Применение ПС – это использование ПС для решения практических задач на компьютере путем выполнения ее программ.

Сопровождение ПС – это процесс сбора информации о качестве ПС в эксплуатации, устранения обнаруженных в нем ошибок, доработки ПС и его модификации, а также извещения пользователей о внесенных в него изменениях [11].

2.4. Понятие качества программной системы

Каждое ПС должно обладать целым рядом свойств, которые дают возможность успешно его использовать в течение длительного периода, другими словами – программная система должно обладать определенным качеством. Под *качеством* ПС будем понимать совокупность черт и характеристик ПС, которые влияют на его способность удовлетворять заданные потребности пользователей. Это не означает, что разные ПС должны обладать одной и той же совокупностью таких свойств в их наивысшей степени. Этому препятствует тот факт, что повышение качества ПС по одному из таких свойств часто может быть достигнуто лишь ценой изменения стоимости, сроков завершения разработки и снижения качества этого ПС по другим его свойствам. Качество ПС является удовлетворительным, когда оно обладает указанными свойствами в такой степени, чтобы гарантировать успешное его использование.

Совокупность свойств ПС, которая образует удовлетворительное для пользователя качество ПС, зависит от условий и характера эксплуатации этого ПС, т.е. от позиции, с которой должно рассматриваться качество этого ПС. Поэтому при описании качества ПС, прежде всего, должны быть указаны критерии отбора требуемых свойств ПС. В настоящее время критериями качества ПС принято считать следующее шесть параметров [12]:

- *Функциональность*, т.е. способность ПС выполнять набор функций, удовлетворяющих заданным или подразумеваемым потребностям пользователей. Набор указанных функций определяется во внешнем описании ПС.
- *Надежность* подробно обсуждалась в предыдущем разделе книги.
- *Легкостью применения* обладают ПС, которые позволяют минимизировать усилия пользователя как при подготовке исходных данных, так и при использовании ПС в целом.
- *Эффективность* – это отношение уровня услуг, предоставляемых ПС пользователю при заданных условиях, к объему используемых ресурсов.

- *Сопровождаемостью* обладают ПС, которые позволяют минимизировать усилия как при внесении изменений для устранения обнаруженных ошибок, так и при его модификации.
- *Мобильность* – это способность ПС быть перенесенным из одной компьютерной платформы на другую при минимальных затратах на модификацию.

Из всех перечисленных, функциональность и надежность являются *обязательными* критериями качества ПС, причем обеспечение надежности будет красной нитью проходить по всем этапам и процессам разработки ПС. Остальные критерии используются в зависимости от потребностей пользователей в соответствии с требованиями к ПС.

2.5. Обеспечение надежности – основной критерий разработки программных систем

Рассмотрим теперь общие принципы обеспечения надежности ПС, что, как уже подчеркивалось, является основной заботой при разработке ПС. В работе [2] указаны четыре основных подхода к обеспечению надежности: 1) предупреждение ошибок; 2) самообнаружение ошибок; 3) самоисправление ошибок; 4) обеспечение устойчивости к ошибкам.

Целью подхода *предупреждения ошибок* является не допустить ошибок в готовых ПС. Указанные ранее пути возникновения ошибок при разработке ПС позволяют для достижения этой цели сконцентрировать внимание на следующих вопросах:

- борьба со сложностью,
- обеспечение точности перевода,
- преодоление барьера между пользователем и разработчиком,
- обеспечение контроля принимаемых решений.

Этот подход связан с организацией процессов разработки ПС, т.е. с технологией программирования. И хотя, как уже отмечалось, гарантировать отсутствие ошибок в ПС невозможно, но в рамках этого подхода можно достигнуть приемлемого уровня надежности ПС.

Остальные три подхода связаны с организацией самих продуктов технологии. Они учитывают возможность появления ошибки в программах.

Самообнаружение ошибки в программе означает, что программа содержит средства обнаружения отказа в процессе ее выполнения.

Самоисправление ошибки в программе означает не только обнаружение отказа в процессе ее выполнения, но и исправление последствий этого отказа, для чего в программе должны иметься соответствующие средства.

Обеспечение устойчивости программы к ошибкам означает, что в программе содержатся средства, позволяющие локализовать область влияния отказа программы, уменьшить его негативные последствия и предотвра-

тить катастрофические последствия отказа. Однако применение этих подходов весьма затруднено, поскольку многие простые методы, используемые в технике в рамках этих методов, неприменимы в программировании. Например, широко применяемое в технике дублирование отдельных блоков и устройств в нашем случае не дает никакого эффекта. Кроме того, добавление в программу дополнительных фрагментов приводит к ее значительному усложнению, что, в конечном счете, препятствует применению метода предупреждения ошибок.

2.5.1. Методы борьбы со сложностью

Прежде всего, нужно уяснить, что же скрывается за термином “сложность”. Следуя работе [4], будем выделять три интеллектуальные способности человека, очень важные при разработке ПС, а именно:

- Способность человека *к перебору* связана с возможностью последовательного переключения внимания с одного предмета на другой, позволяя узнавать искомый предмет. Эта способность весьма ограничена, в среднем человек может уверенно, не сбиваясь перебирать в пределах 1000 предметов.
- Средством преодоления этой ограниченности является другая способность *к абстракции*, благодаря которой человек может объединять разные предметы или экземпляры в одно понятие, заменять множество элементов одним элементом другого рода.
- Способность человека *к математической индукции* позволяет ему справляться с бесконечными последовательностями.

При разработке ПС человек имеет дело с системами. Под системой будем понимать совокупность взаимодействующих друг с другом элементов. ПС можно рассматривать как пример системы. Логически связанный набор программ является другим примером системы. Любая отдельная программа также является системой. Понять систему означает осмысленно перебрать все пути взаимодействия между ее элементами. В силу ограниченности человека к перебору будем различать простые и сложные системы.

Под *простой* будем понимать такую систему, в которой человек может уверенно перебирать все пути взаимодействия между ее элементами, а под *сложной* будем понимать такую систему, в которой он этого делать не в состоянии.

При разработке ПС мы не всегда можем уверенно знать обо всех связях между ее элементами из-за возможных ошибок. Поэтому полезно уметь оценивать сложность системы по числу ее элементов, т.е. числом потенциальных путей взаимодействия между ее элементами. Данная оценка имеет вид $n!$, где n - число элементов системы.

Систему назовем *малой*, если $n < 7$ ($6! = 720 < 1000$), систему назовем *большой*, если $n \geq 7$. Малая система всегда *проста*. Большая система также может оказаться простой, но в подавляющем числе случаев большая система является *сложной*. Задача технологии программирования – научиться делать большие системы простыми.

Заметим, что полученная оценка простых систем по числу элементов широко используется на практике. Так, для руководителя коллектива весьма желательно, чтобы в нем не было больше шести взаимодействующих между собой подчиненных. Весьма важно также следовать правилу: “все, что может быть сказано, должно быть сказано в шести пунктах или меньше”. Этому правилу мы будем стараться следовать в настоящей книге – всякие перечисления взаимосвязанных утверждений будут соответствующим образом группироваться и обобщаться. Полезно этому правилу следовать и при разработке ПС.

Итак, возвращаясь к вопросу борьбы со сложностью систем, укажем два таких общих метода:

Обеспечение независимости компонент системы означает разбиение системы на такие части, между которыми должны остаться по возможности меньше связей. Одним из воплощений этого метода является модульное программирование.

Использование в системах иерархических структур позволяет локализовать связи между компонентами, допуская их лишь между компонентами, принадлежащими смежным уровням иерархии. Этот метод, по существу, означает разбиение большой системы на подсистемы, образующих малую систему. Здесь существенно используется способность человека к абстрагированию.

2.5.2. Обеспечение точности перевода

Обеспечение точности перевода направлено на достижение однозначности интерпретации документов различными разработчиками и, конечно же, пользователями ПС. Это требует придерживаться при переводе определенной дисциплины решения задач. Так, весь процесс перевода можно разбить на следующие этапы:

- поймите задачу, поставленную заказчиком;
- составьте план, включая цели и методы решения;
- выполните план, проверяя правильность каждого шага;
- проанализируйте полученное решение.

2.5.3. Преодоление барьера между пользователем и разработчиком

Как обеспечить, чтобы ПС выполняла то, что пользователь разумно ожидает от нее? Для этого необходимо правильно понять, чего же хочет пользователь, а также узнать уровень его подготовки и окружающую пользователя обстановку. При разработке ПС следует привлекать предполагаемых пользователей для участия в процессах принятия решений, а также детально ознакомиться с особенностями его работы. Самое лучшее – побывайте в его “шкуре”.

2.5.4. Контроль принимаемых решений

Обязательным шагом на каждом этапе разработки ПС должна быть проверка правильности принятых решений. Это позволит обнаруживать и исправлять ошибки на самой ранней стадии после ее возникновения, что существенно снижает стоимость ее исправления и повышает вероятность полного ее устранения.

С учетом специфики разработки ПС необходимо применять везде, где это возможно, следующие два метода:

Смежный контроль означает проверку полученного документа лицами, не участвующими в его разработке, с двух сторон – со стороны автора исходного для контролируемого процесса документа, и лицами, которые будут использовать полученный документ в качестве исходного в последующих технологических процессах. Такой контроль позволяет обеспечивать однозначность интерпретации полученного документа.

Сочетание статических и динамических методов контроля означает то, что нужно не только контролировать документ как таковой, но и проверять какой процесс обработки данных он описывает. Это отражает одну из специфических особенность ПС, которую мы отмечали ранее – статическая форма при динамическом содержании.

3. Архитектура программной системы

3.1. Понятие архитектуры программной системы

Архитектура ПС – это представление программной системы как системы, состоящей из некоторой совокупности взаимодействующих подсистем. В качестве таких подсистем выступают обычно отдельные программы. Разработка архитектуры является первым этапом борьбы со сложностью ПС, на котором реализуется принцип выделения относительно независимых компонент. Основные задачи разработки архитектуры ПС заключаются в следующем:

- Выделение программных подсистем и отображение на них внешних функций, заданных во внешнем описании ПС.
- Определение способов взаимодействия между выделенными программными подсистемами.

С учетом принимаемых на этом этапе решений производится дальнейшая конкретизация функциональных спецификаций.

3.2. Основные классы архитектур программных систем

В работе [2] приводится следующая классификация архитектур программных систем:

- цельная программа;
- комплекс автономно выполняемых программ;
- слоистая программная система;
- комплекс параллельно выполняемых программ.

Цельная программа представляет вырожденный случай архитектуры ПС, поскольку в состав ПС входит только одна программа. Такую архитектуру выбирают обычно в том случае, когда ПС должно выполнять одну какую-либо ярко выраженную функцию и ее реализация не представляется слишком сложной. Естественно, что такая архитектура не требует какого-либо описания, кроме фиксации класса архитектуры, так как отображение внешних функций на эту программу тривиально, а определять способ взаимодействия не требуется в силу отсутствия какого-либо внешнего взаимодействия программы, кроме как взаимодействия ее с пользователем, а последнее описывается в документации по применению ПС.

Комплекс автономно выполняемых программ состоит из набора программ так, что только одна из программ может быть активизирована пользователем, но при выполнении этой программы другие программы из данного набора не могут быть запущены до тех пор, пока не закончит выполнение активизированная программа. Кроме того, все программы этого набора применятся к одной и той же информационной среде.

Следовательно, программы этого набора не взаимодействуют на уровне управления, поскольку взаимодействие между ними осуществляется только через общую информационную среду.

Слоистая программная система состоит из некоторой упорядоченной совокупности программных подсистем, называемых слоями, такой, что:

- на каждом слое ничего не известно ни о свойствах, ни о существовании других, более высоких слоев;
- каждый слой может взаимодействовать с предшествующим, более низким слоем только через заранее определенный интерфейс, ничего не зная о внутреннем строении всех предшествующих слоев;
- каждый слой располагает определенными ресурсами, которые он либо скрывает от других слоев, либо предоставляет последующему слою некоторые их абстракции через указанный интерфейс.

Таким образом, в слоистой программной системе каждый слой может реализовать некоторую абстракцию данных. Связи между слоями ограничены передачей значений параметров обращения каждого слоя к смежному снизу слою и выдачей результатов этого обращения от нижнего слоя верхнему. Использование глобальных данных несколькими слоями невозможно.

В качестве примера использования такой архитектуры рассмотрим некую операционную систему, предложенную Дейкстра. Эта операционная система состоит из четырех слоев. На нулевом слое производится обработка всех прерываний и распределение центрального процессора программам, как процессам. Только этот уровень осведомлен о мультипрограммных аспектах системы. На первом слое осуществляется управление страничной организацией памяти. Всем вышестоящим слоям предоставляется виртуальная непрерывная, но не страничная память. На втором слое осуществляется связь с консолью (пультом управления) оператора. Только этот слой знает технические характеристики консоли. На третьем, последнем слое осуществляется буферизация входных и выходных потоков данных и реализуются так называемые абстрактные каналы ввода и вывода, так что прикладные программы не знают технических характеристик устройств ввода и вывода.

Нетрудно увидеть, как идеи, выдвинутые Дейкстра более 20 лет назад, воплотились в архитектуре операционных систем UNIX и последних версий Windows.

Комплекс параллельно действующих программ представляет собой набор программ, способных взаимодействовать между собой, находясь одновременно в стадии выполнения. Это означает, что такие программы, загружены в оперативную память, активизированы и могут попеременно разделять во времени один или несколько центральных процессоров компьютера. Кроме того, программы могут осуществлять динамическое взаимодей-

ствие между собой, выполняя синхронизацию. Обычно взаимодействие между такими процессами производится путем передачи друг другу некоторых сообщений.

Простейшей разновидностью такой архитектуры является конвейер. Конвейер представляет собой последовательность программ, в которой стандартный вывод каждой программы, кроме самой последней, связан со стандартным вводом следующей программы этой последовательности. Конвейер обрабатывает некоторый поток сообщений. Каждое сообщение этого потока поступает на ввод первой программе, которая передает переработанное сообщение следующей программе, а сама начинает обработку очередного сообщения их входного потока. Таким же образом действует каждая следующая программа конвейера – получив сообщение от предшествующей программы и, обработав его, она передает переработанное сообщение следующей программе и приступает к обработке очередного сообщения. Последняя программа конвейера выводит результат работы всего конвейера в виде результирующего сообщения. Таким образом, в конвейере, состоящим из n программ, может одновременно находиться в обработке до n сообщений. Конечно, в силу того, что разные программы конвейера могут затратить на обработку очередных сообщений разное время, необходимо обеспечить синхронизацию всех процессов.

3.3. Архитектурные функции

Для обеспечения взаимодействия между подсистемами в ряде случаев не требуется создавать какие-либо дополнительные программные компоненты. Для этого может быть достаточно заранее фиксированных соглашений и стандартных возможностей базового программного обеспечения, т.е. операционной системы. Так в комплексе автономно выполняемых программ для обеспечения взаимодействия достаточно спецификации общей внешней информационной среды и возможностей операционной системы для запуска программ. В слоистой программной системе может оказаться достаточным спецификации выделенных программных слоев и обычный аппарат обращения к процедурам.

Однако в ряде случаев для обеспечения взаимодействия между программными подсистемами может потребоваться создание дополнительных программных компонент. Так для управления работой комплекса автономно выполняемых программ часто создают специализированный командный интерпретатор, более удобный в данной предметной области для подготовки требуемой внешней информационной среды и запуска требуемой программы, чем базовый командный интерпретатор используемой операционной системы. В слоистых программных системах может быть создан особый аппарат обращения к процедурам слоя, например, обеспечивающий парал-

лельное выполнение этих процедур. В комплексе параллельно действующих программ для управления портами сообщений требуется специальная программная подсистема. Такие программные компоненты реализуют не внешние функции ПС, а функции, возникшие в результате разработки архитектуры этого ПС. В связи с этим такие функции мы будем называть архитектурными.

4. ТЕСТИРОВАНИЕ И ОТЛАДКА ПРОГРАММНОЙ СИСТЕМЫ

4.1. Основные понятия

Отладка ПС – это комплекс мер, направленных на обнаружение и исправление ошибок в ПС с использованием процессов выполнения его программ.

Тестирование ПС – это процесс выполнения его программ на некотором наборе данных, для которого заранее известен результат применения или известны правила поведения этих программ. Указанный набор данных называется тестовым или просто тестом. Таким образом, отладку можно представить в виде многократного повторения трех процессов: тестирования, в результате которого может быть констатировано наличие в ПС ошибки, поиска места ошибки в программах и документации ПС и редактирования программ и документации с целью устранения обнаруженной ошибки.

4.2. Принципы и виды отладки программной системы

Успех отладки ПС в значительной степени предопределяет рациональная организация тестирования. При отладке ПС отыскиваются и устраняются, в основном, те ошибки, наличие которых в ПС устанавливается при тестировании. Как было уже отмечено ранее, тестирование не может доказать правильность ПС, в лучшем случае оно может продемонстрировать наличие в нем ошибки. Другими словами, нельзя гарантировать, что тестированием ПС практически выполнимым набором тестов можно установить наличие каждой имеющейся в ПС ошибки. Поэтому возникает две задачи.

Первая задача состоит в подготовке такого набора тестов, которые позволят обнаружить в нем по возможности большее число ошибок. Однако чем дольше продолжается процесс тестирования и отладки в целом, тем большей становится стоимость ПС. Отсюда вытекает вторая задача, состоящая в определении момента окончания отладки ПС. Признаком возможности окончания отладки является полнота охвата пропущенными через ПС тестами множества различных ситуаций, возникающих при выполнении программ ПС, и относительно редкое проявление ошибок в ПС на последнем отрезке процесса тестирования. Последнее определяется в соответствии с требуемой степенью надежности ПС, указанной в спецификации его качества.

Для оптимизации набора тестов, т.е. для подготовки такого набора тестов, который позволил бы при заданном их числе обнаруживать наибольшее число ошибок в ПС, необходимо заранее спланировать этот набор и использовать рациональную стратегию планирования тестов. Проектирова-

ние тестов можно начинать сразу же после завершения этапа внешнего описания ПС. Возможны различные подходы к выработке стратегии проектирования тестов, которые можно условно разместить между следующими двумя предельными подходами [2].

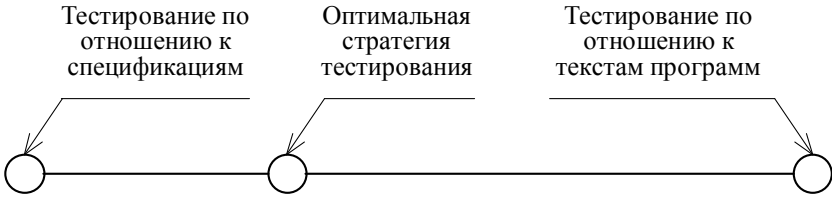


Рис. В. Спектр подходов к проектированию тестов.

Левый крайний подход заключается в том, что тесты проектируются только на основании изучения спецификаций ПС – внешнего описания, описания архитектуры и спецификации программных модулей. Строение модулей при этом никак не учитывается, т.е. они рассматриваются как черные ящики. Фактически такой подход требует полного перебора всех наборов входных данных, так как в противном случае некоторые участки программ ПС могут не работать при пропуске любого теста, а это значит, что содержащиеся в них ошибки не будут проявляться. Однако тестирование ПС полным множеством наборов входных данных практически неосуществимо.

Правый крайний подход заключается в том, что тесты проектируются на основании изучения текстов программ с целью протестировать все пути выполнения каждой программ ПС. Если принять во внимание наличие в программах циклов с переменным числом повторений, то различных путей выполнения программ ПС может оказаться чрезвычайно много, так что их тестирование также будет практически неосуществимо.

Оптимальная стратегия проектирования тестов расположена внутри интервала между этими крайними подходами, но ближе к левому краю. Она включает проектирование значительной части тестов по спецификациям, но требует также выполнения некоторых тестов и по текстам программ. При этом в первом случае эта стратегия базируется на принципах:

- на каждую используемую функцию или возможность требуется хотя бы один тест;
- на каждую область и на каждую границу изменения какой-либо входной величины необходим хотя бы один тест;
- на каждую особую, исключительную ситуацию, указанную в спецификациях, нужен хотя бы один тест.

Во втором случае эта стратегия базируется на принципе – каждая команда каждой программы ПС должна проработать хотя бы на одном тесте.

Кроме того, оптимальную стратегию проектирования тестов можно конкретизировать на основании следующего правила – для каждого программного документа, включая тексты программ, должны проектироваться свои тесты с целью выявления в нем ошибок. Во всяком случае, этот принцип необходимо соблюдать в соответствии с определением ПС и содержанием понятия технологии программирования как технологии разработки надежных ПС.

Обычно различают два основных вида отладки – автономную и комплексную отладку ПС. *Автономная отладка* ПС означает последовательное раздельное тестирование различных частей программ, входящих в ПС, с поиском и исправлением в них фиксируемых при тестировании ошибок. Она фактически включает отладку каждого программного модуля и отладку сопряжения модулей. *Комплексная отладка* означает тестирование ПС в целом с поиском и исправлением фиксируемых при тестировании ошибок во всех документах, включая тексты программ ПС. К таким документам относятся определение требований к ПС, спецификация качества ПС, функциональная спецификация ПС, описание архитектуры ПС и тексты программ ПС.

4.3. Заповеди отладки программной системы

В этом разделе дадим общие рекомендации по организации отладки ПС. Но сначала следует отметить один феномен, который подтверждает важность предупреждения ошибок на предыдущих этапах разработки – по мере роста числа обнаруженных и исправленных ошибок в ПС растет также относительная вероятность существования в нем *необнаруженных ошибок* [2]. Это объясняется тем, что при росте числа ошибок, обнаруженных в ПС, уточняется и наше представление об общем числе допущенных в нем ошибок, а значит, в какой-то мере, и о числе необнаруженных еще ошибок.

Итак, сформулируем рекомендации, которых необходимо придерживаться при организации отладки.

Правило 1. Считайте тестирование ключевой задачей разработки ПС, поручайте его самым квалифицированным и одаренным программистам; нежелательно тестировать свою собственную программу.

Правило 2. Хорошим считается тест, для которого высока вероятность обнаружения ошибки, а не тот, который демонстрирует правильную работу программы.

Правило 3. Готовьте тесты как для правильных, так и для неправильных данных.

Правило 4. Документируйте запуски тестов через компьютер; детально изучайте результаты каждого теста; избегайте уникальных тестов, которые нельзя повторить.

Правило 5. Никогда не упрощайте программу только затем, чтобы облегчить ее тестирование.

Правило 6. Пропускайте заново все тесты, связанные с проверкой работы какой-либо программы ПС, если в нее были внесены изменения, например, в результате устранения ошибки.

4.4. Автономная отладка программной системы

При автономной отладке ПС каждый модуль на самом деле тестируется в некотором программном окружении, кроме случая, когда отлаживаемая программа состоит только из одного модуля. Это окружение состоит из других модулей, часть которых является модулями отлаживаемой программы, которые уже отлажены, а часть – модулями, управляющими отладкой или отладочными модулями. Таким образом, при автономной отладке тестируется всегда некоторая программа, построенная специально для тестирования отлаживаемого модуля. Эта программа лишь частично совпадает с отлаживаемой программой, кроме случая, когда отлаживается последний модуль отлаживаемой программы. В процессе автономной отладки ПС производится наращивание тестируемой программы отлаженными модулями, при переходе к отладке следующего модуля в его программное окружение добавляется последний отлаженный модуль. Такой процесс наращивания программного окружения отлаженными модулями называется *интеграцией программы* [2]. Отладочные модули, входящие в окружение отлаживаемого модуля, зависят от порядка, в каком отлаживаются модули этой программы, от того, какой модуль отлаживается и, возможно, от того, какой тест будет пропускаться.

При восходящем тестировании это окружение будет содержать только один отладочный модуль, который будет головным в тестируемой программе. Такой отладочный модуль называют *ведущим* или драйвером [2]. Ведущий отладочный модуль подготавливает информационную среду для тестирования отлаживаемого модуля, т. е. формирует ее состояние, требуемое для тестирования этого модуля, в частности, путем ввода некоторых тестовых данных, осуществляет обращение к отлаживаемому модулю и после окончания его работы выдает необходимые сообщения. При отладке одного модуля для разных тестов могут составляться разные ведущие отладочные модули.

При нисходящем тестировании окружение отлаживаемого модуля в качестве отладочных модулей содержит отладочные имитаторы или заглушки некоторых еще не отлаженных модулей. К таким модулям относятся, прежде всего, все модули, к которым может обращаться отлаживаемый модуль, а также еще не отлаженные модули, к которым могут обращаться уже отлаженные модули, включенные в это окружение. Некоторые из этих имитаторов при отладке одного модуля могут изменяться для разных тестов.

На практике в окружении отлаживаемого модуля могут содержаться отладочные модули обоих типов, если используется смешанная стратегия тестирования. Это связано с тем, что как восходящее, так и нисходящее тестирование имеет свои достоинства и свои недостатки.

К достоинствам восходящего тестирования относятся: 1) простота подготовки тестов; 2) возможность полной реализации плана тестирования модуля.

Это связано с тем, что тестовое состояние информационной среды готовится непосредственно перед обращением к отлаживаемому модулю.

Недостатками восходящего тестирования являются следующие его особенности: 1) тестовые данные готовятся, как правило, не в той форме, которая рассчитана на пользователя; 2) большой объем отладочного программирования; 3) необходимость специального тестирования сопряжения модулей.

К достоинствам нисходящего тестирования относятся следующие его особенности: 1) большинство тестов готовится в форме, рассчитанной на пользователя; 2) во многих случаях относительно небольшой объем отладочного программирования; 3) отпадает необходимость тестирования сопряжения модулей.

Недостатком нисходящего тестирования является то, что тестовое состояние информационной среды перед обращением к отлаживаемому модулю готовится косвенно, оно является результатом применения уже отлаженных модулей к тестовым данным или данным, выдаваемым имитаторами. Это затрудняет подготовку тестов и требует высокой квалификации разработчика тестов, а также делает затруднительным или даже невозможным реализацию полного плана тестирования отлаживаемого модуля.

Указанный недостаток иногда вынуждает разработчиков применять только восходящее тестирование. Однако чаще применяют некоторые модификации нисходящего тестирования либо некоторую комбинацию нисходящего и восходящего тестирования. Исходя из того, что нисходящее тестирование, в принципе, является предпочтительным, остановимся на приемах, позволяющих в какой-то мере преодолеть указанные трудности.

Прежде всего, необходимо организовать отладку программы таким образом, чтобы как можно раньше были отлажены модули, осуществляющие ввод данных, тогда тестовые данные можно готовить в форме, рассчитанной на пользователя, что существенно упростит подготовку последующих тестов. Далеко не всегда этот ввод осуществляется в головном модуле, поэтому приходится в первую очередь отлаживать цепочки модулей, ведущие к модулям, осуществляющим указанный ввод. Пока модули, осуществляющие ввод данных, не отлажены, тестовые данные поставляются некоторыми

имитаторами – они либо включаются в имитатор как его часть, либо вводятся этим имитатором.

При нисходящем тестировании некоторые состояния информационной среды, при которых требуется тестировать отлаживаемый модуль, могут не возникать при выполнении отлаживаемой программы ни при каких входных данных. В этих случаях можно было бы вообще не тестировать отлаживаемый модуль, так как обнаруживаемые при этом ошибки не будут проявляться при выполнении отлаживаемой программы ни при каких входных данных. Однако так поступать не рекомендуется, так как при изменениях отлаживаемой программы, например, при сопровождении ПС, не использованные для тестирования отлаживаемого модуля состояния информационной среды могут уже возникать, что требует дополнительного тестирования этого модуля. Для осуществления тестирования отлаживаемого модуля в указанных ситуациях иногда используют подходящие имитаторы, чтобы создать требуемое состояние информационной среды. Чаще же пользуются модифицированным вариантом нисходящего тестирования, при котором отлаживаемые модули перед их интеграцией предварительно тестируются отдельно. В этом случае в окружении отлаживаемого модуля появляется ведущий отладочный модуль, наряду с имитаторами модулей, к которым может обращаться отлаживаемый модуль. Однако более целесообразной представляется другая модификация нисходящего тестирования. После завершения нисходящего тестирования отлаживаемого модуля для достижимых тестовых состояний информационной среды следует его отдельно протестировать для остальных требуемых состояний информационной среды.

Часто применяют также комбинацию восходящего и нисходящего тестирования, которую называют методом *сандвича* [2]. Сущность этого метода заключается в одновременном осуществлении как восходящего, так и нисходящего тестирования, пока эти два процесса тестирования не встретятся на каком-либо модуле, где-то в середине структуры отлаживаемой программы. Этот метод при разумном порядке тестирования позволяет воспользоваться достоинствами как восходящего, так и нисходящего тестирования, а также в значительной степени нейтрализовать их недостатки.

Весьма важным при автономной отладке является тестирование сопряжения модулей. Дело в том, что спецификация каждого модуля программы, кроме головного, используется в этой программе в двух ситуациях – при разработке текста этого модуля и при написании обращения к этому модулю в других модулях программы. И в том, и в другом случаях в результате ошибки может быть нарушено требуемое соответствие заданной спецификации модуля. Такие ошибки требуется обнаруживать и устранять. Для этого и предназначено тестирование сопряжения модулей. При нисходящем тестировании тестирование сопряжения осуществляется попутно каждым

пропускаемым тестом, что считают достоинством нисходящего тестирования. При восходящем тестировании обращение к отлаживаемому модулю производится не из модулей отлаживаемой программы, а из ведущего отладочного модуля. В связи с этим существует опасность, что последний модуль может приспособиться к некоторым дефектам отлаживаемого модуля. Поэтому, приступая в процессе интеграции программы к отладке нового модуля, приходится тестировать каждое обращение к ранее отлаженному модулю с целью обнаружения несогласованности этого обращения с телом соответствующего модуля. Таким образом, приходится частично повторять в новых условиях тестирование ранее отлаженного модуля, при этом возникают те же трудности, что и при нисходящем тестировании.

Автономное тестирование модуля целесообразно осуществлять в четыре последовательно выполняемых шага [2].

Шаг 1. На основании спецификации отлаживаемого модуля подготовьте тесты для каждой возможности и каждой ситуации, для каждой границы областей допустимых значений всех входных данных и для каждой области изменения данных, для каждой области недопустимых значений всех входных данных и каждого недопустимого условия.

Шаг 2. Проверьте текст модуля, чтобы убедиться, что каждое направление любого разветвления будет пройдено хотя бы на одном тесте.

Шаг 3. Проверьте текст модуля, чтобы убедиться, что для каждого цикла существуют тесты, обеспечивающие, по крайней мере, три следующие ситуации – тело цикла не выполняется ни разу, тело цикла выполняется один раз и тело цикла выполняется максимальное число раз.

Шаг 4. Проверьте текст модуля, чтобы убедиться, что существуют тесты, проверяющие чувствительность к отдельным особым значениям входных данных.

Обязательно добавьте недостающие тесты, если обнаружилось, что какой-либо из шагов не выполнен.

4.5. Комплексная отладка программной системы

Вполне очевидно, при комплексной отладке тестируется ПС в целом, причем тесты готовятся по каждому из документов программного средства. Тестирование этих документов производится, как правило, в порядке, обратном их разработке. Исключение составляет лишь тестирование документации по применению, которая разрабатывается по внешнему описанию параллельно с разработкой текстов программ – это тестирование лучше производить после завершения тестирования внешнего описания. Тестирование при комплексной отладке представляет собой применение ПС к конкретным данным, которые в принципе могут возникнуть у пользователя, в частности, все тесты готовятся в форме, рассчитанной на пользователя, но,

возможно, в моделируемой, а не в реальной среде. Например, некоторые недоступные при комплексной отладке устройства ввода и вывода могут быть заменены их программными имитаторами.

Тестирование архитектуры ПС. Целью тестирования является поиск несоответствия между описанием архитектуры и совокупностью программ ПС. К моменту начала тестирования архитектуры ПС должна быть уже выполнена автономная отладка каждой подсистемы. Ошибки реализации архитектуры могут быть связаны, прежде всего, с взаимодействием этих подсистем, в частности, с реализацией архитектурных функций. Поэтому хотелось бы проверить все пути взаимодействия между подсистемами ПС. При этом желательно хотя бы протестировать все цепочки выполнения подсистем без повторного вхождения последних. Если заданная архитектура представляет ПС в качестве малой системы из выделенных подсистем, то число таких цепочек будет вполне обозримо.

Тестирование внешних функций. Целью тестирования является поиск расхождений между функциональной спецификацией и совокупностью программ ПС. Несмотря на то, что все эти программы автономно уже отлажены, указанные расхождения могут быть, например, из-за несоответствия внутренних спецификаций программ и их модулей функциональной спецификации ПС в целом. Как правило, тестирование внешних функций производится так же, как и тестирование модулей на первом шаге, т.е. как черного ящика.

Тестирование качества ПС. Целью тестирования является поиск нарушений требований качества, сформулированных в спецификации качества ПС. Это наиболее трудный и наименее изученный вид тестирования. Ясно лишь, что далеко не каждый примитив качества ПС может быть испытан тестированием. Завершенность ПС проверяется уже при тестировании внешних функций. На данном этапе тестирование этого примитива качества может быть продолжено, если требуется получить какую-либо вероятностную оценку степени надежности ПС. Однако методика такого тестирования еще требует своей разработки. Могут тестироваться такие примитивы качества, как точность, устойчивость, защищенность, временная эффективность, в какой-то мере – эффективность по памяти, эффективность по устройствам, расширяемость и, частично, независимость от устройств. Каждый из этих видов тестирования имеет свою специфику и заслуживает отдельного рассмотрения.

Тестирование документации по применению ПС. Целью тестирования является поиск несогласованности документации по применению и совокупностью программ ПС, а также выявление неудобств, возникающих при применении ПС. Этот этап непосредственно предшествует подключению пользователя к завершению разработки ПС, поэтому весьма важно разра-

ботчикам сначала самим воспользоваться ПС так, как это будет делать пользователь [2]. Все тесты на этом этапе готовятся исключительно на основании только документации по применению ПС. Прежде всего, должны тестироваться возможности ПС как это делалось при тестировании внешних функций, но только на основании документации по применению. Должны быть протестированы все неясные места в документации, а также все примеры, использованные в документации. Далее тестируются наиболее трудные случаи применения ПС с целью обнаружить нарушение требований относительности легкости применения ПС.

Тестирование определения требований к ПС. Целью тестирования является выяснение, в какой мере ПС не соответствует предъявленному определению требований к нему. Особенность этого вида тестирования заключается в том, что его осуществляет заказчик ПС с целью преодоления барьера между разработчиком и пользователем [2]. Обычно это тестирование производится с помощью контрольных задач, для которых известен результат решения. В тех случаях, когда разрабатываемое ПС должно придти на смену другой версии ПС, которая решает хотя бы часть задач разрабатываемого ПС, тестирование производится путем решения общих задач с помощью как старого, так и нового ПС с последующим сопоставлением полученных результатов. Иногда в качестве формы такого тестирования используют опытную эксплуатацию ПС, т.е. ограниченное применение нового ПС с анализом использования результатов в практической деятельности.

Часть 2. Windows программирование на основе API

5. Основные характеристики Windows платформ

В этом разделе даются основные понятия о семействе операционных систем Windows, а также детально рассматривается механизм взаимодействия приложений Windows с ядром системы. Раздел является ключевым для понимания последующего материала.

Операционные системы Windows – многозадачные, многопользовательские операционные системы, обеспечивающие полное использование ресурсов PC, при этом достигая абсолютной независимости исполняемых приложений (программ) от особенностей аппаратуры компьютера. Операционные системы Windows имеют единый графический интерфейс, поддерживают обмен данными между приложениями (Clipboard, OLE, DDE), включают сетевое обеспечение и многое другое. Можно сказать, что по мере проникновения на рынок персональных компьютеров, системы Windows почти полностью вытеснили всех имеющихся конкурентов и стали, фактически, эталоном операционной системы для PC.

5.1. Краткая история Windows

О работе над Windows корпорация Microsoft заявила в 1983 году и реализовала ее двумя годами позже, в виде Windows 1.0. В течение двух последующих лет, Windows версии 1.1 претерпела несколько модернизаций, необходимых для удовлетворения требований международного рынка.

Windows 2.0 была создана в 1987 году. Эта версия содержала несколько изменений пользовательского интерфейса. Наиболее важное из этих изменений касалось использования перекрывающихся окон, вместо окон, расположенных рядом, что было характерно для Windows 1.x. Windows 2.0 содержала также улучшенный интерфейс клавиатуры и манипулятора мышь, а также, отчасти, окон меню и диалога.

В то время для Windows требовались только процессоры Intel 8086 или 8088, работающие в реальном режиме, при этом доступ осуществлялся к 1 мегабайту оперативной памяти. Windows-386, созданная вскоре после Windows 2.0, использовала виртуальный режим процессора Intel 80386 для запуска нескольких одновременно работающих программ MS-DOS в окнах. Для симметрии Windows версии 2.1 назвали Windows-286.

Windows 3.0 появилась в марте 1992 года. Здесь были объединены ранние версии Windows-286 и Windows-386. Главным изменением в Windows 3.0 была поддержка защищенного режима процессоров Intel 80286, 80386 и 80486. Это позволило Windows и ее приложениям получить доступ к 16 мегабайтам оперативной памяти. “Оболочка” программ

Windows для запуска программ и поддержки файлов была полностью переделана. Windows 3.0 – это первая версия Windows, которая стала “родной” для множества пользовательских машин в домах и офисах.

В апреле 1992 года появилась Windows 3.1. Сюда были включены такие важные свойства, как технология TrueTupe для шрифтов, что дало возможность масштабировать шрифты для Windows; MultiMedia – звук и музыка; OLE и диалоговые окна общего пользования. Кроме этого, Windows 3.1 работала только в защищенном режиме и требовала наличия процессора 80286 или 80386 и, по крайней мере, одного мегабайта оперативной памяти.

Любая история Windows была бы неполной без упоминания об операционной системе OS/2 – альтернативной для DOS и Windows, которая на первом этапе развивалась корпорацией Microsoft в сотрудничестве с фирмой IBM. Операционная система OS/2 версии 1.0 (только для символьного режима) работала на процессорах Intel 80286 и более поздних и появилась в конце 1987 года. Графическая оболочка Presentation Manager была реализована в OS/2 версии 1.1 в 1988 году. Presentation Manager, как изначально предполагалось, должна была стать основной стратегией развития операционных систем. Хотя у OS/2 все еще имеется немало горячих поклонников, ее популярность не идет ни в какое сравнение с популярностью Windows.

Windows NT, появившаяся в 1993 году, стала первой версией Windows, поддерживающей 32-х разрядную модель программирования для процессоров Intel 80386, 80486 и Pentium. Windows NT имеет сплошное плоское (flat) 32-х разрядное адресное пространство и 32-х разрядные целые. Кроме этого, система Windows NT является переносимой на различные компьютерные платформы.

Windows 95 (или Chicago, или Windows 4.0) появилась в августе 1995 года. Также как Windows NT, Windows 95 поддерживает 32-х разрядную модель программирования. Хотя у Windows 95 и нет некоторых возможностей Windows NT, таких как, например, высокая степень безопасности, тем не менее она способна работать на компьютерах, имеющих всего 4 мегабайта оперативной памяти.

В Windows 98 (Windows 4.1) внесено ряд улучшений по сравнению с Windows 95, например, касающихся поддержки жестких дисков большой емкости, а также изменен пользовательский интерфейс, который теперь базируется на Internet Explorer, который стал компонентом операционной системы.

Windows 2000 (Windows NT 5.0) вышла в конце 1999 года и является последней, к настоящему времени, представительницей семейства операционных систем Windows. По заявлениям фирмы Microsoft, эта система вобрала в себя все лучшее, что было создано до нее, и должна заменить Windows NT. Однако

колоссальные затраты ресурсов PC и незавершенность разработки пока препятствуют ее широкому распространению.

5.2. Отличия и общие свойства Windows платформ

5.2.1. Общие свойства Windows платформ

Для разработчиков Windows приложений фирма Microsoft предоставляет специальный интерфейс программирования приложений под названием **Win32 API**. Кратко можно сказать, что API – это совокупность системных функций, к которым может обращаться приложение.

В настоящее время Win32 API реализован для четырех платформ. Назовем их в порядке создания: Win32s, Windows NT, Windows 95/98 и Windows CE. Фирма Microsoft стремилась включить все Win32 функции во все платформы, поддерживающие интерфейс Win32 API. Для разработчиков программного обеспечения это означает, что тексты программ не придется переписывать для каждой платформы заново, достаточно лишь перекомпилировать приложение под другую платформу. Однако не все функции реализованы в каждой из перечисленных платформ, для таких функций используются так называемые “заглушки”. Это означает, что ваше приложение будет работать по-разному для каждой из этих платформ.

5.2.2. Отличия Win32 платформ

Рассмотрим отличия Win32 платформ между собой и определим роль каждой платформы в стратегии операционных систем фирмы Microsoft.

Win32s была первой платформой, на которой могли выполняться 32-х разрядные приложения. Платформа Win32s состоит из набора *DLL* (dynamic link library – динамически подключаемых библиотек) и драйвера виртуального устройства, дополняющего 16-ти разрядную систему Windows 3.1 интерфейсом Win32 API. Таким образом, Win32s не более чем 32-х разрядная надстройка над обыкновенной 16-ти разрядной системой Windows 3.1. Этот надстройка преобразует 32-х битные параметры функций в 16-ти битные и вызывает соответствующие функции 16-ти разрядной Windows.

В Win32s большинство Win32 функций реализовано в виде заглушек – при их вызове тут же происходит возврат и ничего более. Например, поскольку 16-ти разрядная Windows не поддерживает потоков исполнения, функция **`CreateThread()`**, создающая поток, возвратит нулевой описатель. Однако платформа Win32s все-таки расширяет некоторые возможности операционной системы – в ней, например, частично реализована поддержка проецируемых в память файлов.

Платформа Win32s создавалась для того, чтобы разработчики приступили к разработке 32-х разрядных приложений еще до появления настоящих

Win32 платформ. Фирма Microsoft рассчитывала пробудить интерес к программированию в Win32, чтобы к моменту выпуска Windows NT на рынке уже присутствовали 32-х разрядные приложения.

Windows NT отказоустойчивая операционная система, и даже некорректные приложения, как правило, не способны привести ее к краху. Под некорректными приложениями в данном случае имеются в виду программы, напрямую обращающиеся к оборудованию компьютера, например, к портам ввода/вывода, а именно так и ведут себя многие программы MS-DOS и 16-ти разрядной Windows. При запуске такой программы в Windows NT, система немедленно завершит ее, как только та попытается напрямую обратиться к оборудованию. Windows NT не отягощенная наследством MS-DOS и поэтому считается, что будущее операционных систем за этой архитектурой (например, Windows 2000 является прямой наследницей Windows NT), несмотря на то, что Windows NT требует значительных объемов оперативной памяти и жесткого диска.

Для Windows NT 32-х разрядные приложения являются “родными”, они могут использовать все преимущества системы – мощь, отказоустойчивость, быстродействие. Более того, система Windows NT способна выполнять сразу несколько разнотипных приложений, созданных для таких операционных систем, как OS/2 1.x, MS-DOS и 16-ти разрядной Windows.

Операционная система Windows NT способна работать на машинах с разными типами процессоров. Для того чтобы Windows NT работала на MIPS R4000, DEC Alpha или Motorola PowerPC, Microsoft просто перекомпилирует исходный код операционной системы (большая часть Windows NT написана на C/C++) с помощью компилятора, “родного” для данного процессора. После того, как Microsoft переносит Windows NT на новую процессорную платформу, разработчику программного обеспечения необходимо лишь перекомпилировать свое Win32 приложение, и оно тоже начинает работать на машине с другой архитектурой. Естественно, перенос операционной системы на процессор с другой архитектурой не так прост. Для этого приходится переписывать два низкоуровневых компонента Windows NT Executive: ядро и HAL (Hardware Abstraction Layer – слой абстрагирования оборудования). Эти компоненты пишутся на соответствующей версии языка ассемблера и весьма специфичны для конкретной архитектуры того или иного процессора.

Важнейшей особенностью Windows NT является поддержка многопроцессорных машин. Работая на машине, например, с четырьмя процессорами, Windows NT позволяет одновременно выполнять до четырех потоков. А это значит, что компьютер получает возможность выполнить четыре одинаковых задачи за время, необходимое для выполнения одной задачи на одном компьютере.

Windows 95 является Win32 платформой, пришедшей на замену 16-ти разрядной Windows 3.1. В Windows 95 интерфейс Win32 API реализован полнее, чем в ее предшественнице Win32s, но все-таки не в таком полном объеме, как в Windows NT. Windows 95 была выпущена, во-первых, только потому, что требования Windows NT к оборудованию были – на то время – чрезмерно высоки, а во-вторых, Windows NT отпугивала потенциальных пользователей сложностью тонкой настройки и администрирования.

Чтобы Windows 95 смогла работать на машинах с 8 Мб памяти, Microsoft пришлось урезать некоторые возможности Win32 API. В итоге, Windows 95 не полностью поддерживает целый ряд Win32 функций, связанных, например, с асинхронным вводом/выводом файлов, отладкой, реестром, защитой, регистрацией событий, удаленным запуском процессов и т.д. – функции существуют, но реализованы частично. Однако фирме Microsoft удалось реализовать в Windows 95 значительную часть возможностей Win32 API и тем самым сделать ее весьма мощной операционной системой, которой, к настоящему времени, оборудовано большинство IBM совместимых компьютеров.

Windows CE новейшая Win32 платформа от фирмы Microsoft. Она создана в связи с появлением новой техники и рассчитана, главным образом, на карманные вычислительные устройства, которые работают от аккумуляторов и не имеют ни жестких, ни гибких дисков. У большинства таких машин 2 или 4 Мб памяти, а отсутствие жеского диска делает невозможным использование виртуальной памяти. Из-за столь жестких ограничений Microsoft пришлось создать совершенно новую операционную систему, менее требовательную к памяти, чем Windows 95.

Платформа Windows CE поддерживает проецируемые в память файлы, структурную обработку исключений, динамически подключаемые библиотеки, реестр, исполнение потоков с вытеснением и многое другое. Адресное пространство каждого процесса в ней защищено от других выполняемых процессов. Кроме того, Windows CE – переносимая операционная система, способная работать на процессорах MIPS, SH3 и x86.

Таким образом, все Win32 платформы содержат полный набор Win32 функций, вне зависимости от конкретной платформы. Но реализация реализации рознь. Когда Microsoft заявляет, что все Win32 функции будут реализованы на каждой платформе, то на деле это означает, что все Win32 функции *будут существовать* на каждой платформе, но *поддерживаться* на некоторых из них они будут только частично.

Несмотря на очевидные различия между Win32 платформами, их объединяют более существенные черты. В частности, предполагается, что большинство *простых* приложений совместимы с Windows NT и Windows 95 с незначительными изменениями или вовсе без них.

Рассмотрим теперь основные общие свойства – с точки зрения разработчика, – характерные для всех Windows платформ.

5.2.3. Окна и сообщения

Система Windows является операционной системой передачи сообщений. В основе системы лежит механизм, транслирующий практически каждое событие – нажатие клавиши, перемещение мыши, истечение времени таймера – в соответствующее сообщение. Типичное для Windows приложение построено на основе цикла обработки сообщений, который принимает эти сообщения и отправляет их к соответствующим функциям – обработчикам сообщений. Хотя сообщения передаются приложениям, они адресованы не им, а другим основополагающим компонентам операционной системы – окнам.

С точки зрения пользователя, окно – это прямоугольная область экрана, которую занимает каждая программа Windows. Хотя одна программа может создать несколько окон, всегда имеется одно окно верхнего уровня, которое называется “главным окном” приложения. С точки зрения разработчика, окно – это самостоятельно существующий объект, параметры которого описаны в специальных структурах данных, а поведение – функцией окна.

Итак, окно – это гораздо больше, чем просто прямоугольная область на экране компьютера; оно представляет некую абстрактную сущность, через которую взаимодействуют пользователь и компьютер.

5.2.4. Приложения, потоки и окна

Рассмотрим связь между приложениями и окнами. Типичное Win32 приложение состоит из одного или более потоков, которые выполняются параллельно. Использование потоков можно понимать как многозадачность в рамках отдельного приложения, например, один поток обрабатывает данные, вводимые пользователем, в то время как другой поток занят передачей документа на принтер.

Окно всегда принадлежит какому-то потоку; поток может владеть одним или несколькими окнами или вообще ни одним. Наконец, окна находятся в иерархической зависимости: некоторые из них являются окнами верхнего уровня, а другие – подчинены своим родительским окнам.

В Windows существует несколько типов окон. Самыми очевидными типами окон являются большие прямоугольные окна, связанные с приложениями, которые можно переместить, изменить их размеры, максимизировать или минимизировать. Менее очевидно, что многие элементы, отображаемые внутри главного или диалогового окна, сами по себе являются ок-

нами. Каждая кнопка, поле ввода, полоса прокрутки, окно списка, пиктограмма, даже фон экрана воспринимается операционной системой как окно.

5.2.5. Классы окон

Основное поведение окна определяется классом окон. Класс окна (не путать с понятием "класс" языка программирования C++) несет информацию о начальном внешнем виде окна, пиктограмме по умолчанию, курсоре и ресурсе меню, связанном с окном; и, что более важно – об адресе функции, называемой оконной функцией. Когда приложение обрабатывает сообщения, оно обычно делает это посредством вызова специальной Win32 API функции для каждого принятого сообщения. Эта функция, в свою очередь, вызывает соответствующую оконную функцию, проверяя, для какого класса окна предназначено сообщение.

Заметим, что существует множество стандартных классов окон, предусмотренных самой операционной системой Windows. Эти системные глобальные классы реализуют функциональные возможности общих элементов управления. Любое приложение может использовать эти классы в своих окнах, например, реализовать элемент управления – поле ввода, используя класс окна **`Edit`**.

Приложение может определять собственные классы окон с помощью функции **`RegisterClass()`**. Эта функция позволяет разработчику реализовать поведение окна, которое не является частью ни одного из поддерживаемых системой глобальных классов.

> Именно так должно поступать Windows приложение для того, чтобы реализовать функциональные возможности своего главного окна.

Windows также позволяет разбивать существующие окна на субклассы и суперклассы. Разбиение на субклассы замещает оконную процедуру для класса окна на другую процедуру. Это разбиение осуществляется путем изменения адреса процедуры через функцию API **`SetWindowLong()`**, в случае простого разбиения на подклассы или **`SetClassLong()`**, в случае глобального разбиения на подклассы. При этом в первом случае изменится только поведение определенного окна, а во втором – поведение всех окон указанного класса, создаваемых данным приложением.

Операция создания суперкласса создает новый класс на основе существующего, сохраняя его оконную процедуру. Чтобы создать суперкласс из класса окна, приложение извлекает информацию о классе с помощью функции **`GetClassInfo()`**, изменяет полученную таким образом структуру **`WNDCLASS`** и использует измененную структуру при вызове **`RegisterClass()`**. Через функцию **`GetClassInfo()`** приложение также получает адрес оригинальной оконной процедуры, который остается тот

же. А это значит, что сообщения, которые не обработала новая оконная функция, должны быть переданы оригинальной процедуре.

Несмотря на то, что используемая выше терминология напоминает терминологию объектно-ориентированного программирования, понятие класса окна не следует путать с понятиями C++. Понятие класса окна возникло на несколько лет раньше начала использования в Windows объектно-ориентированных языков.

5.2.6. Венгерская нотация

До настоящего времени указывались только имена API функций, без их параметров. Далее нам нужно будет ознакомиться с некоторыми структурами данных, и здесь возникает одна трудность для понимания – текст Windows приложения выглядит необычно для человека, знакомого с языком C, но не владеющего Windows программированием. В частности, не видно знакомых типов данных, а многосимвольные имена объектов более напоминают Pascal программу. Дело в том, что, действительно, Pascal был выбран стартовым языком для разработки Windows, в следствие его жесткой типитизированности данных. Но ограниченность языка Pascal заставила разработчиков перейти на использование более мощного языка C, однако влияние Pascal, в качестве развернутой мнемоники имен переменных, на стиль Windows программирования осталось.

В конце 80-х годов программист фирмы Microsoft Чарльз Симонаи пошел далее, включив тип переменной в ее имя. Было несколько странно встретить примерно такую строку: **`WORD wBufferSize`**. Однако в этом случае имя переменной говорит и о ее предназначении, и о ее типе. Следовательно, становится возможным обнаружить алгоритмические ошибки еще до компиляции программы. Такая техника завоевала популярность в среде Windows разработчиков и в честь ее автора – венгра по национальности – была названа венгерской нотацией.

В таблице приведены некоторые типы данных и префиксы, используемые в Windows программировании.

Тип данных	Истинный тип	Префикс	Комментарий
`BOOL`	`int`	`f`	`Boolean TRUE or FALSE`
`BYTE`	`unsigned char`	`b`	Тип данных Byte
`LPBYTE`	`BYTE FAR*`	`lpb`	Дальний указатель на Byte
`CHAR`	`char`	`ch`	Windows символ
`DWORD`	`unsigned long`	`dw`	Тип данных двойное слово

LPDWORD	DWORD FAR*	lpdw	Дальний указатель на двойное слово
HANDLE	void*	h	Дескриптор объекта
LPHANDLE	void**	lph	Дальний указатель на дескриптор объекта
HBITMAP	void*	hbm	Дескриптор BitMap
HBRUSH	void*	hbr	Дескриптор кисти
HDC	void*	hdc	Дескриптор контекста устройства
HFILE	int	hf	Дескриптор файла
HGDIOBJ	void*	hgdiobj	Дескриптор GDI объекта
HINSTANCE	void*	hinst	Дескриптор экземпляра приложения
HWND	void*	hwnd	Дескриптор окна
INT	int	n	Тип данных int
LPINT	int FAR*	lpn	Дальний указатель на INT
LONG	long	l	32-х разрядное значение
LPLONG	long FAR*	lpl	Дальний указатель на LONG
LPARAM	LONG	lParam	Параметр сообщения типа LONG
WPARAM	unsigned int	wParam	Параметр сообщения типа UINT
LPCSTR	const char*	lpsz, sz	Константная строка в стиле языка C
LPSTR	char*	lpsz, sz	Строка языка C
LRESULT	LONG	lResult	Возвращаемое значение типа LONG
TCHAR	-	ch	Unicode или Windows символ
UINT	unsigned int	u	Беззнаковое целое
ULONG	unsigned long	ul	Беззнаковое 32-х разрядное значение
USHORT	unsigned short	w	Беззнаковое 16-ти разрядное значение
VOID	void	v	Объект без типа
LPVOID	void*	lpv	Дальний указатель на VOID

WORD	unsigned short	w	Тип данных “слово”
LPWORD	unsigned short FAR*	lpw	Дальний указатель на WORD

Взглянув на таблицу, можно увидеть, что для Win32 типы данных **LONG** и **INT** эквивалентны и, конечно же, вместо них можно использовать обычный C тип данных **int**. Но как будут обстоять дела в Win64, которая уже “не за горами”? Использование Windows типов предпочтительнее, поскольку это потребует только перекомпилирования приложения, а не переписывания его, при переходе на другую платформу.

5.2.7. Типы сообщений

Существует много разновидностей сообщений, представляющих Windows события на разных уровнях. Каждое простое событие, каждое простое действие посылается в виде сообщения окну для обработки. Приложению приходит множество сообщений, однако приложение не должно заботиться о смысле каждого отдельного сообщения. Вместо обработки всех возможных сообщений приложение имеет свободу выбора, так как необработанные сообщения передаются назад в Windows для обработки по умолчанию самой операционной системой.

Все Windows сообщение состоят из нескольких частей и представляются структурой **MSG**:

```
typedef struct tagMSG {
HWND    hwnd;       // дескриптор окна
UINT    message;    // значение самого сообщения
WPARAM  wParam;     // параметр сообщения
LPARAM  lParam;     // параметр сообщения
DWORD   time;       // время наступления события
POINT   pt;         // точка, где произошло событие
} MSG;
```

Элемент структуры **hwnd** однозначно идентифицирует окно, которому посылается это сообщение. Каждое окно в Windows имеет уникальный идентификатор.

Элемент **message** идентифицирует само сообщение. Этот элемент принимает любое значение из допустимого диапазона, который описывает Windows сообщения. Для Windows сообщений обычно используются символьные представления, такие как **WM_PAINT**, **WM_TIMER**, а не конкретные числовые значения. Эти символьные значения определены в стандартных файлах заголовков Windows. Обычно приложению необходимо включить в свой исходный текст только файл **windows.h**, он содержит директивы **#include** для всех остальных файлов.

Сообщения можно разделить на несколько групп в зависимости от их смысловой нагрузки. Самой насыщенной группой сообщений является группа *сообщений управления окнами*. Символьные идентификаторы для этих сообщений начинаются с **WM_.** Эта группа настолько велика, что ее обычно разбивают на категории. Эти категории включают:

- сообщения DDE (dynamic data exchange);
- сообщения буфера обмена;
- сообщения мыши;
- сообщения клавиатуры;
- сообщения неклиентской (non-client) области окна;
- сообщения MDI (multiple-document interface);

и некоторые другие.

Перечисленные категории несколько неточны и не всегда строго определены. Они просто служат для удобства программистов, чтобы можно было представить картину множества событий управления окнами. Множество сообщений **WM_** также не фиксировано, оно растет по мере добавления новых возможностей операционной системы.

Другие группы сообщений связаны с определенными типами окон. Существуют сообщения, определенные для полей ввода, кнопок, списков, комбинированных списков, полос прокрутки, элементов просмотра списка деревьев и так далее.

Приложения могут определять *собственные сообщения*, что можно сделать двумя способами. Первый способ пригоден только для случая, если сообщение пересылается между окнами одного и того же приложения. Для этого необходимо определить символическое имя нового сообщения при помощи директивы **#define**, например:

```
#define      UserMessage1        (WM_USER+100)
#define      UserMessage2        (WM_USER+101)
```

Вторым способом определения собственного сообщения является использование функции **RegisterWindowMessage()**, которая возвращает уникальный идентификатор для сообщения. Использование собственных типов сообщений, полученных таким способом, позволяет частям приложения связываться между собой. Более того, разные приложения могут обмениваться информацией посредством такого идентификатора сообщения.

> В Win16 взаимодействующие приложения могли обмениваться данными, пересылая друг другу дескрипторы глобальных блоков памяти. В Win32 этот механизм *не работает*, так как все приложения имеют собственные адресные пространства, которые не перекрываются. Для взаимодействия приложений в Win32 доступны более мощные механизмы, например, отображаемые в память файлы.

5.2.8. Сообщения и многозадачность

В Win16 цикл обработки сообщений, который принимает сообщения и перенаправляет их к соответствующим оконным функциям, играл и другую важную роль во взаимодействии приложения и операционной среды. А именно: он позволял приложению возвращать управление системе. Поскольку Win16 не является операционной системой с вытесняющей многозадачностью, она не "забирает" управление процессором от приложений и ждет, пока приложение само не освободит процессор. Следовательно, в Win16 возможен захват процессора неправильно работающим приложением, что вызывает "зависание" всей системы.

Для того чтобы обеспечить более равномерное разделение вычислительных ресурсов в Win16, функция **`GetMessage()`** автоматически передает управление системе при отсутствии сообщений в очереди приложения. Значит, если приложение в данный момент не задействовано, операционная система передает управление другому приложению. Однако, если приложение, например, производит большую вычислительную работу, другие задачи в Win16 не могут работать. Итак, корректная работа системы Win16 зависит от поведения приложений, а именно, часто ли они вызывают функции обработки сообщений.

> В Win32 реализована вытесняющая многозадачность, когда сама операционная система делает переключение процессов, вне зависимости от текущего состояния исполняемого приложения. Казалось бы, приложения Win32 больше не нуждаются в своевременном возврате управления. Однако это не так, поскольку во время большой вычислительной работы ваше приложение выглядит "зависшим", и пользователь не может им управлять.

5.2.9. Очереди сообщений

В Win16 операционная система поддерживает *единственную очередь сообщений*. Сообщения, генерируемые различными событиями операционной системы помещаются в общую очередь сообщений. При этом возможна ситуация, когда одно приложение пытается извлечь из очереди "свое" сообщение, а операционная система может выполнить *переключение контекста* и активизировать другое приложение, которого ожидают сообщения в очереди. В результате первому приложению не удается выбрать сообщение, и это "подвешивает" систему. Сообщения начинают накапливаться в очереди сообщений. Так как она имела фиксированный размер, в конце концов очередь переполняется. При поступлении каждого нового сообщения, которое не может быть помещено в очередь, Windows отвечала звуковым сигналом.

В Win32 механизм очереди сообщений более совершенен. В этих операционных системах с вытесняющей многозадачностью больше не гарантируется упорядоченное совместное выполнение конкурирующих задач или потоков. Два или более потока могут одновременно попытаться получить доступ к очереди сообщений. Кроме того, поскольку переключение задач больше не зависит от следующего доступного сообщения в очереди, нет гарантии, что задача будет получать только сообщения, адресованные ей.

Это одна из причин, по которой единая очередь сообщений в 32-х разрядных Windows была разделена на *индивидуальные очереди* для каждого потока.

5.2.10. Процессы и потоки

В однопоточных операционных системах наименьшей единицей выполнения является *задача* или *процесс*. Механизм распределения задач в этих операционных системах заключается в переключении между этими задачами.

В противоположность этому в многопоточной системе наименьшей единицей выполнения является *поток*, а не процесс. Задача или процесс могут состоять из более чем одного потока, при этом один из них определяется как главный поток. Организация нового потока требует меньше системных ресурсов, поскольку потоки одного процесса имеют доступ к одному адресному пространству, а переключение между потоками одного и того же процесса требует незначительных расходов ресурсов системы.

5.2.11. Потоки и сообщения

Как было уже сказано, *окно обязательно принадлежит какому-либо потоку*. В этом случае поток должен иметь свою очередь сообщений, в которую операционная система и будет помещать сообщения, адресованные окну данного потока. Однако это не означает, что поток *всегда должен* иметь окно и содержать цикл сообщений. Потоки могут не владеть окном и, следовательно, не обрабатывать сообщений.

В Win32 можно установить отдельный поток, выполняющий, например, длительные операции, в то время как главный поток приложения, владеющий окном, будет обрабатывать все сообщения, принимаемые приложением.

Поток называется *рабочим потоком*, если он не имеет ни окон, ни очереди сообщений, ни цикла обработки сообщений. Обычно такие потоки предназначены для выполнения интенсивных вычислений. Поток, владеющий хотя бы одним окном и, следовательно, предназначенный для обработки сообщений, именуется *потоком пользовательского интерфейса*.

5.2.12. Оконная функция – функция обратного вызова

Чтобы понять, что такое *функция обратного вызова*, вспомним, что в любой операционной системе вызов большинства функций предназначен для запроса некоторого сервиса у операционной системы. Например, для открытия файла используется вызов функции **`fopen()`**. Другими словами, вашей программе нужен какой-либо сервис, и он запрашивается у операционной системы вызовом соответствующей функции, код которой располагается в одной из системных библиотек. Это – обычные функции, и большинство Win32 API функций принадлежит к данному классу.

Однако операционная система Windows включает функции, поведение которых определено с точностью наоборот. Ваше приложение должно содержать тело оконной функции, которую ваша программа *никогда не вызывает*. Обращаться к этой функции может только ядро Windows. Такие функции именуются функциями обратного вызова.

> Оконная функция (именуется также оконной процедурой) связанна с классом окна, который приложение регистрирует при своем запуске. С этого момента ядро Windows “знает” адрес оконной функции и может ее вызывать, использую данный адрес. При этом все вызовы функции окна имеют форму сообщений.

5.2.13. Синхронные и асинхронные сообщения

Одни и те же Windows сообщения могут быть как *синхронными* (queued), так и *асинхронными* (nonqueued). Асинхронными сообщениями называются сообщения, помещаемые Windows в очередь сообщений приложения, и которые затем извлекаются и диспетчеризируются в цикле обработки сообщений. Синхронные сообщения передаются непосредственно окну, когда Windows вызывает оконную процедуру. В результате оконная процедура получает все предназначенные для окна сообщения, как синхронные, так и асинхронные.

> Асинхронные сообщения помещаются в очередь сообщений в одно время, а обрабатываться могут в другое. Синхронные – посылаются напрямую в оконную процедуру и тут же обрабатываются.

Асинхронными становятся сообщения, в основном, тогда, когда они являются результатом пользовательского ввода, путем нажатия клавиатурных клавиш, например, **`WM_KEYDOWN`** и **`WM_KEYUP`**; как результат движения мыши или щелчков кнопок мыши. Также к асинхронным сообщениям относятся сообщения таймера **`WM_TIMER`** или сообщение завершения цикла обработки сообщений **`WM_QUIT`**. Порой асинхронные сообщения являются результатом синхронных сообщений, и наоборот.

Очевидно, что процесс этот сложен, но большая часть сложностей ложится на саму Windows, а не на пользовательские программы. Дело в том, что ваша оконная процедура может что-либо сделать с поступающими сообщениями, а может и проигнорировать их, передавая назад Windows на обработку по умолчанию.

Некоторые сообщения являются результатом вызова определенных функций Win32 API. Например, когда ваше приложение создает окно, вызывает функцию **`CreateWindowEx()`**, Windows отправляет оконной процедуре синхронное сообщение **`WM_CREATE`**. Когда же вы вызываете функцию **`UpdateWindow()`**, Windows помещает в очередь приложения асинхронное сообщение **`WM_PAINT`**.

> Во время обработки оконной процедурой одного сообщения, приложение не может быть прервано другим сообщением. Только в единственном случае, если оконная процедура сама вызывает функцию, которая становится источником нового синхронного сообщения, то оконная процедура начнет обрабатывать это новое сообщение еще до того, как она вернет управление Windows. Таким образом, оконная процедура должна быть *реентерабельной* (reentrant), т.е. повторно-входимой.

Часто возникает необходимость того, чтобы оконная функция сохраняла информацию, полученную в одном сообщении, и использовала ее при обработке другого сообщения. Такую информацию следует описывать в оконной процедуре в виде статических переменных.

5.2.14. Функции Windows

Win32 API содержит большое количество системных функций (более 600) для выполнения разнообразных задач, включая управление потоками, работу с окнами, обработку файлов, управление памятью, графический сервис, коммуникацию и многое другое.

Основное множество системных вызовов Windows можно разделить на три главных категории – функции GDI, модуль USER и модуль KERNEL.

Функции GDI (Graphics Device Interface – интерфейс графических устройств) используются для выполнения основных, аппаратно-независимых графических операций над контекстом устройства. *Контекст устройства* – это интерфейс для определенного графического устройства. Его можно использовать для получения информации об устройстве и выполнения графического вывода на это устройство. Информация, которая может быть получена через контекст устройства, содержит его подробное описание. Технология устройства (векторная или растровая), его тип, имя, разрешающая способность, цветовые возможности, возможности шрифтов – все это можно получить, вызвав контекст соответствующего устройства.

Графический вывод осуществляется через контекст устройства, путем передачи его дескриптора в соответствующую GDI функцию. Через контекст устройства общие, независимые от устройства графические вызовы транслируются в инструкции, представляющие вывод на конкретное устройство. Когда приложение вызывает функцию отрисовки GDI, контекст устройства определяет, какой драйвер устройства выполнит этот вызов. Драйвер устройства в свою очередь может далее передать этот вызов аппаратному ускорителю, если видеоподсистема имеет такие возможности.

Контекст устройств GDI может описывать несколько устройств:

1. *контекст устройств отображения* служит для графического вывода на экран компьютера;
2. *контекст памяти* предназначен для вывода растрового изображения, хранящегося в памяти;
3. *контекст принтера* транслирует графический вывод в управляющие коды принтера;
4. *контекст метафайла* – это специальный вид контекста устройства Windows, позволяющий приложению создавать постоянные записи вызовов GDI функций.

Записи метафайла не зависят от конкретного устройства и позже могут быть воспроизведены на любом устройстве. Метафайлы это не просто удобство, метафайлы играют решающую роль в независимом от устройства представлении внедренных OLE-объектов. Метафайлы предоставляют механизм, который делает OLE-объекты переносимыми, а также позволяет приложениям-контейнерам отображать или печатать их даже в отсутствии приложения-сервера.

Вывод графики в контекст устройства обычно осуществляется через логические координаты. *Логические координаты* описывают объект, используя независимые от устройства реальные измерения, например, прямоугольник можно описать как два сантиметра шириной и один сантиметр высотой. GDI предлагает необходимые функциональные возможности для преобразования логических координат в физические.

Windows поддерживает простое преобразование из множества логических координат в множество физических координат. Это преобразование определяется значениями, указывающими начало координат, и знаковыми пределами логического и физического пространства. Начало координат обычно указывает горизонтальное и вертикальное смещение, а пределы определяют ориентацию и масштаб объектов после преобразования.

Функции GDI можно разбить на следующие группы:

1. Функции *рисования* различных объектов, таких как прямоугольники, эллипсы, многоугольники. Сюда же относятся функции для быстрого копирования растровых изображений.

2. Функции *управления контекстами устройств*. Контексты различных устройств могут быть созданы и разрушены, их состояние можно запомнить и затем восстановить.

3. Функции *преобразования координат*. Функции, общие для всех платформ Win32, можно использовать для установки и получения начала отсчета и *экстента окна* – пространства логических координат и *области просмотра* – пространства координат целевого устройства.

4. Функции для *работы с палитрой*. Такие функции наиболее эффективны для приложений, стремящихся достичь точности передачи цветов на устройстве, одновременно поддерживающем ограниченное количество цветов. Манипуляции с палитрой можно использовать и для анимации палитры – технологии, использующей изменение палитры для создания впечатления движения на экране.

5. Функции *создания GDI объектов*. Можно создавать кисти, перья, шрифты, растровые изображения или палитры и выбирать их в контекст устройства для определения внешнего вида нарисованных форм. Модуль GDI также предоставляет функции обработки шрифтов, включая шрифты TrueType.

6. Функций для *работы с метафайлами*. Можно создать, сохранить метафайл, загрузить и воспроизвести его на любом контексте устройства.

7. Функции для *работы с областями и отсечениями*. Отсечения крайне важны для среды Windows, так как они позволяют приложениям рисовать на экране, не обращая внимания на границы поверхности вывода.

Модуль User поддерживает системные вызовы для работы с элементами пользовательского интерфейса. Он включат в себя функции обработки сообщений окон, диалоговых панелей, меню, текстовых и графических курсоров, элементов управления, буфера обмена и многих других объектов. Именно благодаря функциям этого модуля становятся доступными высокоуровневые компоненты пользовательского интерфейса:

1. Функции *управления окнами* позволяют изменять размер, расположение, внешний вид окна. Сюда же относятся функции для обработки элементов управления – кнопок, полос прокрутки, полей ввода, а также функции для управления дочерними окнами многодокументного интерфейса.

2. Функции для *работы с меню* поддерживают функциональные возможности создания, и управления строками меню.

3. Функции управления *графическим и текстовым курсором*.

4. Функции управления *буфером обмена* Windows. Буфер обмена (clipboard) – это из механизмов Windows, посредством которого приложения могут обмениваться данными. Одно приложение может поместить данные в

буфер в общедоступных или частных форматах, а другое приложение может извлечь эти данные.

5. Функции управления *сообщениями и очередями сообщений* потоков. Приложения могут использовать эти вызовы для проверки содержимого своих очередей, извлечения и обработки сообщений и создания новых. Новые сообщения могут быть *посланы* или *отправлены* любому окну. Отправленное при помощи функции **`PostMessage()`** сообщение размещается в очереди сообщений потока, владеющего окном назначения. В противоположность этому, посланное при помощи функции **`SendMessage()`** сообщение непосредственно вызывает оконную процедуру окна назначения. Функция **`SendMessage()`** не завершается до тех пор, пока окно назначения не обработает сообщение. Этот механизм не только обходит очередь сообщений, но и позволяет посылающему сообщение приложению перед продолжением выполнения получить возвращаемое значение.

Модуль ядра (KERNEL) включает системные вызовы для управления процессами и потоками, управления ресурсами, файлами и памятью. Это не полный список, но все же эти категории описывают самые часто используемые функции этого модуля.

1. Предпочтительный метод управления *файлами* в Windows отличается от обычно используемого в программах на C/C++. Вместо доступа к файлам через стандартные библиотечные функции C для потоков, приложения должны использовать *файловый объект* Win32 и связанное с ним множество функций. Файловые объекты разрешают доступ к файлам способами, невозможными при использовании только библиотек C/C++, например, перекрывающийся ввод/вывод или размещенные в памяти файлы.

2. В противоположность этому, требования к управлению памятью в большинстве приложений полностью удовлетворяются семейством функций C **`malloc()`** или оператора C++ **`new`**. В Win32 приложениях эти вызовы автоматически транслируются в соответствующие системные вызовы управления памятью Windows. Для приложений с более высокими требованиями управления памятью существует набор функций управления виртуальной памятью.

3. Самым важным аспектом управления процессами и потоками является *синхронизация*. В среде с вытесняющей многозадачностью процессы и потоки не могут получить сведения о состоянии выполнения конкурирующих потоков. Чтобы обеспечить соответствующее выполнение взаимозависимых конкурирующих потоков и избежать тупиковой ситуации, когда два или более потоков останавливаются в неопределенности, ожидая друг друга, требуется сложный механизм синхронизации. В Win32 это выполняется с

помощью множества *объектов синхронизации*, которые потоки могут использовать для информирования других потоков о своем состоянии.

4. Функции для *работы с ресурсами пользовательского интерфейса*. Эти ресурсы содержат пиктограммы, курсоры, шаблоны диалоговых окон, таблицы акселераторов, растровые рисунки.

5. Функции, необходимые для 32-х разрядных программ с текстовым выводом – *консольных приложений*. Хотя на первый взгляд эти программы напоминают обыкновенные старые DOS программы, но на самом деле это полностью 32-х разрядные приложения, которые не используют графический интерфейс Windows. Тем не менее, эти приложения также имеют доступ к множеству системных вызовов Win32, например, консольные приложения могут использовать функции виртуальной памяти, а также они могут быть многопоточными программами.

5.2.15. Другие API

Windows – это намного больше, чем возможности, реализованные в рассматриваемых выше трех основных модулях. Существует множество модулей – других API, каждый из которых реализующий свое определенное множество функциональных возможностей. Укажем лишь наиболее используемые API:

1. API общих элементов управления используются для работы с новыми Win32 элементами управления, расширяющими множество стандартных элементов управления Windows.
2. API общих диалоговых окон содержит стандартные диалоговые окна для открытия файла, выбора цвета, выбора шрифта, определения операций поиска или замены.
3. MAPI (messaging applications programming interface – интерфейс программирования приложений на основе сообщений) предоставляет приложениям доступ к функциям сообщений через системы доставки почты типа Microsoft Mail.
4. MCI (multimedia control interface) предоставляет собой интерфейс управления средствами мультимедиа. Через функции MCI приложения могут получить доступ к видео-, аудио- и MIDI-возможностям Windows.
5. COM API – богатый набор системных вызовов, реализующих все аспекты функциональных возможностей OLE. К ним относятся функции контейнера и сервера OLE, активизации объектов, технологии “drag and drop”, а также элементы управления ActiveX.
6. TAPI – это телефонный API. Приложения могут использовать TAPI для аппаратно-независимого доступа к телефонным ресурсам: модемам, факс-модемам, устройствам речевых сообщений.

7. Группа API для поддержки сетевых возможностей – WinSock, WinInet, RAS и RPC.

6. Структура Windows приложений

Рассмотрим структуру простейшего Windows приложения.

```
        #define  STRICT
        #define  WIN32_LEAN_AND_MEAN
(1)     #include <Windows.h>

(2)     HINSTANCE g_hInst;
        LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);

(3)     int WINAPI WinMain (HINSTANCE hInstance,
                            HINSTANCE hPrevInstance,
                            LPSTR lpCmdLine, int nCmdShow)
        {
(4)       LPCTSTR szClass = "qwerty";
          LPCTSTR szTitle = "Simplest";
          g_hInst = hInstance;

(5)       WNDCLASS wc;
          wc.lpfnWndProc   = WndProc;
          wc.lpszClassName = szClass;
          wc.hInstance     = g_hInst;

          if (!RegisterClass(&wc)) return -1;

(6)       HWND hWnd = CreateWindowEx(
                WS_EX_OVERLAPPEDWINDOW,
                szClass, szTitle, WS_OVERLAPPEDWINDOW,
                0, 0, 100, 100,
                NULL, NULL, g_hInst, NULL);
          if (!hWnd) return -1;

(7)       ShowWindow(hWnd, nCmdShow);
          UpdateWindow(hWnd);

(8)       MSG msg;
          while (GetMessage(&msg, NULL, 0, 0))
                DispatchMessage(&msg);

          return 0;
        } // end of WinMain()

        //=========================================================
(9)     LRESULT CALLBACK WndProc (HWND hWnd, UINT message,
                                  WPARAM wParam, LPARAM lParam)
        {
          switch (message) {
          case WM_DESTROY :
                PostQuitMessage(0);
                return 0;
          }

          return DefWindowProc(hWnd, message, wParam, lParam);
        } // end of WndProc()
```

Итак, мы имеем C++ код Windows приложения, который вы можете откомпилировать и запустить на исполнение. Программа будет отображать на экране главное окно и ... ничего более. На данном этапе этого достаточно, поскольку наша задача сейчас – разобраться в структуре Windows приложений, понять как они работают, как взаимодействуют с операционной системой. Кроме того, такая *жесткая структура* повторяется во всех, даже самых сложных Windows программах.

Легко заметить, что организационно приложение состоит из двух функций: **`WinMain()`** и **`WndProc()`**. Первая является точкой входа в Windows приложение (вспомните функцию **`main()`** для DOS), а вторая – функция окна приложения. Ее имя не имеет значения, поскольку мы будем передавать Windows ее *адрес*, а не имя, но традиционно ее называют **`WndProc`**.

Теперь проведем пошаговый разбор того, что происходит в нашем приложении.

6.1. Файлы заголовков

Все Windows приложения включают стандартный заголовочный файл **`<Windows.h>`**, который содержит необходимые объявления. Включение макроса **`STRICT`** перед заголовочным файлом, позволяет задействовать строгую проверку соответствия типов данных Windows, а макрос **`WIN32_LEAN_AND_MEAN`** убирает излишнюю информацию из файла **`<Windows.h>`**, что сокращает время компиляции.

6.2. Глабальные переменные

В зависимости от сложности решаемой задачи, вы можете определить множество глобальных переменных, однако сохранить дескриптор экземпляра приложения придется любой Windows программе. Здесь же, по правилам C++, указан прототип функции окна, ссылка на которую производится раньше, чем определена сама функции.

6.3. Точка входа в приложение

Как уже было сказано, точкой входа в Windows приложение является функция **`WinMain()`**, которая *всегда* имеет четыре параметра:

`HINSTANCE hInstance` – дескриптор экземпляра приложения. Каждому запущенному приложению операционная система ставит в соответствие блок информации, которая сохраняется Windows в специальном списке все время, пока приложение находится в памяти компьютера. Указатель на этот блок является уникальным в системе и передается приложению в качестве дескриптора. Приложение использует этот дескриптор как параметр при функциональных запросах на выделение системных ресурсов.

HINSTANCE hPrevInstance – дескриптор экземпляра приложения, запущенного ранее. Операционная система Windows позволяет запускать несколько копий одного и того же приложения. В Win16 через этот параметр передавался дескриптор приложения, запущенного последним. Приложение, на основе этой информации, могло, например, запретить запуск повторных копий. В Win32 этот параметр *всегда* равен NULL.

LPSTR lpCmdLine – указатель на командную строку. Вы уже знакомы с командными строками, изучая DOS. Заметим, что командная строка в Windows не разбирается на составляющие опции.

int nCmdShow – параметр, указывающий как должно быть отображено главное окно приложения: свернутым в иконку, распахнутым на весь экран или нормальным образом.

6.4. Необходимые переменные

Вы должны определить имя регистрируемого класса. Строгих правил для выбора данного имени не существует, просто вы должны обеспечить его уникальность в системе. Помните, что некоторые имена классов уже определены в Windows, например, "**Edit**" или "**ListBox**". Здесь же вы можете задать строковую константу, которая будет использоваться как имя программы и будет отображена в заголовке окна.

Нужно сохранить дескриптор экземпляра приложения в глобальной переменной.

6.5. Регистрация класса окна

Перед регистрацией класса окна проводится заполнение полей структуры **WNDCLASS**, которая определена в заголовочном файле **<Windows.h>**. Мы используем только три основных поля, в частности, **wc.lpfnWndProc**, как указатель на функцию окна приложения, **wc.hInstance**, как значение дескриптора экземпляра приложения и **wc.lpszClassName**, как указатель на имя регистрируемого класса. Кроме того, вы можете указать:

- стиль или стили класса, например,

```
wc.style = 0;
```

- пикторгамму приложения, например, стандартную,

```
wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
```

- изображение курсора,

```
wc.hCursor = LoadCursor(NULL, IDC_ARROW);
```

- кисть для закрашивания фона окна,

```
wc.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
```

- имя ресурса меню. Меню обычно загружается для каждого окна в отдельности, а не для всего класса в целом, поэтому этот параметр равен NULL,

```
wc.lpszMenuName = NULL;
```

Полную описание всех полей структуры **WNDCLASS** вы найдете в любой справочной системе по Win32 API.

После заполнения полей структуры **WNDCLASS**, указатель на нее передается в функцию **RegisterClass(&wc)**, которая и проводит регистрацию класса окна.

> С этой точки Windows, зная адрес процедуры окна, может вызвать ее в любой момент.

Обязательно проверяйте код возврата, если он нулевой, то регистрация не состоялась, и работу приложения следует немедленно прекратить. Нужно заметить, что Windows, в отличие от DOS, игнорирует величину возвращаемого значения из приложения, поэтому вы можете использовать любой код возврата, а не (-1) как в примере.

6.6. Создание главного окна

Создание главного окна приложения производится вызовом функции **CreateWindow()** или **CreateWindowEx()**.

> Для многих функций Win32 API содержит два варианта, при этом один из них имеет суффикс **Ex**. Именно последние являются истинными Win32 функциями, их использование предпочтительнее. Функции без суффикса, как правило, унаследованы из Win16 и не являются 32-х разрядными. Кроме того, фирма Microsoft предупреждает, что она может прекратить поддерживать такие функции в будущем.

Кратко укажем значения параметров функции **CreateWindowEx()** (полную информацию о параметрах можно найти, например, в справочной системе компилятора VC++):

DWORD dwExStyle	дополнительные стили окна, определяющие, главным образом, вид обрамления окна;
LPCTSTR lpClassName	имя зарегистрированного ранее класса окна;
LPCTSTR lpWindowName	текст заголовка окна;
DWORD dwStyle	основные стили окна, определяющие его внешний вид.
int x, y,	координаты левого-верхнего угла окна, его
nWidth, nHeight	ширина и высота. Задаются в пикселях;
HWND hWndParent	дескриптор родительского окна или NULL для окон верхнего уровня (главных окон);

HMENU	**hMenu**	дескриптор меню приложения. Если меню отсутствует, то NULL;
HINSTANCE	**hInstance**	дескриптор экземпляра приложения, полученного через параметр WinMain();
LPVOID	**lpParam**	указатель на дополнительную информацию, ассоциированную с данным окном. Не используется, как правило, поэтому NULL.

В Windows поддерживается строгая иерархия окон, при этом *каждое* окно должно чему-либо принадлежать, т.е. иметь родительское окно. Единственным исключением является специальное окно под названием “DeskTop”, которое создается операционной системой при ее старте. На этом окне располагаются все видимые объекты самой Windows. Когда вы указываете нулевое значение для **hWndParent**, это означает, что истинным родительским окном будет “DeskTop”.

В Windows существуют *три основных стиля* окон:

- главные окна приложений имеют стиль **WS_OVERLAPPEDWINDOW**. Как правило, они располагаются на “DeskTop” и именуются окнами верхнего уровня;

- временные окна имеют стиль **WS_POPUPWINDOW**. Они принадлежат главным окнам приложений и служат обычно для вывода какой-либо информации. Типичный пример – диалоговые панели;

- дочерние окна, имеющие стиль **WS_CHILD**, не могут быть перемещены за границы родительского окна. Типичный пример – элементы управления диалогов.

Вернемся к точке (6) на нашей схеме. Если создание окна выполнено успешно, функция **CreateWindowEx()** возвращает его дескриптор. Под дескриптором окна следует понимать уникальное, в рамках операционной системы, значение, которое ассоциировано с каждым окном. Именно через этот дескриптор производятся все операции с конкретным окном.

В этой точке Windows, в своем ядре, создает *персональную очередь сообщений* для только что созданного окна. Сюда операционная система будет помещать сообщения, адресованные данному окну.

6.7. Отображение главного окна

Для того, чтобы окно появилось на экране нужно вызвать функцию **ShowWindow(hWnd, nCmdShow)**, которая принимает значение дескриптора окна и параметр, переданный через заголовок **WinMain()**. После этого, мы заставляем окно перерисовать свое содержимое, обращаясь к функции **UpdateWindow(hWnd)**, которая помещает сообщение **WM_PAINT** в очередь сообщений окна **hWnd**.

6.8. Цикл обработки сообщений

Ключевая точка каждого Windows приложения – цикл обработки сообщений. Как он работает?

Функция **GetMessage()** постоянно просматривает очередь сообщений и, если она пуста, не делает возврата, а передает управление Windows. Приложение находится в состоянии простоя или, как говорят, в состоянии "**idle**". Но как только в очереди появляется сообщение, **GetMessage()** заполняет соответствующие поля структуры **MSG**, удаляет это сообщение из очереди и возвращает *ненулевое* значение. В этом случае говорят – **GetMessage()** выбирает сообщение из очереди. После этого, управление передается функции **DispatchMessage(&msg)**, которая пересылает указатель на заполненную ранее структуру **MSG** обратно ядру Windows. Операционная система блокирует **DispatchMessage()** и вызывает оконную функцию (указатель на нее известен Windows) с параметрами, соответствующими полям структуры **MSG**. Функция окна выполняет действия, которые вы указали (или не указали) для обработки данного сообщения. После этого функция окна возвращает управление Windows, которая, в свою очередь, делает возврат из **DispatchMessage()**. Приложение возвращается к началу цикла **while**, т.е. вновь входит в функцию **GetMessage()**, ожидая следующего сообщения. Так продолжается до тех пор, пока в очереди сообщений не появится **WM_QUIT**. В этом случае **GetMessage()** возвращает *нулевое* значение, цикл **while** заканчивается, и приложение завершается.

> Цикл обработки сообщений и функция окна работают последовательно. Пока функция окна проводит обработку текущего сообщения, функция **GetMessage()** не может выбрать другое сообщение, вследствие блокировки цикла в **DispatchMessage()**. Значит, прежде чем перейти к обработке следующего сообщения, функция окна закончит обработку предыдущего. Это справедливо для асинхронных сообщений, но может быть нарушено для синхронных.

6.9. Функция окна

Функция окна – вторая ключевая точка каждого Windows приложения. От того, обработчики каких сообщений вы предусмотрите в оконной процедуре, зависит поведение вашего приложения. Можно сказать, что одно Windows приложение отличается от другого только наполнением оконных функций. Вся "жизнь" вашей программы определяется функцией окна.

Как уже было сказано, функция окна является функцией обратного вызова, что подчеркивается макросом **CALLBACK** в ее заголовке. Параметры функции *всегда* соответствуют первым четырем полям структуры **MSG**. За-

метим, что дескриптор окна передается первым параметром, следовательно нет необходимости запоминать его в глобальной переменной во время создания окна.

> Первый параметр функции окна подчеркивает тот факт, что данная функция будет обслуживать *все* окна, созданные на базе данного класса, отличая одно окно от другого по его уникальному дескриптору.

Наша первая функция окна весьма проста – она передает все поступающие сообщения на обработку по умолчанию, вызывая функцию API **`DefWindowProc()`**. Однако *каждое*, даже самое простое Windows приложение должно обработать сообщение **`WM_DESTROY`**, которое поступает в очередь сообщений, когда Windows разрушает ваше окно, что, в свою очередь, является следствием нажатия пользователем кнопки закрытия окна. Стандартным ответом вашей оконной функции должно быть обращение к API функции **`PostQuitMessage()`**, которая помещает последнее для приложения сообщение **`WM_QUIT`** в очередь сообщений. Это, как мы уже знаем, приводит к завершению цикла в **`WinMain()`**. Параметр функции **`PostQuitMessage()`** обычно не используется и равен нулю.

Таким образом, для создания Windows приложения вы обязательно должны написать две функции – **`WinMain()`** и **`WinProc()`**.

Функция **`WinMain()`** проводит регистрацию класса окна, создает главное окно приложения и запускает цикл обработки сообщений.

Оконная функция **`WinProc()`** определяет поведение приложения и должна обрабатывать сообщение **`WM_DESTROY`**, как минимум, чтобы завершить цикл обработки сообщений в **`WinMain()`**.

7. Обработка сообщений в оконной функции

Как было отмечено в предыдущем разделе, поведение Windows приложения определяется обработчиками сообщений в функции окна. О том, какие основные сообщения получает оконная функция, как их нужно обрабатывать, и будет посвящен данный раздел. Нет никакой необходимости обрабатывать *абсолютно все* сообщения Windows. Вы должны выявить главные для вашего конкретного приложения и сконцентрироваться на них.

Каждое получаемое окном сообщение идентифицируется номером, который содержится во втором параметре оконной процедуры, а дополнительная информация – в третьем и четвертом.

Если оконная функция обрабатывает сообщение, то ее возвращаемым значением, как правило, будет нулевое значение. Именно так нужно поступать, если в излагаемом ниже материале не приводится дополнительных сведений о коде возврата из обработчика.

Все сообщения, не обрабатываемые оконной процедурой, должны передаваться функции Windows **`DefWindowProc()`**. Такой механизм позволяет Windows проводить обработку сообщений окна совместно с приложением. При этом, значение, возвращаемое функцией **`DefWindowProc()`**, должно быть возвращаемым значением оконной процедуры.

Типичный вид функции окна мы рассмотрели в предыдущем разделе.

Напомним, что функция окна получает сообщение из двух источников: из цикла обработки сообщений и непосредственно от Windows. Из цикла обработки сообщений поступают все асинхронные сообщения, например, сообщения клавиатурного ввода, сообщения о перемещениях и нажатиях клавиш мыши, а также сообщения событий таймера. Непосредственно Windows вызывает функцию окна в виде реакции на синхронные сообщения.

7.1. Создание окна WM_CREATE

Это первое сообщение, которое получает функция окна. Его особенность в том, что оконная функция получает **`WM_CREATE`** только *один раз до возврата* управления из функции **`CreateWindowEx()`**. Это означает, что окно создано не до конца и, следовательно, очередь сообщений еще отсутствует. Отсюда следует вывод, что сообщение **`WM_CREATE`** является синхронным и посылается, минуя цикл обработки сообщений, который также еще не запущен. Поскольку сообщение посылается до того, как окно становится видимым, вы можете использовать его для какого-либо “досоздания” окна, например, для создания дочерних окон. В этом случае вам может понадобиться информация из структуры **`CREATESTRUCT`**, указатель на которую передается через параметр **`lParam`**.

Если обработчик сообщения возвращает (-1), создание главного окна прекращается. Если обработчик сообщения возвращает 0, создание окна продолжается, и функция **CreateWindowEx()** возвращает дескриптор нового окна. В этом случае, лучшим решением выхода из **WM_CREATE** будет передача управления на обработку по умолчанию.

7.2. Определение размера окна WM_SIZE

Это второе сообщение, которое получает функция окна после создания. Кроме того, оно поступает всякий раз, когда пользователь уже изменил размеры окна (в отличие от **WM_SIZING**, которое сигнализирует о том, что пользователь меняет размеры окна). Сообщение передается через очередь сообщений.

Windows приложение, как правило, не знает заранее размеров своих окон. К тому же в любой момент эти размеры можно изменить, например, при помощи мыши. Поэтому приложение должно быть готовым к такому изменению.

Дополнительные параметры сообщения несут информацию о новых размерах окна и о способе, которым оно изменило размер, а именно: **wParam** сообщает о способе изменения размеров. Особый интерес представляют значения **SIZE_MAXIMIZED** и **SIZE_MINIMIZED**, что дает возможность организовать собственную обработку ситуаций, когда окно распахнуто на весь экран или минимизировано. Младшее слово параметра **lParam** сообщает о новой ширине клиентской части окна, а старшее слово –о новой высоте.

Типичное использование **WM_SIZE** – посылка окном самому себе сообщения о перерисовке только измененной части окна. Как это сделать – мы рассмотрим в следующем сообщении.

> Для того, чтобы автоматически перерисовывалась *вся* клиентская часть окна при изменении его размеров, при регистрации класса окна определите поле стилей: **wc.style = CS_HREDRAW | CS_VREDRAW;**

Нужно заметить, что узнать текущие размеры клиентской части окна в любом обработчике можно вызвав функцию API:

```
GetClientRect (HWND hWnd, RECT* lpRect)
```

которая заполняет поля структуры **RECT** для окна с дескриптором **hWnd**.

7.3. Отображение содержимого окна WM_PAINT

Обработка сообщения **WM_PAINT** крайне важна для Windows программирования. Нужно помнить, что в мультипрограммной операционной системе в любой момент может потребоваться перерисовка любого, даже неактивного окна. Проще всего это можно сделать, сосредоточив всю работу

по отрисовке в одном месте, которым и является обработчик сообщения **WM_PAINT**. Это сообщение сигнализирует окну, что вся его рабочая область или некоторая ее часть становится недействительной (invalid), и ее следует перерисовать.

Здесь нужно пояснить, что окно разделяется на неклиентскую и рабочую (клиентскую) области. К неклиентской части относится заголовок окна и его обрамление. Перерисовка этой области – забота Windows, поскольку операционная система "знает" о стилях каждого окна, а они то и определяют, что нужно нарисовать в неклиентской части. Вся внутренняя часть окна относится к клиентской области, вот ее то перерисовку вы и должны запрограммировать в обработчике **WM_PAINT**, поскольку Windows не может знать, чего хочет каждый программист.

7.3.1. Случаи генерации сообщения WM_PAINT

Перечислим основные ситуации, когда клиентская область становится недействительной.

- При создании окна недействительна вся его рабочая область. Сообщение **WM_PAINT** помещается в очередь сообщений, когда приложение вызывает функцию **UpdateWindow()**.
- Увеличение (но не уменьшение!) размеров окна, в стиле класса которого заданы флаги **CS_HREDRAW** и **CS_VREDRAW**, приводит к тому, что вся рабочая область также становится недействительной. Операционная система вслед за этим посылает в очередь сообщение **WM_PAINT**.
- Пользователь минимизирует окно программы, а затем вновь восстанавливает его до прежнего размера. Операционная система не сохраняет содержимое рабочей области каждого окна, поскольку в графической среде это было бы слишком накладно. Вместо этого Windows объявляет недействительной всю клиентскую область окна, а затем оконная процедура получает сообщение **WM_PAINT** и сама восстанавливает содержимое окна.
- Во время перемещения окон Windows не сохраняет ту часть окна, которая перекрывается другими окнами. Когда же эта часть позже открывается, Windows вновь отмечает эту область как недействительную, а функция окна получает сообщение **WM_PAINT** для восстановления содержимого окна.
- Если при обработке любого сообщения требуется изменить содержимое окна, то приложение может объявить любую область собственного окна недействительной при помощи функции **InvalidateRect()**, а затем сообщить Windows, что необходимо перерисовать эту часть, вызвав функцию **UpdateWindow()**.

Функция **InvalidateRect()** объявлена следующим образом:

```
void InvalidateRect(HWND hwnd, const RECT* lprc, BOOL fErase)
```

Первый параметр – дескриптор окна, для которого выполняется операция. Второй – указатель на структуру типа **RECT**, определяющую прямоугольную область, подлежащую обновлению. Если указатель равен **NULL**, вся клиентская область объявляется недействительной. Третий параметр указывает на необходимость стирания фона окна, если он задан как **TRUE**, фон окна подлежит стиранию.

7.3.2. Особенность сообщения WM_PAINT

Первая особенность сообщения **WM_PAINT** состоит в том, что оно является низкоприоритетным. Операционная система помещает его в очередь сообщений только тогда, когда там нет других необработанных сообщений.

Вторая особенность – сообщение **WM_PAINT** аккумулируется системой. Это значит, что если в окне объявлены несколько областей, подлежащих обновлению, то приложение получает только одно сообщение **WM_PAINT**, в котором определена суммарная недействительная область, охватывающая все указанные части.

7.3.3. Правила обработки WM_PAINT

Обработка сообщения **WM_PAINT** *всегда* начинается с вызова функции **BeginPaint()**, а заканчивается – вызовом функции **EndPaint()**.

Функция **BeginPaint()** имеет прототип:

```
HDC BeginPaint (HWND hwnd, PAINTSTRUCT* lpPaint)
```

Первый параметр – дескриптор окна, а второй – указатель на структуру **PAINTSTRUCT**, поля которой содержат информацию, полезную для проведения профессиональной отрисовки в рабочей области окна. Так, поле **RECT rcPaint** представляет прямоугольник, охватывающий все области окна, требующие перерисовки. Ограничив свои действия по рисованию только этой прямоугольной областью, приложение, несомненно, ускоряет процесс перерисовки.

При обработке вызова **BeginPaint()**, Windows обновляет [или *не обновляет*, это зависит от последнего параметра, с которым была вызвана функция **InvalidateRect()** перед этим] фон рабочей области с помощью кисти, которая указывалась при регистрации класса окна. Поле **fErase** структуры **PAINTSTRUCT** указывает произведено ли перекрашивание фона клиентской области или нет. Знание этого может быть полезным, чтобы не делать повторной закраски в обработчике.

Возвращаемым значением **BeginPaint()** является дескриптор контекста устройства. Этот дескриптор необходим для вывода в рабочую область текста и графики. Заметим, что при использовании данного дескриптора контекста устройства приложение не может рисовать вне клиентской области.

Функция **EndPaint()**, принимающая те же параметры, что и **BeginPaint()**, выполняет обязательное освобождение дескриптора контекста устройства. Обработчик **WM_PAINT** заканчивается вызовом этой функции.

7.3.4. Отрисовка вне WM_PAINT

Функции **BeginPaint()** и **EndPaint()** используются *только* в обработчике **WM_PAINT**. Нужно воспользоваться другими функциями Win32 API, если вашему приложению требуется получить контекст устройства клиентской области окна *вне* **WM_PAINT**. Примерный фрагмент такого кода должен выглядеть следующим образом:

```
// Получить контекст
HDC hDC = GetDC(hWnd);
// операции с дескриптором контекста
// и его освобождение
ReleaseDC(hWnd, hDC);
```

После получения контекста и выполнения каких-либо операций, его обязательно нужно освободить посредством функции **ReleaseDC()**.

7.3.5. Определение возможностей контекста устройства

В программном интерфейсе Windows имеется функция, позволяющая по контексту устройства определить возможности и параметры драйвера, обслуживающего устройство ввода-вывода:

```
int GetDeviceCaps(HDC hdc, int iCapability)
```

Первый параметр функции задает контекст устройства, для которого необходимо получить информацию о его возможностях. Получить контекст устройства можно от функций **BeginPaint()** при обработке сообщения **WM_PAINT** или **GetDC()** при обработке иных сообщений. Второй параметр функции **GetDeviceCaps()** определяет параметр устройства, значение которого необходимо получить.

Остановимся на некоторых из возможных значений.

Значения **ASPECTX, ASPECTY** и **ASPECTXY** определяют размеры пиксела. Дело в том, что пикселы не всегда квадратные. Поэтому масштаб изображения по оси x может отличаться от масштаба по оси y. Размеры пиксела позволяют вычислить отношение сторон пиксела и выполнить правильное масштабирование. В этом случае отображаемые окружности будут

круглыми, а квадраты – квадратными. К счастью, для подавляющего большинства современных видеоадаптеров такая коррекция не требуется.

Иногда бывает важно знать цветовое разрешение устройства вывода. Для этого можно использовать значение **BITPIXEL**, которое соответствует количеству бит **nPixel**, используемых для представления цвета. Если возвести число 2 в степень значения **BITPIXEL**, получится количество цветов, которое может быть представлено одним пикселом.

Некоторые устройства работают с цветовыми плоскостями. Количество этих плоскостей **nPlanes** можно определить, пользуясь значением **PLANES**. Тогда количество одновременно отображаемых цветов можно определить по формулу **nColors** = $2^{(nPixel * nPlanes)}$.

7.3.6. Системные метрики

Метрики системных компонент Windows можно определить при помощи функции

```
int GetSystemMetrics(int nIndex)
```

Единственный аргумент этой функции задает параметр, значение которого необходимо определить. Значение требуемого параметра возвращается данной функцией.

Для определения того или иного компонента в заголовочных файлах Windows имеются константы с префиксом **SM_**. Здесь перечислены некоторые из них:

SM_CXCURSOR , **SM_CYCURSOR** – ширина и высота курсора.

SM_CXICON, **SM_CYICON** – ширина и высота пиктограммы.

SM_CXSCREEN, **SM_CYSCREEN** – разрешение экрана.

SM_CYCAPTION – высота заголовка окна.

SM_CYMENU – высота одной строки в полосе меню.

SM_CYHSCROLL – высота горизонтальной полосы прокрутки.

SM_CXVSCROLL – ширина вертикальной полосы прокрутки.

7.4. Определение расположения окна WM_MOVE

При обсуждении сообщения **WM_SIZE** было показано, как следует определять размеры окна. Другая важная задача – определение расположения окна на экране.

При *завершении* перемещения окна его функция получает сообщение **WM_MOVE** (в отличие от сообщения **WM_MOVING**, которое поступает *во время* перемещения), а вместе с ним – новые координаты окна в параметре **lParam**:

```
xPos = LOWORD(lParam);
yPos = HIWORD(lParam);
```

Для окон, имеющих стили **WS_OVERLAPPED** и **WS_POPUP**, координаты отсчитываются от верхнего левого угла экрана. Для окон стиля **WS_CHILD** эти координаты отсчитываются от верхнего левого угла внутренней области родительского окна.

В любом обработчике, а не только в **WM_MOVE**, можно узнать расположение окна на экране с помощью функции

```
GetWindowRect(HWND hWnd, RECT* lpRect)
```

которая заполняет поля структуры **RECT** текущими координатами окна с дескриптором **hWnd**.

7.5. Использование оконных полос прокрутки

Нередко возникает ситуация, когда выводимая в окно информация, допустим, многострочный документ, превышает текущие размеры окна. Стандартным решением этой задачи в Windows программировании является использование полос прокрутки, которые позволяют проводить вертикальный или горизонтальный скроллиг содержимого окна.

Добавить в окно приложения вертикальную или горизонтальную полосу прокрутки не представляет затруднений. Все, что необходимо сделать, это включить битовые маски **WS_VSCROLL** и **WS_HSCROLL** в список стилей окна, создаваемого функцией **CreateWindowEx()**. Следует заметить, что клиентская область окна не включает в себя пространство, занятое полосами прокрутки. Ширина вертикальной полосы и высота горизонтальной полос прокрутки постоянны для конкретного дисплейного драйвера. Эти значения можно получить с помощью функции **GetSystemMetrics()**.

Заметим, что после создания окна с полосами прокрутки, они будут присутствовать на экране, но будут бездействовать, т.к. программирование логики их работы – ваша задача.

Заметим также, что Windows обеспечивает только логику работы мыши с полосами прокрутки. Однако у полос прокрутки отсутствует клавиатурный интерфейс. Если приложение желает дублировать клавишами клавиатуры некоторые функции полос прокрутки, оно должно самостоятельно реализовать эту логику при обработке клавиатурных сообщений.

7.5.1. Диапазон и положение полос прокрутки

Каждая полоса прокрутки имеет соответствующий диапазон – два целых числа, отражающих минимальное и максимальное значение, и положение бегунка – его местоположение внутри диапазона. При этом, положение бегунка всегда дискретно. Например, для полосы прокрутки с диапазоном от 0 до 4 имеется пять возможных положений бегунка.

Win32 API содержит две функции, которые позволяют выполнить все операции с полосами прокрутки. Это:

```
GetScrollInfo(HWND hwnd, int fnBar, SCROLLINFO* lpsi)

SetScrollInfo(HWND hwnd, int fnBar, SCROLLINFO* lpsi,
                                              BOOL fRedraw)
```

Первая из них возвращает, через поля заполненной структуры **SCROLLINFO**, всю необходимую информацию о полосе прокрутки, а вторая – позволяет установить новые значения для полосы.

В качестве первого параметра этих функций задается дескриптор окна, владеющего полосой прокрутки. Параметр **fRedraw** устанавливается в **TRUE**, если необходимо, чтобы Windows перерисовала полосы в соответствии с новыми значениями. Параметр **fnBar** указывает на горизонтальную (**SB_HORZ**) или вертикальную (**SB_VERT**) полосу. Третий параметр **lpsi** является указателем на структуру **SCROLLINFO**, которая имеет следующие поля:

`cbSize` размер структуры в байтах. Вы можете указать `cbSize=sizeof(SCROLLINFO);`

`fMask` флаг, определяющий какие из полей структуры будут заполнены. Так, если `fMask=SIF_RANGE`, будут заполнены значения диапазона полосы, если `fMask=SIF_PAGE`, то поле `nPage`, если `fMask=SIF_POS`, то поле `nPos`. Можно комбинировать значения, принимаемые флагом, для того, чтобы выбрать (или установить) интересующую информацию за одно обращение к функции.

`nMin` минимальное значение диапазона полосы.

`nMax` максимальное значение диапазона полосы.

`nPage` объем информации, укладывающийся на одной странице.

`nPos` текущее положение бегунка. `nMin <= nPos <= nMax`.

`nTrackPos` текущее положение бегунка при протаскивании его мышью.

> При создании полосы Windows, по умолчанию, устанавливает минимальное значение диапазона в 0, а максимальное – в 100.

Если вам требуется убрать полосу просмотра у окна, установите `nMin=0` и `nMax=0`.

Значение параметра `nPage` влияет на размер бегунка полосы, что вы можете наблюдать в ряде современных Windows приложений. Чтобы этого достичь в вашем приложении, рассчитайте сколько, например, строк документа размещается в видимой части окна и установите это значение вызовом **SetScrollInfo()**. Windows определит сколько страниц укладывается во всем документе и, в соответствии с этим, установит размер бегунка – чем меньше размер, тем больше исходный документ. Если хотите сохранить

постоянные размеры бегунка при любом объеме документа, установите `nPage=0`.

Прежде, чем перейти к обработке сообщений полос просмотра, укажем сферы ответственности Windows и приложения при поддержке полос прокрутки.

Сферы ответственности Windows:

1. Управление логикой работы мыши с полосой прокрутки.
2. Обеспечение временной “инверсии цвета” при нажатии на кнопку мыши в полосе прокрутки.
3. Перемещение бегунка в соответствие с тем, как внутри полосы прокрутки его перемещает пользователь с помощью мыши.
4. Отправка сообщений полосы прокрутки в оконную процедуру окна, содержащего полосу.

Сферы ответственности приложения по поддержке полос покрутки:

1. Инициализация диапазона полосы прокрутки.
2. Обработка сообщений полосы прокрутки.
3. Обновление положения бегунка полосы прокрутки.

7.5.2. Сообщения полос прокрутки

Windows посылает оконной процедуре сообщения **`WM_VSCROLL`** (для вертикальной полосы) и **`WM_HSCROLL`** (для горизонтальной), когда пользователь щелкает мышью на полосе прокрутки или перетаскивает бегунок. Параметры этих сообщений:

```
nScrCode = LOWORD(wParam);
nPos     = HIWORD(wParam);
```

Следует отметить, что при работе с оконными полосами прокрутки параметр **`lParam`** игнорируется, поскольку он используется *только* для полос прокрутки как элементов управления.

Для Win32 приложений также следует игнорировать старшее слово параметра **`wParam`**, представляющего значение положения бегунка. Его нужно получать через функцию **`GetScrollInfo()`**.

Оставшееся младшее слово параметра **`wParam`** данных сообщений показывает какие действия мышью осуществляются на полосе прокрутки. Этот код операций может принимать следующие значения для вертикальной полосы:

`SB_TOP`	скроллиг к началу документа.
`SB_BOTTOM`	скроллиг к концу документа.
`SB_LINEUP`	прокрутка на одну строку вверх.
`SB_LINEDOWN`	прокрутка на одну строку вниз.
`SB_PAGEUP`	скроллиг на одну страницу вверх.

SB_PAGEDOWN скроллиг на одну страницу вниз.
SB_THUMBTRACK протаскивание бегунка мышью.

И значения для горизонтальной полосы:

SB_LEFT скроллиг на одну страницу влево.
SB_RIGHT скроллиг на одну страницу вправо.
SB_LINELEFT прокрутка на одну строку влево.
SB_LINERIGHT прокрутка на одну строку вправо.
SB_THUMBTRACK протаскивание бегунка мышью.

Алгоритм обработчика прокрутки должен включать обновление положения ползунка в соответствии с новым значением, расчет отображаемой части документа и перерисовку клиентской области окна.

Обработкик будет работать значительно эффективнее, если воспользоваться функцией **ScrollWindowEx()**, позволяющей осуществлять быстрое перемещение прямоугольной части клиентской области в пределах окна.

7.6. Клавиатурный ввод

Архитектура Windows, основанная на сообщениях, идеальна для работы с клавиатурой. Приложение узнает о нажатиях клавиш посредством сообщений, которые посылаются оконной процедуре. Когда пользователь нажимает и отпускает клавиши, драйвер клавиатуры передает эту информацию в Windows, которая сохраняет данную информацию в *системной очереди сообщений*, а оттуда – в очередь сообщений окна приложения.

Но какого окна? Ответ – окна, имеющего *фокус ввода*. Смысл этого двухступенчатого процесса – сохранение сообщений в системной очереди сообщений и дальнейшая их передача в очередь сообщений приложения – в синхронизации. Если пользователь нажимает клавиши клавиатуры быстрее, чем приложение может обрабатывать поступающую информацию, Windows сохраняет информацию о дополнительных нажатиях клавиш в системной очереди сообщений.

Одно из дополнительных нажатий может быть переключением фокуса ввода на другую программу. Значит информацию о последующих нажатиях следует направлять окну другого приложения.

Именно таким образом операционная система корректно синхронизирует события клавиатуры.

Для отражения различных событий клавиатуры, Windows посылает программе восемь различных сообщений. Приложения вполне могут игнорировать многие из них. Кроме того, в большинстве случаев в этих сообщениях от клавиатуры содержится значительно больше закодированной ин-

формации, чем нужно приложению. Залог успешной работы с клавиатурой – это знание того, какие сообщения важны для приложения, а какие нет.

7.6.1. Фокус ввода и активное окно

Клавиатура должна разделяться между всеми приложениями, работающими под Windows. Некоторые приложения могут иметь больше одного окна, и клавиатура должна разделяться между этими окнами в рамках одного и того же приложения. Когда на клавиатуре нажата клавиша, только одна оконная процедура может получить сообщение об этом. Окно, которое в настоящее время получает все клавиатурные сообщения, именуется окном, имеющим *фокус ввода*.

Концепция фокуса ввода тесно связана с концепцией *активного окна*. Окно, имеющее фокус ввода – это либо активное окно, либо дочернее окно активного окна.

Определить активное окно достаточно просто:

- Windows выделяет заголовок активного окна;
- если у активного окна вместо заголовка имеется рамка диалога, то Windows выделяет ее;
- если активное окно минимизировано, то Windows выделяет текст заголовка в панели задач.

> Если активное окно минимизировано, то окна с фокусом ввода нет. Windows продолжает слать программе сообщения клавиатуры, но эти сообщения выглядят иначе, чем сообщения, направленные активным и еще не минимизированным окнам.

Можно обрабатывать сообщения **`WM_SETFOCUS`** и **`WM_KILLFOCUS`**, чтобы определить какое окно имеет фокус ввода, однако эти сообщения носят чисто информирующий характер. Программный интерфейс Windows содержит две функции, позволяющие узнать или изменить окно, владеющее фокусом ввода, – **`GetFocus()`** и **`SetFocus()`.**

7.6.2. Генерация клавиатурных сообщений

Сообщения, которые приложение получает от Windows о событиях, относящихся к клавиатуре, различаются на *аппаратные* (keystrokes) и *символьные* (characters). Такое положение соответствует двум представлениям о клавиатуре. Во-первых, клавиатуру можно считать набором клавиш. В клавиатуре имеется только одна клавиша <A>. Нажатие и отпускание этой клавиши представляют собой аппаратные события. Во-вторых, клавиатура также является устройством ввода, генерирующим отображаемые символы. Клавиша <A>, в зависимости от состояния клавиш <Ctrl>, <Shift> и <CapsLock>, может стать источником нескольких символов. Обычно, этим

символом является строчная, латинская ‘a’. Если же нажата клавиша <Shift> или установлен режим <CapsLock>, то этим символом является прописная ‘A’. Если же нажата клавиша <Ctrl>, этим символом будет <Ctrl+A>.

Для сочетаний двух аппаратных событий, которые генерируют отображаемые символы, Windows посылает программе и аппаратные, и символьные сообщения. Некоторые клавиши не генерируют символов. Это такие клавиши, как клавиши переключения, функциональные клавиши, клавиши управления курсором и специальные клавиши, например, <Insert> и <Delete>. Для таких клавиш Windows вырабатывает только аппаратные сообщения.

7.6.3. Аппаратные сообщения

Когда пользователь нажимает клавишу, Windows помещает в очередь окна, имеющего фокус ввода, либо сообщение **WM_KEYDOWN,** либо сообщение **WM_SYSKEYDOWN**. Когда же клавиша отпускается, Windows посылает в очередь либо сообщение **WM_KEYUP,** либо сообщение **WM_SYSKEYUP**. Сообщения **WM_SYS...** вырабатываются при нажатии клавиш в сочетании с клавишей <Alt>.

Итак, **WM_KEYDOWN** и **WM_KEYUP** это несистемные аппаратные сообщения , а системные – **WM_SYSKEYDOWN** и **WM_SYSKEYUP.**

Обычно сообщения о нажатии и отпускании появляются парами. Однако, если пользователь оставит клавишу нажатой так, чтобы включился автоповтор, то Windows посылает оконной процедуре серию сообщений **WM_KEYDOWN** или **WM_SYSKEYDOWN** и *только одно* сообщение **WM_KEYUP** или **WM_SYSKEYUP**, когда в конце концов клавиша будет отпущена. Аппаратные сообщения клавиатуры являются асинхронными сообщениями, и становятся в очередь сообщений окна.

7.6.3.1. Системные аппаратные сообщения

WM_SYSKEYDOWN и **WM_SYSKEYUP** – системные аппаратные сообщения более важны для Windows, чем для приложений. Эти сообщения генерируются при нажатии клавиш в сочетании с клавишей <Alt>. Они вызывают опции меню программы или системного меню, или используются для системных функций, таких как смена активного приложения по <Alt+Tab>.

Программы обычно игнорируют системные аппаратные сообщения и передают их в функцию **DefWindowProc()**. Оконная процедура в конце концов получает другие сообщения, являющиеся результатом этих аппаратных сообщений клавиатуры, например, выбор меню.

Если приложение проводит обработку системных аппаратных сообщений, после этого *обязательно* передавайте их в **DefWindowProc()**,

чтобы Windows могла использовать эти сообщения в своих целях. Если этого не сделать, то происходит *полная блокировка всех операций* с клавишей <Alt>.

7.6.3.2. Несистемные аппаратные сообщения

Для клавиш, которые нажимаются и отпускаются без участия клавиши <Alt>, генерируются несистемные сообщения **WM_KEYDOWN** и **WM_KEYUP**. Приложение может использовать или не использовать эти сообщения клавиатуры. Это не влияет на работу операционной системы, поскольку Windows их полностью игнорирует.

7.6.3.3. Битовые поля параметра lParam

Для всех – системных и несистемных – аппаратных сообщений клавиатуры 32-х разрядная переменная **lParam**, передаваемая в оконную процедуру, состоит из следующих полей:

Биты	Значение
00-15	счетчик повторений (repeat count); равен числу нажатий клавиши. В большинстве случаев 1. Однако, если клавиша остается нажатой, а оконная процедура недостаточно быстра, чтобы обрабатывать эти сообщения в темпе автоповтора, то Windows объединяет несколько аппаратных сообщений о нажатии клавиши в одно сообщение и соответственно увеличивает счетчик повторений. При отпускании клавиши счетчик всегда 1.
16-23	скан-код OEM (Original Equipment Manufacturer scan code); код клавиатуры, генерируемым аппаратурой компьютера. Приложения Windows обычно игнорируют скан-код OEM.
24	флаг расширенной клавиатуры (extended key flag); устанавливается в 1, если сообщение клавиатуры появилось при работе с дополнительными клавишами расширенной клавиатуры IBM. Расширенная клавиатура имеет функциональные клавиши сверху и отдельную комбинированную область клавиш управления курсором и цифр.
25-28	зарезервировано;
29	код контекста (context code); Устанавливается в 1, если нажата клавиша <Alt>. Этот разряд всегда равен 1 для системных аппаратных сообщений и 0 для несистемных аппаратных сообщений клавиатуры за исключением одного случая: если активное окно минимизировано, оно не имеет фокуса ввода, при этом *все нажатия* клавиш вырабатывают сообщения **WM_SYSKEYDOWN** и **WM_SYSKEYUP**.
30	флаг предыдущего состояния клавиши; 1 если клавиша была нажата перед этим;

31	флаг состояния клавиши (transation state).

7.6.3.4. Виртуальные коды клавиш

Гораздо более важным параметром аппаратных сообщений клавиатуры является параметр **wParam**. В этом параметре содержится *виртуальный код клавиши* (virtual key code), идентифицирующий нажатую или отпущенную клавишу.

Параметр **wParam** содержит код виртуальной клавиши, соответствующей нажатой клавише. Именно этот параметр используется приложением для идентификации клавиши. Код виртуальной клавиши не зависит от аппаратной реализации клавиатуры. Коды виртуальных клавиш имеют символьные обозначения, определенные в заголовочных файлах Windows, и имеют префикс **VK_**.

7.6.4. Символьные сообщения

Для того, чтобы символьные сообщения клавиатуры появились в очереди сообщений окна нужно дополнить цикл обработки сообщений приложения:

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

Функция **GetMessage()** по-прежнему заполняет поля структуры **msg** данными следующего сообщения из очереди, а **DispatchMesage()** отправляет **msg** на обработку в оконную процедуру. Между ними находится функция **TranslateMessage()**, преобразующая аппаратные сообщения клавиатуры в символьные сообщения в соответствии с состоянием драйвера клавиатуры, а также положением управляющих клавиш <Shift> и <CapsLock>. Именно благодаря **TranslateMessage()**, в очереди сообщений появляются символьные сообщения, а приложение сможет отличить <ф> от <A> при активном драйвере кириллицы. Символьное сообщение *всегда* будет следующим, после сообщения о нажатии клавиши, которое функция **GetMessage()** извлечет из очереди.

Существует четыре вида символьных сообщений – **WM_CHAR** и **WM_DEADCHAR** относятся к несистемным и приходят вслед за **WM_KEYDOWN**. Сообщения **WM_SYSCHAR** и **WM_SYSDEADCHAR** являются системными и вызваны появлением сообщения **WM_SYSKEYDOWN**.

7.6.5. Обработка сообщения WM_CHAR

В большинстве случаев программы для Windows могут игнорировать все клавиатурные сообщения за исключением **WM_CHAR**. Параметр **lParam**, передаваемый в оконную процедуру, является таким же, как и параметр **lParam** аппаратного сообщения, из которого сгенерировано символьное сообщение. Параметр **wParam** – это код символа ANSI.

Наиболее типичный обработчик сообщения **WM_CHAR** выглядит следующим образом:

```
case WM_CHAR:
      switch((char)wParam) {
      case 'b': // получен символ Backspace
            //....
            break;
            //....
      case '\t':
            //....
            break;
      case 'A':
            //....
            break;
      case 'a':
            //....
            break;
      //....
      }
return 0;
```

7.6.6. Определение состояния управляющих клавиш

Параметры **wParam** и **lParam** аппаратных сообщений клавиатуры ничего не сообщают о состоянии управляющих клавиш <Shift>, <Ctrl>, <Alt> и клавиш переключателей <CapsLock>, <NumLock>, <ScrollLock>. Приложение может получить текущее состояние любой виртуальной клавиши с помощью функции **GetKeyState()**. Например:

```
case WM_KEYDOWN:
// Нажата ли комбинация <Ctrl>+S ?
if (wParam == 'S' &&
   (0x8000 & GetKeyState(VK_CONTROL))) {
      // ваши действия
}
return 0;
```

Функция **GetKeyState()** не отражает состояние клавиатуры в реальном времени. Она позволяет узнать состояние клавиатуры *на момент, когда последнее сообщение от клавиатуры было выбрано из очереди*. Следовательно, ее можно использовать только в обработчиках клавиатурных

сообщений. Недостаток? Это обращается преимуществом, потому что **GetKeyState()** обеспечивает возможность получения точной информации, даже если сообщение обрабатывается асинхронно, уже после того, как состояние переключателя было изменено.

Если действительно нужна информация о *текущем положении* клавиши, то можно использовать функцию **GetAsyncKeyState()**.

7.6.7. Наборы символов

В операционной системе MS-DOS использовался расширенный набор символов ASCII, определенный фирмой IBM, в частности, был разработан расширенный набор символов ASCII с кириллицей. В терминологии Windows такие таблицы кодов называются наборами символов OEM (Original Equipment Manufacturer).

Сама же система Windows для представления символов использует набор символов ANSI. В этом наборе определены не все коды и отсутствуют символы псевдографики.

Если программа MS-DOS запускается в окне Windows, для нее выбирается набор символов OEM. Поэтому в Windows используются как набор символов ANSI, так и набор символов OEM. По умолчанию в контекст отображения выбирается системный шрифт, для которого используется набор символов ANSI.

Для одинаковых символов наборы ANSI и OEM используют разные коды, это приводит к необходимости перекодировки символов, например, при переносе текстов, подготовленных в среде MS-DOS, в среду Windows. В составе программного интерфейса Windows имеются функции, которые берут на себя работу по преобразованию и перекодировке символов.

Для перекодировки строк символов используют Win32 API функции **CharToOem(), CharToOemBuff(), OemToChar(), OemToCharBuff().**

Для преобразования символов в строчные или прописные приложение Windows должно пользоваться функциями **CharLower(), CharUpper(), CharLowerBuff(), CharUpperBuff().**

Еще одна проблема связана с необходимостью позиционирования указателя в текстовой строке. Если используется однобайтовое представление символов, позиционирование сводится к увеличению или уменьшению значения указателя на один байт. Однако в некоторых национальных алфавитах набор символов ANSI для представления символов использует два байта. Для правильного позиционирования необходимо использовать специальные функции **CharNext(), CharPrev().**

7.7. Системный таймер

Таймер в Windows можно отнести к устройству ввода информации, которое периодически извещает приложение о том, что истек заданный интервал времени. Приложение сообщает операционной системе интервал времени, а затем Windows периодически сигнализирует программе об истечении этого интервала. Это достигается посылкой сообщения **WM_TIMER**.

Случаи применения таймера в Windows:

1. Режим автосохранения – таймер может предложить программе сохранять работу пользователя на диске, когда истекает заданный интервал времени.
2. Поддержка обновления информации – программа может использовать таймер для вывода на экран обновляемой в реальном времени, постоянно меняющейся информации, связанной либо с системными ресурсами, либо с процессом выполнения отдельной задачи.
3. Поиск другого приложения, запущенного из текущего.
4. Многозадачность – хотя Windows является вытесняющей многозадачной средой, иногда самое эффективное решение для программы – как можно быстрее вернуть управление Windows. Если программа должна выполнять большой объем работы, она может разделить задачу на части и обрабатывать каждую часть при получении сообщения от таймера.
5. Задание темпа изменения – графические объекты в играх или окна с результатами в обучающих программах могут нуждаться в задании установленного темпа изменения.

Поскольку приложения Windows получают сообщения **WM_TIMER** из обычной очереди сообщений, приложение не должно беспокоиться о том, что его работа будет прервана внезапным сообщением **WM_TIMER**. В этом смысле таймер похож на клавиатуру и мышь: драйвер обрабатывает асинхронные аппаратные прерывания, а Windows преобразует эти прерывания в регулярные, структурированные, последовательные сообщения.

Сообщения таймера ставятся в очередь сообщений и обрабатываются как все остальные сообщения, т.е. это сообщение не выделяется среди других, наоборот – оно имеет *наименьший приоритет*.

> Операционная система Windows *не является операционной системой реального времени*.

Особенности Windows таймера.

1. *Не гарантируется*, что приложение будет получать сообщение **WM_TIMER** точно по истечению заданного интервала, он будет колебаться. Если приложение занято больше чем секунду, то оно *вообще не получит* ни одного сообщения **WM_TIMER** в течение этого времени. Фактически,

Windows обрабатывает сообщение **WM_TIMER** во многом так же, как сообщения **WM_PAINT**. Оба эти сообщения имеют низкий приоритет, и приложение получит их, только если в очереди нет других сообщений.

2. Сообщения **WM_TIMER** похожи на сообщения **WM_PAINT** и в другом смысле – Windows *никогда* не хранит в очереди сообщений несколько сообщений **WM_TIMER**. Вместо этого Windows объединяет несколько сообщений таймера в одно. В результате у приложения нет возможности определить число "потерянных" сообщений **WM_TIMER**.

Windows программа может запустить сколько угодно таймеров. Каждый из них характеризуется уникальным – в рамках приложения – идентификатором. Присоединить таймер к Windows окну можно при помощи вызова функции

```
UINT SetTimer(HWND hWnd, UINT nIDEvent, UINT uElapse,
                                        TIMERPROC lpTimerFunc);
```

Второй параметр функции идентифицирует таймер, а третий – задает интервал в миллисекундах. Это значение определяет темп, с которым Windows будет посылать приложению сообщения **WM_TIMER**.

Для остановки потока сообщений от таймера приложение должно вызвать функцию

```
BOOL KillTimer(HWND hWnd, UINT uIDEvent);
```

Вызов **KillTimer()** очищает очередь сообщений от всех необработанных сообщений **WM_TIMER**.

> Приложение должно перед завершением программы уничтожить все активные (запущенные) таймеры.

7.7.1. Использование таймера. Первый способ

Если при запуске таймера определить последний параметр функции **SetTimer()** как NULL, это будет сигналом для Windows генерировать сообщения **WM_TIMER**, которые будут обрабатываться *в обычной оконной процедуре* вашего приложения. При этом код может выглядеть следующим образом:

```
#define TIMER_SEC   1
#define TIMER_MIN   2

//. . . . .

SetTimer(hWnd, TIMER_SEC,  1000, NULL);
SetTimer(hWnd, TIMER_MIN, 60000, NULL);
```

Первый параметр функции **SetTimer()** – это дескриптор того окна, чья оконная процедура будет получать сообщения **WM_TIMER**. Вторым параметром является идентификатор таймера. Его значение должно быть

больше нуля. Третий параметр – это 32-х разрядное беззнаковое целое, которое задает интервал в миллисекундах. Например, значение 1000 задает генерацию сообщений **WM_TIMER** один раз в секунду.

Когда оконная процедура получает сообщение **WM_TIMER**, значение **wParam** равно значению идентификатора таймера, а **lParam** для данного случая NULL.

Значение параметра **wParam** позволяют отличать передаваемые в оконную процедуру сообщения **WM_TIMER** от разных таймеров. Логика обработки сообщения **WM_TIMER** выглядит примерно так:

```
case  WM_TIMER:
KillTimer(hWnd, wParam); // остановить таймер
switch(wParam) {
      case TIMER_SEC: // раз в секунду
      // ваши действия ...
      // вновь запускает таймер
      SetTimer(hWnd, TIMER_SEC, 1000, NULL);
      break;

      case TIMER_MIN: // один раз в минуту
      // ваши действия ...
      // вновь запускает таймер
      SetTimer(hWnd, TIMER_MIN, 60000, NULL);
      break;
}

return 0;
```

Для того, чтобы установить новое время срабатывания для существующего таймера, следует сначала его остановить функцией **KillTimer()** и вновь запустить при помощи **SetTimer()**.

7.7.2. Использование таймера. Второй способ

При первом способе установки таймера сообщения **WM_TIMER** посылаются в обычную оконную процедуру. С помощью второго способа можно заставить Windows пересылать сообщения другой функции этого же приложения.

Функция, которая будет получать эти таймерные сообщения должна быть функцией обратного вызова (**CALLBACK**). Эта функция приложения, которую, как и оконную процедуру, вызывает Windows. Приложение должно сообщить Windows адрес этой функции, а позже Windows вызывает именно ее, а не оконная процедуру. Как и оконная процедура, функция обратного вызова должна определяться как **CALLBACK**, поскольку Windows вызывает ее вне кодового пространства программы.

В случае функции обратного вызова для таймера, входными параметрами являются те же параметры, что и параметры оконной процедуры. Од-

нако таймерная функция обратного вызова не имеет возвращаемого в Windows значения.

Допустим, что в качестве имени таймерной функции обратного вызова выбрано имя **TimerProc**, тогда эта функция, которая будет обрабатывать *только* сообщения **WM_TIMER**, должна иметь следующее определение:

```
void CALLBACK TimerProc(HWND hwnd, UINT uMsg,
                        UINT idEvent, DWORD dwTime)
{
       KillTimer(hWnd, idEvent);    // остановить таймер
       if (idEvent == TIMER_SEC) { // раз в секунду
              // ваши действия ...
              // вновь запускает таймер
              SetTimer(hWnd, TIMER_SEC, 1000, TimerProc);
       }
}
```

Первый параметр для **TimerProc()** – дескриптор окна, задаваемый при вызове функции **SetTimer().** Поскольку Windows будет посылать функции **TimerProc()** только сообщения **WM_TIMER,** следовательно, параметр **uMsg** всегда равен **WM_TIMER**. Значение **idEvent** – идентификатор таймера, а значение **dwTime** – системное время.

При использовании функции обратного вызова для обработки сообщений **WM_TIMER**, четвертый параметр функции **SetTimer()** должен быть адресом функции обратного вызова:

```
SetTimer(hWnd, TIMER_SEC, 1000, TimerProc);
```

Второй способ обычно применяется, чтобы разгрузить оконную функцию. На первый взгляд получение значения системного времени тоже относится к преимуществам второго способа. Однако обращение к Win32 API функции

```
GetSystemTime (SYSTEMTIME* lpSystemTime)
```

позволяет решить эту задачу в произвольном месте вашего приложения.

7.8. Удаление окна, сообщение WM_DESTROY

Сообщение **WM_DESTROY** появляется в очереди сообщений одним из последних. Оно показывает, что Windows находится в процессе ликвидации окна в ответ на полученную от пользователя команду. Пользователь вызывает поступление этого сообщения, если нажмет на кнопке закрытия окна, выберет пункт “Закрыть” из системного меню или нажмет комбинацию клавиш Alt+F4.

Функция главного окна стандартно реагирует на это сообщение, вызывая функцию **PostQuitMessage(0)**, которая ставит последнее для приложения сообщение **WM_QUIT** в очередь сообщений. Это заставляет функ-

цию **WinMain()** закончить цикл обработки сообщений и выйти в систему, завершив работу приложения.

Таким образом, мы рассмотрели основные принципы написания обработчиков сообщений, однако к этой теме мы будем возвращаться при обсуждении других аспектов Windows программирования.

8. Ресурсы приложения и их использование

Подавляющее большинство Windows программ включает множество графических элементов, именуемых ресурсами (resources) Windows.

Перечислим виды ресурсов в Win32 API:

1) меню, диалоги, панели управления, акселераторы;
2) иконки, битовые изображения, курсоры;
3) строковые таблицы, таблица версии приложения;
4) ресурсы, определяемые разработчиком.

Все перечисленные ресурсы размещаются в *файле ресурсов*, что представляет собой одну из самых характерных особенностей Windows программирования. Именно с помощью файлов ресурсов большинство приложений определяет визуальные элементы своего пользовательского интерфейса – меню, диалоговые окна, надписи, растровые изображения и прочее.

Файлы ресурсов, имеющие расширение **.rc* создаются по известным спецификациям, в текстовом формате. Это позволяет создавать файл ресурсов приложения либо вручную – любым текстовым редактором, либо специальным редактором ресурсов, что предпочтительнее. Затем этот файл компилируется *компилятором ресурсов*. Полученный в результате файл с расширением **.res* далее компонуется с остальными частями приложения – **.obj* и **.lib* файлами, образуя единый двоичный образ, содержащий выполняемый программный код и информацию о ресурсах. При этом в заголовке каждого Windows приложения формируется специальная *таблица ресурсов*. Эта таблица используется Windows для поиска и загрузки ресурсов в оперативную память.

Хотя ресурсы являются данными и хранятся в *exe* файле программы, но расположены они не в сегменте данных (DS), где хранятся обычные данные исполняемых модулей. Таким образом, к ресурсам нет непосредственного доступа через переменные, определенные в исходном тексте программы. Они должны быть явно загружены из *exe* файла в память.

Основная причина такого построения – экономия оперативной памяти. Действительно, трудно придумать ситуацию, когда *абсолютно все* ресурсы могут одномоментно понадобиться приложению. А раз в каждый момент нужны только некоторые, то более разумно – подгружать их по мере надобности.

Нужно сказать, что ресурсы не обязательно должны компоноваться с исполняемым файлом приложения, они также могут сводиться в отдельную библиотеку DLL. Преимущество этого подхода заключается в том, что при изменении файла ресурсов не требуется перекомпилировать все приложение, а нужно только заменить DLL файл. Можно также сопровождать приложение несколькими DLL, представляющими ресурсы на различных язы-

ках. Именно так поступают разработчики при создании приложения, предназначенного для работы в многоязыковой среде.

Еще одно преимущество – с точки зрения технологии программирования – использования ресурсов состоит в том, что локализация приложения в этом случае требует наименьших затрат.

8.1. Меню приложения

Меню – это наиболее консервативная и, одновременно, наиболее важная часть пользовательского интерфейса. Трудно найти Windows приложение, у которого отсутствует меню, – это считается “дурным тоном” у разработчиков.

8.1.1. Виды меню

Обычно выделяют четыре вида меню:

1. Главное меню окна или меню верхнего уровня (*top-level menu*). Это меню представляет собой горизонтальную строку, которая расположена непосредственно под заголовком окна и состоит из нескольких элементов.

2. Подменю (*submenu*) появляется под главным меню при выборе одного из его элементов.

3. Плавающее меню (*floating menu*). Оно не связано с главным меню, и может быть создано в любом месте экрана как независимое всплывающее меню.

4. У подавляющего большинства главных окон приложений в левой части заголовка находится пиктограмма. Щелчок мышью по ней приводит к появлению системного меню (*system menu*), которое похоже на подменю главного меню приложения. Обычно системное меню всех приложений одинаково, с его помощью можно минимизировать или максимизировать окно приложения, перемещать его и прочее. Однако приложение имеет возможность изменить системное меню, дополняя его новыми строками, или удаляя существующие.

8.1.2. Возможные состояния пунктов меню

1. При выборе пункта меню, строка инвертирует цвет. Это применимо к пунктам меню всех видов.

2. Пункты всплывающих меню могут находиться в состоянии *отмечено* (checked), при этом слева от текста пункта меню выставляется специальная метка. Эта метка позволяет пользователю узнать о том, какие опции программы выбраны из этого меню. Пункты главного меню *не могут быть отмечены*.

3. Пункты меню могут находиться в состоянии *активно* (enabled), *неактивно* (disabled) и *недоступно* (grayed). Имейте в виду, что пункты, поме-

ченные как активные или неактивные, выглядят для пользователя одинаково, а недоступный пункт меню выводится в сером цвете.

Только при выборе *активных* пунктов меню Windows генерирует сообщения, поступающие в функцию окна, содержащего это меню.

Итак, каждый пункт меню определяется тремя характеристиками:

1. Внешний вид пункта меню, другими словами – то, что будет отображено в меню. Это может быть либо строка текста, либо графический, точнее битовый образ.

2. Атрибут пункта меню, который определяет, является ли данный пункт активным, недоступным или отмеченным.

3. Уникальный числовой идентификатор, который Windows посылает в функцию окна, когда пользователь выбирает данный пункт меню.

8.1.3. Сообщения от пунктов меню

При выборе пунктов меню Windows генерирует сообщение **WM_COMMAND**. Это сообщение можно назвать сообщением пользовательского интерфейса, поскольку оно посылается всякий раз, когда пользователь что-то выбирает, что-то меняет и так далее. В общем случае это сообщение имеет параметры:

```
WM_COMMAND
wEvent  = HIWORD(wParam);
wID     = LOWORD(wParam);
hWndCtl = (HWND)lParam;
```

Параметр **wEvent** именуется нотификационным кодом или кодом извещения. Для пунктов меню он *всегда* равен нулю. Параметр **wID** – это уникальное числовое значение, ассоциированное с каждым элементом. В случае меню, он содержит идентификатор пункта меню. Параметр **hwndCtl** представляет собой дескриптор окна элемента управления. Для пунктов меню он *всегда* равен NULL.

При выборе пунктов системного меню генерируется сообщение **WM_SYSCOMMAND**, которое имеет такие же параметры.

Напомним еще раз, что сообщение **WM_COMMAND** поступает в функцию окна, когда пункт меню *уже выбран*. Однако решаемая вами задача может потребовать знания, какой пункт меню *выбирается* пользователем в настоящий момент. Допустим, вы хотите динамически менять внешний вид пункта меню, когда пользователь наводит на него курсор мыши. В этом случае функция окна должна обрабатывать сообщение **WM_MENUSELECT**, которое поступает до **WM_COMMAND** и имеет параметры:

```
Item   = LOWORD(wParam);
fFlags = HIWORD(wParam);
hMenu  = (HMENU)lParam.
```

Параметры **Item** и **hMenu** соответствуют идентификатору и дескриптору выбираемого пункта меню, а значение **fFlags** содержит флаги состояния этого пункта.

8.1.4. Создание главного меню приложения

Добавление меню к программе – сравнительно простая задача Windows программирования. Структура меню достаточно просто определяется в описании ресурсов. Каждому пункту меню присваивается числовой идентификатор. Обеспечение уникальности идентификаторов берут на себя современные редакторы ресурсов, вам остается придумать мнемонические имена этим идентификаторам, например **ID_EXIT**.

Для создания меню применяются три метода:

1. Шаблон меню создается в файле ресурсов приложения, а затем загружается при создании главного окна приложения. Этот способ подходит для статических меню, неменяющихся или незначительно меняющихся в процессе работы программы.

2. Меню без шаблона создается динамически только при помощи функций Win32 API. Этот способ подходит для приложений, существенно меняющих внешний вид меню, когда разработать подходящий шаблон заранее не представляется возможным.

3. Шаблон меню подготавливается непосредственно в оперативной памяти и с помощью специальных функций подключается к приложению.

Мы, преимущественно, будем использовать первый способ. Как было сказано, прежде чем подключить меню к приложению, его нужно загрузить из ресурсов. Последовательность действий может быть такой:

```
HMENU hMenu = LoadMenu(g_hInst, "Main_Menu");
HWND   hWnd = CreateWindowEx(
      WS_EX_OVERLAPPEDWINDOW,
      szClass, szTitle, WS_OVERLAPPEDWINDOW,
      0, 0, 100, 100,
      NULL, hMenu, g_hInst, NULL);
if (!hWnd) return -1;
```

Заметим, что при указании ресурсов приложения чаще используются числовые значения, идентифицирующие ресурсы, а не строковые описания, например, **IDR_MAIN_MENU**. В этом случае следует использовать макрос **MAKEINTRESOURCE()** для преобразования идентификатора ресурса в строковое описание. Пример загрузки меню выглядит так:

```
HMENU hMenu = LoadMenu(g_hInst,
              MAKEINTRESOURCE(IDR_MAIN_MENU));
```

Обработчик сообщений меню несложен и может выглядеть следующим образом:

```
case WM_COMMAND: {
    int wID = LOWORD(wParam);
    switch (wID) {
       case ID_EXIT :
           DestroyWindow(hWnd);
           break;
    }
    return 0;  }
```

8.1.5. Функции для работы с меню

1. Если окно **hWnd** приложения имеет главное меню, то получить его дескриптор можно с помощью функции

```
HMENU hMenuMain = GetMenu(hWnd);
```

Затем можно получить дескрипторы всех подменю, вызывая функцию

```
HMENU hSubMenu = GetSubMenu(hMenuMain, nPos);
```

Здесь целочисленное значение **nPos** изменяется от 0 до **nMenu**-1, где переменная **nMenu** обозначает количество подменю. Узнать это значение можно с помощью универсальной функции

```
nMenu = GetMenuItemCount(HMENU hMenu),
```

которая возвращает количество элементов в *любом* меню, а не только в главном.

2. Для полной замены меню у главного окна приложения нужно воспользоваться функцией

```
BOOL SetMenu (HWND hWnd, HMENU hMenuNew),
```

которая позволяет прикрепить к окну **hWnd** новое меню с дескриптором **hMenuNew**.

> В этом случае “старое” меню требуется разрушить вызовом функции **DestroyMenu(hMenu)**. Если же меню прикреплено к окну при создании последнего (см. пример предыдущего пункта), Windows автоматически разрушит такое меню при закрытии окна.

3. Изменить состояние пункта меню с идентификатором **uID** можно с помощью вызова функции

```
EnableMenuItem(hMenu, uID, uEnable);
```

Параметр **uEnable** может принимать одно из следующих значений: **MF_DISABLED** (недоступно), **MF_ENABLED** (активно) или **MF_GRAYED** (недоступно и отмечено серым цветом). Это значение должно быть объединено с константой **MF_BYCOMMAND**, если параметр **uID** задает идентификатор меню, либо с константой **MF_BYPOSITION**, если **uID** задает номер позиции в меню.

4. Если вы изменяете состояние какого-либо пункта *главного меню* окна **hWnd**, то следует выполнить перерисовку меню вызовом функции

```
DrawMenuBar(hWnd);
```

Перерисовывать подменю не требуется, поскольку это делается Windows автоматически при каждом вызове подменю.

5. Программный интерфейс Win32 содержит две универсальные функции

```
BOOL GetMenuItemInfo(HMENU hMenu, UINT uItem,
      BOOL fByPosition, MENUITEMINFO* pMI)

BOOL SetMenuItemInfo(HMENU hMenu, UINT uItem,
      BOOL fByPosition, MENUITEMINFO* pMI)
```

первая из которых возвращает полную информацию о пункте любого меню, а вторая – позволяет совершить все необходимые операции с этим пунктом. Параметр **uItem** является позицией в меню, если переменная **fByPosition** имеет ненулевое значение, либо идентификатором – в противном случае. Исчерпывающее описание полей структуры **MENUITEMINFO** вы можете найти в любой справочной системе по Win32 API.

8.2. Стандартные элементы управления

Следующим после меню, часто используемым ресурсом приложения, является диалог. Практически любое стандартное приложение Windows использует диалоговые панели. Однако вначале следует рассмотреть составляющие компоненты диалогов – элементы управления.

В Windows предопределен целый ряд различных элементов управления, таких, как кнопки, редакторы текстов и списки и прочее. Эти элементы управления именуются дочерними окнами управления (*child window control*). Все они создаются на базе предопределенных классов, но разработчик может определить и собственные классы дочерних окон, зарегистрировав их при помощи функции **RegisterClass()**.

Вспомним, что дочернее окно, во-первых, определяется стилем **WS_CHILD** и *всегда* располагаются на поверхности родительского окна, как бы “прилипая” к нему. Во-вторых, при любом перемещении дочернее окно никогда не выходит за границы родительского окна. Родительское окно может содержать несколько элементов управления, которые будут перемещаться вместе с окном-родителем.

> Чтобы родительское окно различало дочерние окна, последние должны иметь уникальный идентификатор и уникальный дескриптор окна.

Итак, достаточно просто создать нужные дочерние окна, указав их размеры, расположение и некоторые другие атрибуты. После этого приложение может взаимодействовать с элементами управления, передавая им и

получая от них различные сообщения. При этом каждый элемент управления самостоятельно обрабатывает сообщения мыши и клавиатуры и извещает родительское окно о том, что его состояние изменилось. В этом случае дочернее окно становится для родительского окна устройством ввода. При этом оно инкапсулирует особые действия, связанные с графическим представлением окна на экране, реакцией на пользовательский ввод, и извещения другого окна при вводе любой информации. Приложению нет необходимости беспокоиться о логике обработки мыши этими окнами, или о логике их отрисовки. Все это входит в зону ответственности Windows, а все, что остается приложению – это обрабатывать сообщение **WM_COMMAND**. Этим сообщением дочерние окна информируют оконную процедуру о различных событиях.

Win32 API содержит новые элементы управления, которые используют сообщение **WM_NOTIFY** для извещения родительского окна.

8.2.1. Создание стандартных элементов управления

Динамически – вне шаблона – создать стандартный элемент управления проще всего с помощью функции **CreateWindow()**, используя один из предопределенных классов: “**button**”, ”**edit**”, “**static**”, “**listbox**”, “**combobox**” и “**scrollbar**”.

Например, кнопку можно создать так:

```
HWND hWndBotton = CreateWindow(
       “button”, “Отмена”,
       dwStyle, x, y, nWidth, nHeight,
       hWndParent,
       (HMENU)nIDctrl,
       g_hInst, NULL);
```

Параметр **nIDctrl** – числовой идентификатор окна. Для каждого создаваемого дочернего окна необходимо определить собственный уникальный идентификатор. Родительское окно будет различать элементы управления по этому параметру, получая сообщение **WM_COMMAND** от *всех* дочерних окон. Дескриптор **hWndParent** – это описатель родительского окна.

Кроме того, нужно указать конкретные параметры стиля **dwStyle** создаваемого окна для более точного определения вида и свойств каждого из элементов управления.

Оконные процедуры для стандартных элементов управления уже включены в состав ядра Windows. Они необходимы для обработки сообщений тех дочерних окон, которые созданы на основе перечисленных классов.

8.2.2. Разрушение элементов управления

При необходимости созданный элемент управления можно удалить функцией **DestroyWindow().** Однако помните, при разрушении родительского окна, Windows *сама удаляет* все его дочерние окна.

8.2.3. Функции для работы с элементами управления

Вызывая функции Win32 API, можно динамически перемещать элементы управления, делать их активным или неактивным, скрывать или, наоборот, отображать в окне.

Переместить элемент управления внутри родительского окна можно с помощью функции **SetWindowPos()**. Эта функция определяет новое расположение и размеры окна в системе координат, связанной с родительским окном.

```
BOOL SetWindowPos (HWND hWnd,
      HWND hWndInsertAfter,
      int X, int Y, int cx, int cy,
      UINT uFlags)
```

Кроме того, данная функция позволяет манипулировать, так называемым z-порядком окон. Эта задача становится актуальной, когда на поверхности родительского окна располагается несколько перекрывающихся дочерних окон, и одно окно нужно "убрать" под другое, или наоборот – вывести его на передний план.

Функция

```
BOOL EnableWindow (HWND hWndChild, BOOL bEnable)
```

делает окно **hWndChild** либо активным, если второй параметр **TRUE**, либо недоступным, если **bEnable** равен **FALSE**.

Функция

```
BOOL IsWindowEnabled (HWND hWndChild)
```

проверяет активно ли окно **hWndChild** в настоящий момент.

Функция

```
BOOL ShowWindow (HWND hWndChild, int nCmdShow)
```

позволяет скрыть (`nCmdShow=SW_HIDE`) или вновь отобразить (`nCmdShow=SW_SHOW`) окно **hWndChild** на поверхности родительского окна.

Укажем еще две вспомогательные функции. Одна из них дает возможность узнать дескриптор дочернего окна, зная его целочисленное значение идентификатора **nID**:

```
HWND GetDlgItem (HWND hWndParent, int nID)
```

а вторая, обратная первой – возвращает идентификатор дочернего окна по его известному дескриптору `hWndChild`:

```
UINT GetDlgCtrlID (HWND hWndChild)
```

Хотя часть "Dlg" в именах этих функций относится к окнам диалога, на самом деле – это функции общего назначения.

8.2.4. Сообщения дочерних окон

Если стандартный элемент управления изменяет свое состояние, то функция родительского окна получает сообщение **WM_COMMAND** со следующими параметрами:

```
WM_COMMAND
int  nIDCtrl = LOWORD(wParam);
int  nEvent  = HIWORD(wParam);
HWND hWndCtl = (HWND) lParam;
```

Код уведомления **nEvent** – это дополнительный код, который дочернее окно использует для того, чтобы сообщить родительскому окну более точные сведения о сообщении. Константы, идентифицирующие различные коды уведомления, определены в заголовочных файлах Windows и имеют соответственно следующие префиксы: **BN_** – "button", **EN_** – "edit", **LBN_** – "listbox", **CBN_** – "combobox" и **SB_** – "scrollbar".

Параметр **nIDCtrl** – это уникальное числовое значение, которое указывается при создании элемента управления.

Параметр **hwndCtl** представляет собой дескриптор окна элемента управления.

8.2.5. Сообщения родительского окна дочерним окнам

Родительское окно может передавать сообщения дочерним окнам, в ответ на которые это дочернее окно будет выполнять различные действия. Передать можно как обычное оконное сообщение (с префиксом **WM_**), так и специфические для каждого типа элемента управления. Константы, идентифицирующие различные сообщения для дочерних окон управления, определены в заголовочных файлах Windows и имеют соответственно следующие префиксы: **BM_** – "button", **EM_** – "edit", **LB_** – "listbox", **CB_** – "combobox".

Напомним, что существует два способа передачи сообщений.

Асинхронный способ – это запись сообщения в очередь приложения. Он основан на использовании функции **PostMessage()**. Эта функция помещает сообщение в очередь сообщений для окна, указанного в параметрах, и сразу же возвращает управление. Позже (асинхронно по времени) записанное при помощи функции **PostMessage()** сообщение будет выбрано и обработано в цикле обработки сообщений.

Синхронный способ – это непосредственная передача сообщения функции дочернего окна, минуя очередь сообщений. Этот метод реализуется функцией **SendMessage()**. В отличие от предыдущей функции, функция

SendMessage() вызывает оконную процедуру и возвращает управление только после выхода из функции дочернего окна.

Для элементов управления синхронный способ является основным, что подчеркивается наличием в Win32 API специально для этого предназначенного варианта функции **SendMessage()**

```
LONG SendDlgItemMessage(
       HWND hWndParent, int nIDitem,
       UINT Msg, WPARAM wParam, LPARAM lParam)
```

которая отправляет сообщение **Msg** дочернему окну с идентификатором **nIDitem**, расположенному на родительском окне **hWndParent**.

8.2.6. Расширенное управление дочерними окнами

Рассмотрим эту возможность на примере решения задачи об изменении цвета фона элемент управления. Когда элемент управления собирается отрисовать свою рабочую область, он посылает процедуре родительского окна соответствующее сообщение с префиксом **WM_CTLCOLOR** (например, для EditBox это сообщение **WM_CTLCOLOREDIT**). Родительское окно на основании этой информации может изменить цвет, который будет использован в оконной процедуре дочернего окна при рисовании. В итоге родительское окно может управлять цветами своих дочерних окон.

> Возвращаемым значением для сообщений `WM_CTLCOLOR...` является *дескриптор* на кисть `hBrush`.

8.2.7. Оконные процедуры элементов управления

Как уже отмечалось, функции окон элементов управления находятся в ядре Windows. Однако у разработчика имеется возможность получить адрес этой оконной процедуры с помощью функции **GetWindowLong()**, для которой в качестве второго параметра используется идентификатор **GWL_WNDPROC**. С помощью же функции **SetWindowLong()** можно не только получить адрес оконной процедуры элемента управления, но и установить новую оконную процедуру для него. Этот очень мощный прием именуется *window subclassing*. Он позволяет дополнять код стандартной оконной процедуры новыми возможностями, другими словами, “влезть” в существующие внутри ядра Windows оконные процедуры, обработать некоторые сообщения специфическим для приложения способом, а все остальные сообщения оставить для прежней оконной процедуры. Например, необходимо создать элемент EditBox, который предназначен для ввода только вещественных чисел. Последовательность ваших действий должна быть следующей:

1. Объявить глобальную переменную, как указатель на CALLBACK функцию

```
static WNDPROC OldEditWindowProc;
```

Указатель OldEditWindowProc будет использован для получения адреса оконный процедуры стандартного элемента управления EditBox. Применение глобального класса памяти объясняется тем, что использовать его придется в различных обработчиках.

2. В обработчике **WM_CREATE** родительского окна создается элемент управления, допустим имеющий дескриптор hEdit. После этого, нужно получить адрес "старой" оконный процедуры для hEdit

```
OldEditWindowProc = (WNDPROC)GetWindowLong(
              hEdit, GWL_WNDPROC);
```

и уставить "новую" функцию окна NewEditWindowProc

```
SetWindowLong(hEdit,
              GWL_WNDPROC, (LONG)NewEditWindowProc);
```

3. Новая оконная процедура, конечно же, является функцией обратного вызова и определяется следующим образом:

```
LRESULT CALLBACK NewEditWindowProc (HWND hEdit,
       UINT uMsg, WPARAM wParam, LPARAM lParam)

{
     switch (uMsg) {
     // ваши обработчики
     case ...

     case WM_DESTROY :
       // при разрушении элемента управления
       // нужно восстановить старый адрес.
       SetWindowLong(hEdit, GWL_WNDPROC,
              (LONG)OldEditWindowProc);
     }

     return CallWindowProc(OldEditWindowProc,
              hEdit, uMsg, wParam, lParam);
}
```

Обратите внимание на то, что обработка по умолчанию для подмененной оконной процедуры выполняется с помощью **CallWindowProc()**, а не **DefWindowProc()**.

8.2.8. Элемент управления кнопка

Рассмотрим особенности каждого стандартного элемента управления в отдельности. Материал по каждому элементу состоит из трех частей:

- стили, характерные для элемента управления;
- сообщения, поступающие в оконную функцию родительского окна;

- управляющие сообщения от родительского окна.

8.2.8.1. Стили кнопок

При создании кнопок функцией **CreateWindow()** можно использовать следующие дополнительные стили:

BS_PUSHBUTTON	Обычная кнопка
BS_DEFPUSHBUTTON	Создается так называемая “кнопка по умолчанию”. Этот стиль применяется только для диалоговых панелей. Тогда при нажатии клавиши Enter родительскому окну (диалогу) посылается сообщение о нажатии кнопки, имеющей этот стиль. Только одна кнопка на диалоговой панели может иметь такой стиль.
BS_NOTIFY	Если кнопка имеет данный стиль, родительскому окну посылаются сообщения **BN_DBLCLK**, **BN_KILLFOCUS** и **BN_SETFOCUS**. Заметим, что основное “кнопочное” сообщение **BN_CLICKED** посылается вне зависимости от этого стиля.
BS_OWNERDRAW	Кнопка с расширенными возможностями отрисовки, когда графическое отображение элемента становится обязанностью приложения, а не Windows. Кнопка с этим стилем посылает родительскому окно сообщения **WM_MEASUREITEM**, при создании кнопки, и **WM_DRAWITEM**, всякий раз, когда требуется перерисовка. Этот стиль несовместим с другими стилями.
BS_MULTILINE	На поверхности кнопки отображается многострочная надпись, а не однострочная – как в большинстве случаев.
BS_CENTER **BS_VCENTER**	Центрирование текста надписи. Несовместимо со стилем **BS_MULTILINE**.

8.2.8.2. Сообщения от кнопок, получаемые родительским окном

Ниже указаны сообщения, которые кнопка посылает своему родительскому окну. Часть из них является самостоятельными Windows сообщениями, а другие приходят в составе сообщения **WM_COMMAND**. В последнем случае старшее слово параметра **wParam** соответствует указанному нотификационному значению:

BN_CLICKED (WM_COMMAND)	Кнопка нажата.
BN_DBLCLK (WM_COMMAND)	Двойной щелчок левой клавишей мыши на поверхности кнопки. Это сообщение генерируется только для кнопок, имеющих стиль **BS_OWNERDRAW**.
BN_SETFOCUS BN_KILLFOCUS, (WM_COMMAND)	Кнопка получает или теряет фокус ввода.
WM_CTLCOLORBTN	Посылается родительскому окну, когда кнопке требуется перерисовать свой фон. Родительское окно должно вернуть дескриптор кисти для закрашивания фона.
WM_DRAWITEM	Сообщение генерируется только для кнопок, имеющих стиль **BS_OWNERDRAW**. Уведомляет о том, что родительское окно должно выполнить графическое отображение кнопки. Параметры сообщения: **wParam** - идентификатор кнопки **lParam** - указатель на структуру **DRAWITEMSTRUCT**. Описание полей структуры смотри в пункте 8.2.9.

8.2.8.3. Сообщения от родительского окна к кнопке

Родительское окно может изменить состояние кнопки, посылая ей сообщение **BM_SETSTATE**. Если параметр **wParam** установлен в значение **TRUE**, кнопка переводится с состояние “нажато”, если **wParam** равен **FALSE**, то в состояние “отжато”. Параметр **lParam** для данного сообщения не используется и должен быть равен нулю.

8.2.8.4. Кнопки-переключатели

Обычно переключатель не выделяют в самостоятельный элемент управления, а рассматривают его как разновидность кнопки, т.е. элемент управления, требующий нажатия.

При создании переключателей функцией **CreateWindow()** можно использовать следующие дополнительные стили:

BS_AUTOCHECKBOX	Переключатель, который может находиться только в двух состояниях: включено и выключено.
BS_AUTO3STATE	Переключатель, который может находиться в трех состояниях: включено, выключено и неактивно.

BS_AUTORADIOBUTTON Группа переключателей, при этом только один из группы может быть в состоянии включено, все остальные - выключены.

8.2.8.5. Сообщение от переключателей

Родительское окно получает от переключателей ранее рассмотренное сообщение **BN_CLICKED** в составе **WM_COMMAND**.

8.2.8.6. Сообщение от родительского окна к переключателям

1. Родительское окно с дескриптором **hWnd** может запросить состояние переключателя, который имеет идентификатор **nID**, через сообщение **BM_GETCHECK**. Например:

```
int k=SendDlgItemMessage(hWnd,nID,BM_GETCHECK,0,0);
```

Возвращается одно из следующих состояний переключателя:

BST_CHECKED включен;
BST_UNCHECKED выключен;
BST_INDETERMINATE неопределенное состояние.

Последний код ответа может иметь только переключатель, для которого был определен стиль **BS_AUTO3STATE**.

2. Родительское окно может изменить состояние переключателя, посылая ему сообщение **BM_SETCHECK**, значение **wParam** параметра которого должно принимать одну из указанных выше величин, а параметр **lParam** не используется и должен быть равен нулю.

8.2.9. Структура DRAWITEMSTRUCT

Структура **DRAWITEMSTRUCT** используется при отрисовке пунктов меню, а также для некоторых элементов управления (см. поле **CtlType** в приведенной ниже таблице), которые имеют стиль **OWNERDRAW**. Как уже было сказано, этот стиль сообщает Windows, что отрисовку необходимых элементов приложение берет под свой контроль. Обязанностью Windows остается заполнение полей структуры, которая содержит информацию, что и как нужно перерисовать. После этого указатель на структуру передается в функцию родительского окна для обработки. Заметим, что оконная функция будет получать *столько* сообщений **WM_DRAWITEM**, *сколько пунктов* требуется перерисовать. Перечислим допустимые значения для полей структуры:

CtlType Определяет тип элемента управления, который имеет стиль **OWNERDRAW**:

- **ODT_MENU** меню;
- **ODT_BUTTON** кнопка;
- **ODT_LISTBOX** обычный список;
- **ODT_LISTVIEW** расширенный список (Win32);
- **ODT_COMBOBOX** комбинированный список;
- **ODT_STATIC** статический текст;
- **ODT_TAB** набор закладок (Win32).

CtlID идентификатор элемента управления. Не используется для меню.

itemID номер пункта меню или элемента списков, которые требуют перерисовки.

itemAction Определяет тип перерисовки:

- **ODA_DRAWENTIRE** перерисовка всего пункта;
- **ODA_FOCUS** пункт имеет фокус ввода;
- **ODA_SELECT** пункт является выбранным.

itemState Состояние пункта:

- **ODS_DEFAULT** обычное состояние;
- **ODS_FOCUS** имеет фокус ввода;
- **ODS_GRAYED** недоступен (только для меню);
- **ODS_DISABLED** недоступен (только для меню);
- **ODS_CHECKED** отмечен (только для меню);
- **ODS_SELECTED** выбран (только для меню).

hwndItem Дескриптор элемента управления. Не используется для меню.

hDC Дескриптор контекста отображения.

rcItem Структура **RECT**, ограничивающая прямоугольную область, которая требует перерисовки.

itemData Двойное слово, ассоциированное с данным пунктом.

8.2.10. Стандартный элемент управления окно ввода

Задавая класс “edit”, приложение определяет прямоугольное дочернее окно, в которое пользователь может вводить текст с клавиатуры. Характерная особенность этого органа управления – наличие мигающего курсора ввода информации. Кроме того, в данном окне функционируют все клавиши редактирования.

> Для Windows 95/98 объем текста в одном окне ввода не может превышать 32К.

По умолчанию, Windows для отображения текста использует системный шрифт, соответствующий ANSI набору символов. Однако вы можете изменить шрифт для отображения информации, используя сообщение **WM_SETFONT** для данного окна.

8.2.10.1. Стили окна редактирования

Windows поддерживает два типа окон редактирования: *обычный* или однострочковый редактор и *многострочное окно* редактирования. При создании окна редактирования с помощью функции **CreateWindow()** можно использовать следующие дополнительные стили. Горизонтальная черта отмечает стили, относящиеся только к многострочным редакторам.

ES_AUTOHSCROLL	Выполняется автоматическая горизонтальная прокрутка текста, при этом полоса прокрутки не появляется.
ES_LOWERCASE	Переводит все вводимые символы в нижний регистр.
ES_UPPERCASE	Переводит все вводимые символы в верхний регистр.
ES_OEMCONVERT	Проводится конвертирование вводимого текста в ASCII символы.
ES_NUMBER	Разрешен ввод *только* цифр.
ES_READONLY	Ввод информации невозможен.
ES_PASSWORD	Все вводимые символы закрываются символом-заполнителем. По умолчанию это символ ‘*’, так что вводимая строка будет выглядеть как “****”. При желании можно переопределить его собственным заполнителем.
ES_NOHIDESEL	Не убирает выделение даже при потере фокуса ввода.
ES_MULTILINE	Создается многострочное окно редактирования.
ES_AUTOVSCROLL	Выполняется автоматическая вертикальная прокрутка текста в окне редактора.
ES_WANTRETURN	Нажатие клавиши Enter трактуется как переход на новую строку в окне редактора.

8.2.10.2. Сообщения от редактора к родительскому окну

Ниже указаны сообщения, которые редактор посылает своему родительскому окну. Часть из них является самостоятельными Windows сообщениями, а другие приходят в составе сообщения **WM_COMMAND**. В последнем

случае старшее слово параметра **wParam** соответствует указанному нотификационному значению:

EN_UPDATE (WM_COMMAND)	Пользователь изменил текст в окне редактирования, но изменения еще не отображены.
EN_CHANGE (WM_COMMAND)	Изменения вступили в силу и уже отображены.
EN_SETFOCUS, EN_KILLFOCUS (WM_COMMAND)	Редактор получает или теряет фокус ввода.
EN_MAXTEXT (WM_COMMAND)	Вводимый символ превышает допустимое количество. Смотри сообщение **EM_SETLIMITTEXT** в следующем разделе.
WM_CTLCOLOREDIT	Посылается родительскому окну, когда редактору требуется перерисовать свой фон. Родительское окно должно вернуть дескриптор кисти для закрашивания фона.

8.2.10.3. Сообщения от родительского окна к редактору

В первую очередь укажем две универсальные функции, которые позволяют изменить заголовок окна с дескриптором **hWnd**

```
SetWindowText(hWnd, szString);
```

и получить текст заголовка окна **hWnd**

```
GetWindowText(hWnd, szString, nString);
```

Параметр **szString** для функции **SetWindowText()** представляет собой константный указатель на строку символов, а для функции **GetWindowText()** – указатель на буфер длиной **nString** символов.

Применительно к окну редактирования эти функции позволяют изменить или получить содержимое редактора.

Ниже представлены основные сообщения, с помощью которых родительское окно может управлять редактором. Напомним, что для отправки этих сообщений используется **SendDlgItemMessage()**, следовательно, редактор может возвращать информацию родительскому окну. Горизонтальная черта отмечает сообщения, относящиеся только к многострочным редакторам.

WM_SETFONT	Установить новый шрифт для элемента управления. Через **wParam** передается дескриптор нового фонта, а **lParam=1**, что требует перерисовки элемента.

EM_SETLIMITTEXT	Ограничивает длину текста, вводимого в окно редактора, величиной **nMax** символов. Параметры сообщения: **wParam = nMax; lParam = 0;**
EM_GETLIMITTEXT	Возвращает текущую длину ограничения текста. Параметры сообщения: **wParam = 0; lParam = 0;**
EM_GETSEL	Получить выделенный пользователем текст в окне редактора. Определите две переменные типа **DWORD dwStart** как позиция начала выделения и **dwEnd** как конечная позиция. Передайте адреса этих переменных через параметры сообщения: **wParam = (WPARAM)&dwStart; lParam = (LPARAM)&dwEnd;** После возврата из сообщения, переменные будут содержать требуемую информацию.
EM_SETSEL	Установить выделение текста. Параметры сообщения: **wParam=nStart; lParam=nEnd;** где **nStart**, **nEnd** - начальная и конечная позиции выделения. Если задать (0) и (-1) соответственно, то выделяется весь текст в редакторе. Если начальная позиция равна (-1), то выделение снимается.
EM_SETREADONLY	Изменить стиль “только для чтения”. Параметры сообщения: **wParam=0** или **1**; **lParam=0;** Если **wParam=1** стиль устанавливается, если **wParam=0** стиль снимается.
EM_GETPASSWORDCHAR	Возвращает текущий символ-заполнитель для стиля **ES_PASSWORD**. Параметры сообщения: **wParam=0; lParam=0;**
EM_SETPASSWORDCHAR	Установить символ-заполнитель для стиля **ES_PASSWORD**. Параметры сообщения: **wParam=ch; lParam=0;** где **ch** - новый символ-заполнитель.
EM_GETLINECOUNT	Возвращает количество строк в многострочном редакторе. Параметры сообщения: **wParam=0; lParam=0;**

EM_LINELENGTH	Возвращает длину для **i**-той строки. Параметры сообщения: **wParam=i; lParam=0;**
EM_GETLINE	Скопировать **i**-ю строку редактора в буфер. Параметры сообщения: **wParam=i; lParam=szBuff;** где **szBuff** адрес буфера копирования. Возвращает длину **i**-той строки.

Кроме того, для окон редактирования определены сообщения **WM_COPY**, **WM_CUT**, **WM_PASTE**, **WM_UNDO**, которые сопровождают манипуляции с буфером обмена. Обработчики по умолчанию для этих сообщений обеспечивают полноценную логику операций с буфером, так что в приложениях эти сообщения обрабатываются очень редко.

8.2.11. Стандартный элемент управления статический текст

Задавая класс “static”, приложение определяет простое прямоугольное дочернее окно, которое не имеет фокус ввода. Этот орган управления обычно используется для создания текстовых меток, может применяться в качестве контейнера для отображения иконок – обычно в диалогах, а также как разделитель других органов управления.

8.2.11.1. Стили элемент управления STATIC

При создании статического текста функцией **CreateWindow()** используются следующие дополнительные стили.

SS_SIMPLE	Простой текст с выравниванием влево.
SS_LEFT **SS_RIGHT** **SS_CENTER**	Задает режим выравнивания текста внутри прямоугольной области.
SS_NOTIFY	Уведомляет родительское окно о нажатиях клавиш мыши внутри окна.
SS_BITMAP **SS_ICON**	Статический элемент представляет собой обрамление для графического образа, загружаемого из ресурсов приложения.
SS_BLACKFRAME **SS_BLACKRECT**	Черная рамка и черный прямоугольник.
SS_WHITEFRAME **SS_WHITERECT**	Белая рамка и белый прямоугольник.

8.2.11.2. Сообщения от статического элемента управления

Сообщения, которые статический элемент управления посылает своему родительскому окну, немногочисленны. Часть из них является самостоя-

тельными Windows сообщениями, а другие приходят в составе сообщения **WM_COMMAND**. В последнем случае старшее слово параметра **wParam** соответствует указанному нотификационному значению.

STN_CLICKED (WM_COMMAND)	Пользователь щелкнул клавишей мыши в окне элемента управления. Только при наличии стиля **SS_NOTIFY**
STN_DBLCLK (WM_COMMAND)	Пользователь сделал двойной щелчок клавишей мыши в окне элемента управления. Только при наличии стиля **SS_NOTIFY**
WM_CTLCOLORSTATIC	Посылается родительскому окну, когда элементу требуется перерисовать свой фон. Родительское окно должно вернуть дескриптор кисти для закрашивания фона.

8.2.11.3. Сообщения от родительского окна к STATIC

Основные операции со стороны родительского окна заключаются в изменении текста этого элемента управления. Мы уже рассмотрели их в пункте 8.2.10.3.

8.2.12. Стандартный элемент управления список

Задавая класс “listbox”, приложение, в простейшем случае, определяет прямоугольное дочернее окно в виде списка текстовых строк. Пользователь может выбирать одну или несколько строк из списка. Элемент управления автоматически добавляет и убирает полосы прокрутки, если полный список не вмещается в клиентскую область дочернего окна.

> Для Windows 95/98 максимальное количество строк в списке ограничено значением 32К.

8.2.12.1. Стили элемента управления список

При создании списка функцией **CreateWindow()** используются следующие основные стили.

LBS_STANDARD	Стандартный список. Включает стили **LBS_SORT** и **LBS_NOTIFY**.
LBS_SORT	При добавлении строк к списку производится их автоматическая сортировка.
LBS_NOTIFY	Список генерирует сообщения для родительского окна о двойном щелчке мышью.

LBS_NOINTEGRALHEIGHT	Возможен частичный показ строк в окне списка. Если стиль не указан, то окно уменьшает свои размеры так, что его высота становится кратной высоте строк.
LBS_MULTICOLUMN	Создается многоколоночный список, при этом все колонки имеют одинаковую ширину..
LBS_MULTIPLESEL	В списке возможен выбор нескольких строк. Если стиль не указан, то только одна строка может быть отмеченной.
LBS_OWNERDRAWFIXED	Список, элементы которого будут отрисовываться родительским окном, а не Windows.

8.2.12.2. Сообщения от списка к родительскому окну

Ниже указаны сообщения, которые список посылает своему родительскому окну. Часть из них является самостоятельными Windows сообщениями, а другие приходят в составе сообщения **WM_COMMAND**. В последнем случае старшее слово параметра **wParam** соответствует указанному нотификационному значению:

LBN_DBLCLK (WM_COMMAND)	Пользователь сделал двойной щелчок клавишей мыши по строке списка.
LBN_SELCHANGE (WM_COMMAND)	Пользователь выбрал строку списка.
LBN_SELCANCEL (WM_COMMAND)	Пользователь отменил выбор строки.
LBN_SETFOCUS LBN_KILLFOCUS (WM_COMMAND)	Список получает или теряет фокус ввода.
WM_CTLCOLORLISTBOX	Посылается родительскому окну, когда списку требуется перерисовать свой фон. Родительское окно должно вернуть дескриптор кисти для закрашивания фона.

8.2.12.3. Сообщения от родительского окна к списку

Ниже представлены основные сообщения, с помощью которых родительское окно может управлять списком. Напомним, что список может возвращать информацию родительскому окну из некоторых сообщений. Горизонтальная черта отмечает сообщения, относящиеся только к списку с множественным выбором, для которого определен стиль **LBS_MULTIPLESEL**. В этом случае понятие “текущее выделение” распространяется на несколько строк списка.

LB_ADDSTRING Добавляет строку **szText** к списку. Если указан стиль **LBS_SORT**, то выполняется сортировка списка, если нет, то строка добавляется в конец списка. Параметры сообщения:
wParam=0; lParam=(LPARAM)szText;
Возвращает позицию новой строки.

LB_INSERTSTRING Вставляет строку **szText** в **i**-тую позицию, при этом сортировка не выполняется, даже если определен стиль **LBS_SORT**. Если **i=-1**, строка добавляется в конец списка. Параметры сообщения:
wParam=i; lParam=(LPARAM)szText;
Возвращает позицию строки.

LB_GETCOUNT Возвращает текущее количество строк в списке. Параметры сообщения:
wParam=0; lParam=0;

LB_DELETESTRING Уничтожает **i**-тую строку в списке.
Параметры сообщения:
wParam=i; lParam=0;
Возвращает количество оставшихся строк.

LB_FINDSTRING Поиск контекста **szFind** в строках списка. Поиск начинается с **i**-той позиции списка.
Параметры сообщения:
wParam=i; lParam=(LPARAM)szFind;
Возвращает номер строки, включающей **szFind**, или **LB_ERR**, если контекст не найден.

LB_GETCURSEL Возвращает номер выделенной пользователем строки. Если список не содержит выделенной строки, возвращается значение **LB_ERR**.
Параметры сообщения:
wParam=0; lParam=0;

LB_SETCURSEL Пометить **i**-тую строку в списке как выделенную.
Параметры сообщения:
wParam=i; lParam=0;

LB_GETTEXTLEN Возвращает длину текста в **i**-той строке списка.
Параметры сообщения:
wParam=i; lParam=0;

LB_GETTEXT	Копирует текст из **i**-той строки списка в буфер **szBuff**. Длина буфера должна быть достаточной, чтобы вместить копируемый текст (см. предыдущее сообщение). Параметры сообщения: **wParam=i; lParam=(LPARAM)szBuff;**
LB_GETSEL	Позволяет определить, выделил ли пользователь **i**-тую строку списка. Параметры сообщения: **wParam=i; lParam=0;** Если возвращается ненулевое значение, строка выделена, если нуль – строка не выделена.
LB_SETSEL	Изменяет выделение для **i**-той строки списка. Параметры сообщения: **wParam=fSel; lParam=i;** Если **fSel** равно 1, **i**-тая строка становится выбранной, если **fSel=0**, то выделение снимается.
LB_SELITEMRANGE	Изменить статус у диапазона строк от **nFist** до **nLast**. Параметры сообщения: **wParam=fSel; lParam=MAKELPARAM(nFist,nLast);**

8.2.13. Стандартный элемент управления список с вводом

Задавая класс “combobox”, приложение определяет прямоугольное дочернее окно, которое, в простейшем случае, содержит список текстовых строк. Однако список снабжен полем ввода, где пользователь может изменить содержимое строки из списка.

Для Windows 95/98 максимальное количество строк в комбинированном списке ограничено значением 32К.

8.2.13.1. Стили элемента управления combobox

При создании комбинированного списка функцией **CreateWindow()** используются следующие основные стили. Можно заметить, что стили для combobox являются комбинацией рассмотренных ранее окна редактирования и элемента список.

CBS_DROPDOWN	Элемент управления состоит из выпадающего списка и окна редактирования.
CBS_DROPDOWNLIST	Окно редактирования отсутствует.
CBS_SIMPLE	Окно редактирования со списком, который виден всегда.

CBS_AUTOHSCROLL	Выполняется автоматическая горизонтальная прокрутка текста в редакторе, при этом полоса прокрутки не появляется.
CBS_SORT	Выполняется сортировка строк списка.
CBS_NOINTEGRALHEIGHT	Возможен частичный показ строк в окне списка. Если стиль не указан, то окно уменьшает свои размеры так, что его высота становится кратной высоте строк.
CBS_LOWERCASE	Переводит все вводимые в окне редактора символы в нижний регистр.
CBS_UPPERCASE	Переводит все вводимые в окне редактора символы в верхний регистр.
CBS_OEMCONVERT	Проводится конвертирование вводимого в окне редактора текста в ASCII символы.
CBS_OWNERDRAWFIXED	Элементы списка будут отрисовываться родительским окном, а не Windows.

8.2.13.2. Сообщения от combobox к родительскому окну

Ниже указаны сообщения, которые комбинированный список посылает своему родительскому окну. Часть из них является самостоятельными Windows сообщениями, а другие приходят в составе сообщения **WM_COMMAND**. В последнем случае старшее слово параметра **wParam** соответствует указанному нотификационному значению:

CBN_DROPDOWN (WM_COMMAND)	Список стал видим.
CBN_CLOSEUP (WM_COMMAND)	Список стал невидим.
CBN_DBLCLK (WM_COMMAND)	Пользователь сделал двойной щелчок клавишей мыши по строке списка.
CBN_SELCHANGE (WM_COMMAND)	Пользователь выбрал строку в списке.
CBN_SELCANCEL (WM_COMMAND)	Пользователь отменил выбор.
CBN_SETFOCUS **CBN_KILLFOCUS (WM_COMMAND)**	Комбинированный список получает или теряет фокус ввода.
CBN_EDITUPDATE (WM_COMMAND)	Пользователь изменил текст в окне редактирования, но изменения еще не отображены.
CBN_EDITCHANGE (WM_COMMAND)	Изменения уже отображены в окне редактирования.

8.2.13.3. Сообщения от родительского окна к combobox

Ниже представлены основные сообщения, с помощью которых родительское окно может управлять элементом combobox. Напомним, что комбинированный список может возвращать информацию родительскому окну из некоторых сообщений. Напомним также, что занести или получить текст в окно редактирования combobox можно с помощью функций Win32 API **SetWindowText()** и **GetWindowText()**.

CB_GETDROPPEDSTATE Получить состояние списка.
Параметры сообщения:
wParam=0; lParam=0;
Если возвращается ненулевое значение, то список видим на экране, если 0, то - невидим.

CB_SHOWDROPDOWN Показать или скрыть список.
Параметры сообщения:
wParam=fShow; lParam=0;
Если **fShow** имеет ненулевое значение, список открывается, если **fShow=0**, список закрывается.

CB_GETCOUNT Возвращает текущее количество строк в списке. Параметры сообщения:
wParam=0; lParam=0;

CB_ADDSTRING Добавляет строку **szText** в список. Если указан стиль **CBS_SORT**, то выполняется сортировка списка, если нет, то строка добавляется в конец списка. Параметры сообщения:
wParam=0; lParam=(LPARAM)szText;
Возвращает позицию новой строки.

CB_INSERTSTRING Вставляет строку **szText** в **i**-тую позицию, при этом сортировка не выполняется, даже если определен стиль **CBS_SORT**. Если **i=-1**, строка добавляется в конец списка. Параметры сообщения:
wParam=i; lParam=(LPARAM)szText;
Возвращает позицию строки.

CB_DELETESTRING Уничтожает **i**-тую строку в списке.
Параметры сообщения:
wParam=i; lParam=0;
Возвращает количество оставшихся строк.

CB_FINDSTRING Поиск контекста **szFind** в строках списка. Поиск начинается с **i**-той позиции списка. Параметры сообщения:
```
wParam=i; lParam=(LPARAM)szFind;
```
Возвращает номер строки, включающей **szFind**, или **CB_ERR**, если контекст не найден.

CB_GETCURSEL Возвращает номер выделенной пользователем строки. Если список не содержит выделенной строки, возвращается значение **CB_ERR**. Параметры сообщения:
```
wParam=0; lParam=0;
```

CB_SETCURSEL Пометить **i**-тую строку в списке как выделенную. Параметры сообщения:
```
wParam=i; lParam=0;
```

CB_GETLBTEXTLEN Возвращает длину текста в **i**-той строке списка. Параметры сообщения:
```
wParam=i; lParam=0;
```

CB_GETLBTEXT Копирует текст из **i**-той строки списка в буфер **szBuff**. Длина буфера должна быть достаточной, чтобы вместить копируемый текст (см. предыдущее сообщение). Параметры сообщения:
```
wParam=i; lParam=(LPARAM)szBuff;
```

9. Создание и использование диалоговых окон

Кроме основного окна приложения, которое имеет строку заголовка, меню и свое собственное содержания, для обмена информацией с пользователем наиболее часто применяются диалоговые окна. С точки зрения пользователя, главная отличительная черта диалоговых окон заключается в том, что основное окно существует на протяжении всей работы приложения, в то время как диалоговые окна по своей природе более непостоянны и требуются только на время обмена данными. Но это не главное различие между основными и диалоговыми окнами. В действительности существуют приложения, использующие диалоговые панели как основные окна, т.е. диалоговые окна могут оставаться видимыми на протяжении всего существования приложения.

Каждое диалоговое окно обычно содержит несколько элементов управления, которые являются дочерними окнами по отношению к диалогу, а диалог, являясь временным окном по отношению к основному окну, позволяет проводить обмен данными между пользователем и приложением. Существует несколько функций Win32 API, которые помогают в конструировании, отображении и управлении содержимым диалоговых окон. Разработчикам приложений обычно не нужно заботиться о рисовании элементов управления или об обработке событий пользовательского интерфейса, достаточно сосредоточиться на самом обмене данными между элементами управления диалогового окна и приложением.

9.1. Этапы создания диалога

9.1.1. Создание шаблона диалога

Для конструирования внешнего вида диалоговой панели обычно используют два способа:

- первый способ размещения элементов управления предполагает использование специального редактора диалогов. Этот редактор позволяет отобразить диалоговую панель и сохранить ее текстовое описание в файле ресурсов приложения. Такой подход при создании диалогов приложений носит зачатки визуального программирования, когда внешний вид и поведение приложения определяется с помощью графических средств проектирования без традиционного программирования на каком-либо алгоритмическом языке;
- второй способ предполагает создание шаблона диалоговой панели в оперативной памяти во время работы приложения. Этот способ более сложен и используется менее часто, обычно только в тех случаях, когда

внешний вид диалоговой панели нельзя определить на этапе создания приложения.

9.1.2. Функция диалога

Диалоговая процедура или функция диалога обрабатывает сообщения, получаемые окном диалога. Она напоминает обычную оконную процедуру, но настоящей оконной процедурой не является. Настоящая оконная процедура панели диалога находится в ядре Windows. Эта оконная процедура вызывает диалоговую процедуру приложения, передавая ей большинство сообщения, которые получает сама.

Простейшая диалоговая процедура DlgProc для модального диалога имеет следующий вид:

```
BOOL CALLBACK DlgProc(HWND hDlg, UINT message,
                      WPARAM wParam, LPARAM lParam)
{
     switch (message) {
     case WM_INITDIALOG:
         // инициализация элементов управления
         return TRUE;
     case WM_COMMAND: {
         int  nID     = LOWORD(wParam);
         int  nEvent  = HIWORD(wParam)
         HWND hWndCtl = (HWND) lParam;
         // обработка сообщений от элементов управления
         return TRUE;
         }
     }
     return FALSE;
}
```

Даже на первый взгляд, у функции диалога и у обычной оконной процедуры имеется много общего, однако видны и отличия. Перечислим и те, и другие.

9.1.2.1. Сходства между диалоговой функцией и оконной процедурой

1. Обе процедуры являются функциями обратного вызова и определяются с макросом CALLBACK.

2. Параметры для диалоговой функции и для обычной оконной процедуры одинаковы.

9.1.2.2. Различия между диалоговой функцией и оконной процедурой

1. Оконная процедура возвращает значение LRESULT, а диалоговая процедура – значение типа BOOL;

2. Если оконная процедура не обрабатывает сообщение, то она вызывает обработчик по умолчанию **DefWindowProc()**. Процедура диалога, если не обрабатывает сообщение, возвращает **FALSE** и **TRUE** – в противном случае.

3. Процедура диалога никогда не получает сообщения **WM_CREATE**, вместо этого вся инициализация выполняется при обработке специального сообщения **WM_INITDIALOG**. Это сообщение является первым, которое получает процедура диалоговой панели. Если процедура диалога возвращает **TRUE**, то Windows помещает фокус ввода на первый элемент управления, который имеет стиль **WS_TABSTOP**. В противном случае, процедура диалога должна использовать функцию **SetFocus()** для того, чтобы самостоятельно установить фокус на один из элементов управления, и тогда она должна возвратить значение **FALSE**.

4. Процедура диалога, как правило, не обрабатывает сообщения **WM_PAINT** и **WM_DESTROY**.

9.2. Типы диалоговых панелей

В Windows существуют два принципиально разных типа диалоговых окон: немодальные и модальные. Рассмотрим особенности модальных диалогов в первую очередь, поскольку они используются более часто.

9.3. Создание модального диалога

При выводе на экран модальной диалоговой панели работа приложения приостанавливается. Функции главного окна приложения и всех его дочерних окон перестают получать сообщения от мыши и клавиатуры. Все эти сообщения попадают в окно диалоговой панели. Когда работа с диалоговой панелью будет завершена, главное окно приложения и его дочерние окна будут вновь разблокированы.

> Диалоговая панель *не должна* создаваться как дочернее окно. В этом случае она будет заблокирована наряду с остальными дочерними окнами. Для создания диалогов используется стиль временного окна.

> Сообщения модального диалога не проходят через очередь сообщений приложения, поэтому клавиши-акселераторы *не влияют* на работу окна диалога.

Модальная диалоговая панель, тем не менее, позволяет пользователю переключиться на работу с другими приложениями. Если требуется запретить такую возможность переключения, это достигается использованием *системных модальных диалоговых панелей*, однако использовать это средство нужно с чрезвычайной осторожностью.

Итак, для создания модального диалога наиболее часто используется функция

```
int DialogBox(HINSTANCE hInst, LPCTSTR pTemplate,
              HWND hWndParent, DLGPROC pDlgFunc)
```

Эта функция создает диалоговое окно, используя ресурс шаблона **pTemplate**, и отображает эту панель как модальное диалоговое окно. Функция **DialogBox()** не возвращает управления до тех пор, пока диалоговое окно не будет закрыто пользователем. В точке вызова **DialogBox()** происходит блокировка окна, владеющего диалогом. Дескриптор этого окна передается через параметр **hWndParent**.

> Не рекомендуется создавать модальные окна без владельца. Windows не разрушает и не скрывает таких диалоговых окон.

9.4. Закрытие модального диалога

Поскольку окно-владелец заблокировано во время работы модального диалога, единственно корректным способом уничтожения диалогового окна является саморазрушение в диалоговой процедуре, что достигается вызовом функции **EndDialog()**.

```
if (nID == IDCANCEL) {
     EndDialog(hDlg, nID);
     return TRUE;
}
```

Эта функция не только закрывает диалоговую панель, но и разблокирует вызывающую функцию **DialogBox()**, которая передает в приложение, в качестве возвращаемого значения, параметр **nID** функции **EndDialog()**. Как правило, диалог завершает работу при нажатии кнопки. Ее идентификатор **nID** и возвращается в представленном выше фрагменте кода. Следовательно, приложение может знать, при нажатии какой кнопки произошло закрытие модального диалога.

Заметим, что идентификаторы **IDOK** и **IDCANCEL**, которые определены в заголовочном файле **windows.h**, занимают особое место для модальных диалогов. Дело в том, что все диалоги снабжается кнопкой “Cancel” для того, чтобы пользователь смог отменить внесенные изменения. Итак, при нажатии на кнопку “Cancel” генерируется сообщение **WM_COMMAND** с идентификатором **IDCANCEL**, которое и поступает в функцию диалога. Эта же ситуация возникает тогда, когда пользователь нажмет клавишу **Esc**.

Все диалоги снабжается также кнопкой “Ok” для того, чтобы пользователь смог принять внесенные изменения. При нажатии на эту кнопку генерируется сообщение **WM_COMMAND** с идентификатором **IDOK**, которое по-

ступает в функцию диалога. Обработчик этого сообщения должен иметь вид:

```
if (nID == IDOK) {
  // получить значения из элементов управления
  EndDialog(hDlg, nID);
  return TRUE;
}
```

Кроме того, сообщение **WM_COMMAND** с идентификатором **IDOK** поступает и в том случае, если пользователь нажимает клавишу **Enter** в тот момент, когда выполняются одновременно два условия: 1) ни одна из кнопок, расположенных в диалоговой панели, не имеет фокус ввода; 2) ни одна из кнопок не имеет стиль **WS_DEFPUSHBUTTON**.

9.5. Окна сообщений.

Окна сообщений являются специальными видом модального диалога. Они предназначены только для отображения коротких информационных сообщений, поэтому могут иметь только заголовок, текст сообщения, одну из предопределенных пиктограмм и одну или несколько кнопок. Например, окно сообщения можно использовать для уведомления пользователя о возникновении какой-либо ошибки или запроса на подтверждение необратимого действия.

Окно сообщения, с заголовком **szCaption** и выводимым текстом **szText**, создается с помощью функции **MessageBox()**.

```
int MessageBox(HWND hWnd, LPCTSTR szText,
               LPCTSTR szCaption, UINT uType)
```

Как правило, параметром **hWnd** является дескриптор того окна – обычного или диалогового – которое создает окно сообщения. Если окно сообщения закрывается, то фокус ввода передается окну **hWnd**. Если дескриптор окна недоступен, или если приложению не нужно, чтобы фокус ввода получило одно из окон приложения, вместо этого дескриптора можно использовать значение **NULL**.

Четвертый параметр **uType** функции **MessageBox()** представляет собой набор битовых флагов.

Первая группа флагов указывает, какие кнопки будут отображены в окне сообщения, например, **MB_OK**, **MB_OKCANCEL**, **MB_YESNO**. Заметим, что окно сообщения может использовать максимум четыре кнопки.

Вторая группа флагов задает то, какая из этих четырех кнопок получит по умолчанию фокус ввода, например, **MB_DEFBUTTONx**, где **x** = 1..4.

Третья группа задает пиктограмму, которая появится в окне сообщения, например, **MB_ICONINFIRMATION** – информационный значок, **MB_ICONWARNING** – значок напоминания, **MB_ICONSTOP** – значок крити-

ческой ошибки, **MB_ICONQUESTION** – значок запроса. Пиктограмма по умолчанию не задается.

В зависимости от нажатой кнопки, приводящей к удалению панели, окно сообщений возвращает один из следующих идентификаторов: **IDOK**, **IDCANCEL**, **IDYES**, **IDNO**, **IDRETRY**, **IDIGNORE**, **IDABORT**.

9.6. Немодальные диалоги

Создание немодальной диалоговой панели не останавливает выполнение приложения. Более того, немодальные диалоговые окна всегда остаются над поверхностью окна владельца, даже если это окно владелец получает фокус ввода. Немодальные окна используются значительно реже модальных диалогов в случаях, когда пользователю необходимо одновременно работать как с окном диалога, так и с основным окном приложения.

Перечислим характерные отличия немодальных диалогов.

1. При использовании немодальной диалоговой панели все сообщения для нее поступают в очередь сообщений приложения. Следовательно, очередь сообщений теперь содержит как сообщения для главного окна, так и для немодального диалога. Для обеспечения корректной реакции диалогового окна на события клавиатуры необходимо изменить цикл обработки сообщений, чтобы он выглядел следующим образом:

```
while(GetMessage(&msg, NULL, 0, 0)) {
   if(!IsDialogMessage(hDlgModaless,&msg)) {
       TranslateMessage(&msg);
       DispatchMessage(&msg);
   }
}
```

Если сообщение предназначено для немодального диалога с дескриптором **hDlgModaless**, то функция **IsDialogMessage()** отправляет это сообщение оконной процедуре окна диалога и возвращает **TRUE**, в противном случае – она возвращает **FALSE**. Следовательно, функции цикла обработки сообщений **TranslateMessage()** и **DispatchMessage()** будут вызываться только, если текущее сообщение **msg** *не предназначено* для окна диалога.

2. Дескриптор немодального диалогового окна должен быть объявлен как глобальная переменная, поскольку будет использоваться в разных функциях приложения.

3. Немодальные диалоговые окна создаются (но не отображаются!) с помощью функции **CreateDialog()**

```
hDlgModaless = CreateDialog(HINSTANCE hInst,
             LPCTSTR lpTempl, HWND hWndParent,
             DLGPROC lpDlgFunc)
```

Параметры функций **CreateDialog()** и **DialogBox()** совпадают, однако **CreateDialog()** возвращает дескриптор окна и не вызывает блокировку родительского окна **hWndParent**.

4. Для отображения немодальной диалоговой панели используется уже знакомая вам пара функций **ShowWindow()** и **UpdateWindow()**.

5. Закрытие немодального диалога производится вызовом функции **DestroyWindow()**, а не **EndDialog()**. Обработчик нажатий кнопок оконной процедуры диалоговой панели может выглядеть следующим образом:

```
if (nID == IDOK || nID == IDCANCEL) {
     DestroyWindow(hDlgModaless);
     hDlgModaless = NULL;
     return TRUE;
}
```

6. Обнуление дескриптора **hDlgModaless** – это программистский прием при работе с немодальным диалогом. Дело в том, что дескриптор может использоваться в разных частях приложения. Следовательно, нужен надежный критерий для проверки существования немодального окна диалога в каждый конкретный момент времени. Нулевой дескриптор и означает отсутствие диалога на экране.

> Отмеченные особенности позволяют провести разрушение немодального диалога вызовом функции **DestroyWindow()** в любом месте приложения, а не только в оконной процедуре диалоговой панели.

9.7. Диалоговые окна общего пользования

В Win32 API реализован ряд часто употребляемых диалоговых окон, что освобождает разработчика от необходимости их повторной реализации для каждого своего приложения. К этим диалоговым окнам, которые носят название общего использования, относятся следующие диалоги.

Назначение диалога	Вызывающая функция
Получить имя открываемого файла	**BOOL GetOpenFileName (OPENFILENAME* pOfn)**
Получить имя для сохранения файла	**BOOL GetSaveFileName (OPENFILENAME* pOfn)**
Выбрать цвет	**BOOL ChooseColor (CHOOSECOLOR* pColor)**

Назначение диалога	Вызывающая функция
Выбрать шрифт	**BOOL ChooseFont (CHOOSEFONT* pCfont)**
Получить параметры для печати документа	**BOOL PrintDlg (PRINTDLG* pPD)**
Получить параметры для настройки страницы при печати	**BOOL PageSetupDlg (PAGESETUPDLG* pPs)**
Получить текст для поиска	**HWND FindText (FINDREPLACE* pFr)**
Получить текст для замены	**HWND ReplaceText (FINDREPLACE* pFr)**

Вызывающие функции двух последних диалогов возвращают дескриптор окна, а не **BOOL** переменную, как все остальные. Легко догадаться, что эти последние диалоги являются немодальными.

Общие диалоговые окна можно использовать двумя способами.

1. Приложения могут использовать эти диалоги в их стандартном виде, вызывая соответствующую функцию общего диалога как обычную Win32 функцию.

2. Расширенный способ позволяет настраивать общие диалоговые окна, в соответствии с целями приложения, например, заменяя специальным образом шаблоны диалоговых панелей.

В заключение приведем пример программного кода, который дает возможность пользователю выбрать имя файла для сохранения некоторой информации. В этом фрагменте **hWnd** - дескриптор главного окна, **g_hInst** - дескриптор копии приложения, **ofn** - структура **OPENFILENAME**, поля которой используются для настройки диалоговой панели.

```
TCHAR szFullPath[MAX_PATH];
TCHAR szFileName[MAX_PATH];
*szFullPath = 0;
*szFileName = 0;
OPENFILENAME ofn;
// проводим "обнуление" всей структуры
ZeroMemory(&ofn, sizeof(ofn));
ofn.lStructSize = sizeof(ofn);
ofn.hwndOwner   = hWnd;
ofn.hInstance   = g_hInst;
// фильтр для отображения типов файлов
ofn.lpstrFilter = "Текстовые файлы\0*.txt\0
                   Все файлы\0*.*\0\0";
ofn.nFilterIndex= 1;
ofn.lpstrFile   = szFullPath;
```

```
ofn.nMaxFile    = sizeof(szFullPath);
ofn.lpstrFileTitle= szFileName;
ofn.nMaxFileTitle = sizeof(szFileName);
// используем текущий катаог
ofn.lpstrInitialDir=NULL;
// определяем заголовок диалога
ofn.lpstrTitle  = "Сохранить в файле";
// настраиваем флаги диалога
ofn.Flags = OFN_PATHMUSTEXIST|OFN_OVERWRITEPROMPT|
            OFN_HIDEREADONLY |OFN_EXPLORER;
// отображаем диалог
if (GetSaveFileName(&ofn)) {
  // в этой точке имеем:
  // ofn.lpstrFile - полный путь файла
  // ofn.lpstrFileTitle - имя выбранного файла
  // делаем запись содержимого файла
}
```

10. Управление файлами

Прежде чем проводить файловые операции, приложению может потребоваться общая информация о магнитном носителе или CD устройстве. Win32 API включает несколько функций такого назначения. Здесь будут указаны лишь основные.

Если требуется узнать тип устройства, нужно обратиться к функции

```
UINT GetDriveType (LPCTSTR szRoot)
```

которая принимает наименование раздела. Напомним, что имя раздела изменяется от “A:” до “Z:”.

Функция возвращает одно из следующих значений:

DRIVE_FIXED	Жесткий диск.
DRIVE_REMOTE	Сетевой диск.
DRIVE_CDROM	CD-ROM привод.
DRIVE_REMOVABLE	Извлекаемый диск.
DRIVE_RAMDISK	RAM диск.
DRIVE_UNKNOWN	Носитель неизвестного типа.
DRIVE_NO_ROOT_DIR	Указанный раздел отсутствует.

Следующая функция

```
BOOL GetVolumeInformation (
      LPCTSTR szRoot,
      LPTSTR  szVolName, DWORD nVolName,
      LPDWORD pSerialNumber,
      LPDWORD pMaxFileLength,
      LPDWORD pFileSysFlags,
      LPTSTR  szFileSysName, DWORD nFileSysName)
```

дает возможность получить некоторую информацию о разделе **szRoot**:

szVolName	метка тома, может быть пустой строкой;
pSerialNumber	номер, присваиваемый при форматировании раздела;
pMaxFileLength	максимальная длина полного имени файла;
pFileSysFlags	сведения о разделе в виде набора битовых флагов;
szFileSysName	имя файловой системы, например, “FAT32”.

Узнать общее и свободное пространство на диске **szRoot** можно аналогично приведенному фрагменту:

```
DWORD SectorsPerCluster = 0,
      BytesPerSector = 0,
      NumberOfFreeClusters = 0,
      TotalNumberOfClusters = 0;
```

```
      GetDiskFreeSpace(szRoot,
            &SectorsPerCluster,
            &BytesPerSector,
            &NumberOfFreeClusters,
            &TotalNumberOfClusters);

// общий объем в МБ
DWORD lenTotal =    TotalNumberOfClusters*
                    SectorsPerCluster*
                    BytesPerSector;
      lenTotal >>= 20;

// объем свободного пространства в МБ
DWORD lenFree =     NumberOfFreeClusters*
                    SectorsPerCluster*
                    BytesPerSector;
      lenFree >>= 20;
```

Следует заметить, что данная функция дает неверные результаты для дисков, объем которых превышает 2 ГБ, поэтому использование ее расширения **GetDiskFreeSpaceEx()** предпочтительнее, поскольку она корректно работает и с дисками большого объема.

Для ранних версий Visual C++ (4 и 5) приложению придется осуществить прямое обращение к библиотеке KERNEL32.DLL для вызова этой функции.

10.1. Доступ к файловой системе

Win32 API поддерживает два принципиально разных подхода при обмене информацией с файловой системой:

- потоковый ввод/вывод, пришедший из MS DOS;
- функции ядра Win32.

10.2. Потоковый ввод/вывод

Большинство программ на языке C, разработанных для MS DOS, использовало потоковый ввод/вывод, который базируется на структуре **FILE** и семействе функций с ней связанных. Напомним общую схему работы с файлом при использовании потокового ввода/вывода.

1. Файл открывается вызовом функции **fopen()**, которая возвращает указатель на структуру **FILE** либо в текстовом, либо в двоичном режиме. Если возникли ошибки при открытии файла, то этот указатель будет равен **NULL**.

2. Допустимы следующие операции с текстовым файлом: **fgets()** - чтение строки (до “\r\n”); **fputs()** - запись строки; **fprintf()** - форматированный вывод. Для бинарного файла допустимы функции **fread()**, **fwrite()**, **fseek()** и **ftell()**.

3. Файл обязательно нужно закрыть функцией **fclose()**.

Эти функции обновлены для 32-х разрядной версии Windows. Это означает, что можно читать и записывать файл большими блоками информации за один прием, используя однократный вызов функций **fread()** и **fwrite()**. Таким образом, отпадает необходимость в использовании циклов при работе с файлами большого размера.

Однако имеются три ограничения:

1) потоковые функции работают только для файлов длиной менее 2 Гб;

2) эти функции не поддерживают разделенный доступ к файлу;

3) работа с файлом должна проводиться в одном обработчике Windows сообщения, т.е. указатель **FILE** на открытый файл не может передаваться как статическая переменная от одного участка кода к другому.

10.3. Функции ядра Windows для работы с файлами

В Win32 API появились принципиально новые 32-х разрядные функции. Именно они рекомендуются для разработчиков, поскольку дают несомненные преимущества от использования Win32 API в сравнении с потоковым вводом/выводом:

- высокая скорость дискового обмена данными;
- произвольная длина файла, точнее обрабатываются файлы с длиной до 2^{64} байт;
- обеспечивается переносимость программного кода на другие платформы (PowerPC, Alpha и MIPS).

Кроме того, Win32 API функции позволяют работать со следующими Windows объектами: 1) дисковые файлы; 2) каналы (pipes); 3) почтовые слоты (mailslots); 4) коммуникационные устройства (модемы); 5) консоли (consoles).

Второй особенностью Win32 функций при работе с файлами является возможность организации асинхронных операций чтения-записи. Это означает, что запустить, к примеру, процесс чтения информации можно в одном обработчике, а контролировать его окончание – в другом.

1. Стартовой функцией является **CreateFile()**, которая позволяет создать или открыть перечисленные выше объекты.

```
HANDLE CreateFile (
       LPCTSTR szFileName,
       DWORD   dwAccess,
       DWORD   dwShareMode,
       SECURITY_ATTRIBUTES* pSecurityAttr,
       DWORD   dwCreation,
       DWORD   dwFlags,
       HANDLE  hTemplateFile)
```

Здесь:

szFileName	имя Windows объекта. Имя может содержать пробелы;
dwAccess	режим открытия;
dwShareMode	режим доступа;
pSecurityAttr	указатель на структуру SECURITY_ATTRIBUTES. Обычно равен NULL;
dwCreation	режим создания;
dwFlags	флаги атрибутов. Особо следует упомянуть о флаге **FILE_FLAG_OVERLAPPED**, который задает асинхронные операции с файлом;
hTemplateFile	дескриптор ранее открытого файла, атрибуты которого берутся за образец при создании нового файла.

Функция возвращает дескриптор файла или мнемоническую константу **INVALID_HANDLE_VALUE** в случае ошибки.

Приведем типичные примеры.

Открыть существующий файл на чтение можно следующим образом:

```
HANDLE hFile = CreateFile ("Just File.txt",
       GENERIC_READ,
       FILE_SHARE_READ, // разделение на чтение
       NULL,
       OPEN_EXISTING,
       FILE_ATTRIBUTE_NORMAL,
       NULL);
```

Создать новый или перезаписать существующий файл в монопольном режиме можно так:

```
HANDLE hFile = CreateFile ("New_Just_File.txt",
       GENERIC_WRITE,
       0,              // монопольный режим
       NULL,
       CREATE_ALWAYS, // перезапись существующего
       FILE_ATTRIBUTE_NORMAL,
       NULL);
```

2. После окончания работы с объектом, открытым с помощью функции **CreateFile()**, его *обязательно* нужно закрыть для освобождения внутренних ресурсов

```
CloseHandle(hFile);
```

3. Получить размер файла **hFile** произвольной длины можно через функцию

```
DWORD GetFileSize (HANDLE hFile, LPDWORD pSizeHigh),
```

которая возвращает младшие четыре байта длины, а параметр **pSizeHigh** указывает на переменную, содержащую старшие четыре байта. Таким образом, результирующее значение длины файла составляет восемь байт.

4. Прочитать **nBytes** байт в буфер **pBuffer** можно через вызов:

```
BOOL ReadFile (HANDLE hFile,
       LPVOID  pBuffer,
       DWORD   nBytes,
       LPDWORD pBytesReaded,
       OVERLAPPED* pOverlapped)
```

при этом в переменную, адрес которой передается через **pBytesReaded**, будет записано реальное количество прочитанных байт. Если указатель **pOverlapped** равен NULL, то выполняется синхронное чтение информации. Это означает, что возврат из функции произойдет *только* после того, как будет выполнено чтение указанных **nBytes** байт или возникнет ошибка. Если параметр **pOverlapped** указывает на структуру **OVERLAPPED**, выполняется асинхронное чтение, когда **ReadFile()** немедленно возвращает управление, а для установления момента окончания чтения нужно предусмотреть специальный код.

5. Записать **nBytes** из буфера **pBuffer** в файл можно через вызов:

```
BOOL WriteFile(HANDLE hFile,
       LPCVOID pBuffer,
       DWORD   nBytes,
       LPDWORD pBytesWritten,
       OVERLAPPED* pOverlapped)
```

Функция возвращает **TRUE**, если операция выполнена успешно. Остальные параметры соответствуют функции **ReadFile()**.

6. Для перемещения файлового указателя используется вызов:

```
DWORD SetFilePointer (HANDLE hFile,
       LONG  Len,
       PLONG pLenHigh,
       DWORD dwMethod)
```

Новое положение файлового указателя задается двумя переменными типа **LONG** – **Len** (младшее двойное слово) и **pLenHigh** (старшее двойное слово). Параметр **dwMethod** задает направление перемещения и принимает три значения: **FILE_BEGIN**, **FILE_CURRENT** и **FILE_END**.

Функция возвращает младшее двойное слово нового положения файлового указателя, а старшее двойное слово передается через переменную, адрес которой указан на третьем позиции.

10.4. Специализированные функции для работы с файлами

Одним из компонентов Windows являются так называемые файлы инициализации, имеющие расширение .INI. Это обычные текстовые файлы, однако их отличительная особенность состоит в том, что они имеют специальную структуру:

```
[Section]
Key=Value
```

Количество секций в файле не ограничено, не лимитируется также и количество ключей в каждой секции. Одним из требований является обеспечение уникальности ключей внутри каждой секции.

Чтение значения ключа **szKey** из секции **szSection** файла **szFilePath** проводится функцией

```
GetPrivateProfileString(
      LPCTSTR szSection, LPCTSTR szKey,
      LPCTSTR szDefVal,  LPTSTR  szValue,
      DWORD   nValue,    LPCTSTR szFilePath)
```

если ключ в указанной секции отсутствует, в буфер **szValue** будет записано значение по умолчанию, указанное в **szDefVal**.

Запись нового значения **szValue** ключа **szKey** в секции **szSection** осуществляется функцией

```
WritePrivateProfileString(
      LPCTSTR szSection, LPCTSTR szKey,
      LPCTSTR szValue,   LPCTSTR szFilePath);
```

Как правило, *.INI файлы расположены в той директории, в которую была установлена операционная система. Если в строке **szFilePath** указано только имя файла, то он будет расположен в этой Windows директории. Если же **szFilePath** содержит полный путь, то Windows директория будет проигнорирована.

Две функции **GetProfileString()** и **WriteProfileString()** специально предназначены для работы с Win.ini файлом.

11. Печать документов

В прежние времена разработчикам приходилось много времени тратить на то, чтобы предусмотреть возможность печати различных документов на всевозможных принтерах. Разные типы принтеров, требовали тонкой и специфической настройки, поскольку разные модели использовали различную систему команд. Теперь эта забота входит в зону ответственности Windows. Разработчик же должен получить контекст принтера, и с помощью обычных GDI функций “отрисовать” документ в этом контексте. Другими словами вы имеете “страницу”, отображаете в любой его части, в любой последовательности то, что нужно, и когда эта “страница” полностью сформирована, отправляете ее на печать. Только в этот момент происходит проигрывание созданного Windows метафайла в контексте устройства принтера, а не тогда когда вы делаете отрисовку.

11.1. Последовательность печати документа

1. Получить или создать контекст устройства принтера.
2. Создать немодальный диалог для отмены печати.
3. Активизировать процесс “спуллинга” печати.
4. “Нарисовать” каждую страницу документа, отправляя ее на печать по мере готовности.
5. Завершить процесс “спуллинга” печати.
6. Закрыть немодальный диалог и освободить контекст устройства принтера.

Итак, разберем содержание каждого шага. При этом нужно иметь в виду, что для реализации данной задачи потребуются две глобальные переменные

```
BOOL g_UserBreak = FALSE;
HWND hDlgBreak;
```

Первая – выполняет роль флага, сигнализирующего о том, что пользователь решил отменить печать документа, нажав кнопку отмены в немодальном диалоге, имеющем дескриптор **`hDlgBreak`**.

11.2. Контекст устройства принтера

Наиболее простой путь получения дескриптора контекста устройства принтера – это использование стандартного Windows диалога. Программный код может выглядеть примерно так:

```
// дескриптор контекста принтера
HDC PrnDC;
// структура инициализации диалога
PRINTDLG pd;
pd.Flags = PD_RETURNDC;
```

```
if (PrintDlg(&pd)) {
      // сохранить дескриптор контекста принтера
      PrnDC = pd.hDC;
}
```

Второй путь связан с созданием дескриптора контекста принтера с помощью вызова функции **CreateDC()**

```
HDC CreateDC(LPCTSTR szDriver, LPCTSTR  szDevice,
             LPCTSTR szOutput, DEVMODE* pInitData),
```

которая возвращает искомый контекст при задании следующих параметров:
szDriver - имя драйвера, обслуживающего принтер, например, “escp2ms”;
szDevice - имя принтерного устройства, например, “Epson LQ-100”;
szOutput - имя порта, к которому подключен принтер, например, “LPT1:”;
pInitData - указатель на структуру **DEVMODE**, поля которой содержат дополнительную информацию о специфической настройке принтера. Например, можно изменить размер бумажного листа, его ориентацию при печати, размер полей и прочее.

В Windows 95 получить сведения обо всех установленных принтерах можно из файла **Win.ini**, который содержит секцию **Devices**. Эта секция выглядит примерно так:

```
[Devices]
Epson LQ-100=ESCP2MS,LPT1:
```

что означает, что в системе имеется один принтер с именем **Epson LQ-100**, который обслуживается драйвером **ESCP2MS** и подключен принтер к порту **LPT1:**.

Для Windows NT информация об установленных принтерах располагается в системном реестре.

11.3. Диалог отмены печати

Шаблон этого диалога *обязательно* должен содержать кнопку с идентификатором **IDCANCEL**, нажатие на которую и будет сигналом об отмене печати документа. Конечно, шаблон может иметь и другие элементы, например, статические поля для вывода имени принтера и количества печатаемых страниц документа. Это не регламентировано и определяется только решаемой задачей.

Напомним, что для создания немодального диалога на родительском окне **hWndParent** следует воспользоваться функцией **CreateDialog()**:

```
hDlgBreak = CreateDialog(g_hInst, pTemplate,
                         hWndParent, DlgFunc);
```

Функция окна, как уже отмечалось, предназначена только для фиксации события нажатия пользователем кнопки отмены печати документа. Исходя из этого, код данной функции может выглядеть примерно так:

```
BOOL CALLBACK DlgProc(HWND hDlg, UINT message,
                      WPARAM wParam, LPARAM lParam)
{
     switch (message) {
     case WM_INITDIALOG:
         // инициализация элементов управления
         return TRUE;
     case WM_COMMAND:
         // только фиксируем нажатие,
         // но не разрушаем диалог!
         if (LOWORD(wParam) == IDCANCEL) {
            g_UserBreak = TRUE;
            return TRUE;
         }
     }
     return FALSE;
}
```

В приложении понадобится еще одна функция обратного вызова, именуемая функцией прекращения печати. Она представляет собой дополнительный цикл обработки сообщений во время работы немодального диалога.

```
BOOL CALLBACK AbortProc(HDC, short)
{
   MSG Msg;
   while (PeekMessage(&Msg, 0, 0, 0, PM_REMOVE)) {
       if (!IsDialogMessage(&Msg)) {
         TranslateMessage(&Msg);
         DispatchMessage(&Msg);
       }
   }
   return !g_UserBreak;
}
```

Особенность этой функция состоит в том, что при возврате значения **FALSE** процесс печати прекращается.

Итак, после создания и отображения немодатьного диалога, приложение должно сделать неактивным родительское окно и подключить функцию прекращения печати к контексту принтера:

```
EnableWindow(hWndParent, FALSE);

SetAbortProc(PrnDC, (ABORTPROC)AbortProc);
```

11.4. Запуск процесса печати

Активизация процесса “спуллинга” печати не вызывает затруднений. Приложение должно заполнить поля структуры **DOCINFO** и обратиться к функции начала печати документа:

```
DOCINFO din;
din.cbSize = sizeof(din);
din.lpszDocName = "MyPrnDoc";
int Error = StartDoc(PrnDC, &din); // good if > 0
```

Этот фрагмент имеет лишь одну особенность, а именно, длина строки поля **din.lpszDocName** не должна превышать 32 символа.

Если возвращаемое значение **Error** больше нуля, что свидетельствует об отсутствии ошибок, то приложение может переходить к следующему этапу.

11.5. Печать страницы документа

Печать на принтере обычно сопровождается сменой режима отображения. Это обусловлено тем, что разрешение принтера в несколько раз превосходит разрешение стандартного дисплея. Следовательно, указанная смена режима существенно повысит качество печатаемого материала. Зачастую это приводит к усложнению алгоритма, так как, отображая документ на экране, обычно используется **MM_TEXT** режим, когда одна логическая единица соответствует одному пикселю, а для контекста принтера применяется, как правило, **MM_HIMETRIC** режим.

Изменить режим отображения для *любого* контекста устройства с дескриптором **hdc** можно с помощью вызова функции:

```
SetMapMode(hdc, nMapMode);
```

Переменная целого типа **nMapMode** указывает новый режим:

Режим отображения	Направление оси Y	Логическая единица
MM_TEXT	вниз	пиксель
MM_LOMETRIC	вверх	0.1 мм
MM_HIMETRIC	вверх	0.01 мм
MM_LOENGLISH	вверх	0.01 дюйма
MM_HIENGLISH	вверх	0.001 дюйма
MM_TWIPS	вверх	1/1440 дюйма

Возвращаясь к схеме печати, следует указать, что для каждой печатаемой страницы приложение выполняет три шага:

- сообщает Windows о начале печати страницы;

- отрисовывает каждую страницу, используя GDI функции;
- сообщает Windows об окончании печати страницы.

Примерный код этого этапа может выглядеть так:

```
do {
       StartPage(PrnDC);
       // отрисовка страницы
       Error = EndPage(PrnDC);
       // счетчик страниц
} while (Error > 0 && !g_UserBreak);
```

Легко заметить, что в цикле проверяется не только наличие ошибки, но и значение флага принудительной остановки процесса печати. Данный цикл может содержать счетчик отпечатанных страниц, величина которого обычно отражается на панели немодального диалога.

11.6. Завершение печати документа

После выхода из цикла печати страниц, приложение должно прекратить процесс печати всего документа с помощью одной из Win32 API функций

```
if (Error <= 0 || g_UserBreak)
    // аварийное завершение
    AbortDoc(PrnDC);
else
    // нормальное окончание
    EndDoc(PrnDC);
```

Следует заметить, что большинство страниц, отправленных приложением на печать, помещается во внутренний кэш, и уже оттуда посылается непосредственно на принтер. Отмена печати приводит к удалению документа из кеша. Если часть страниц уже отправлена на принтер, то, конечно же, отменить их печать из приложения не представляется возможным.

Последним этапом печати документа является освобождение контекста принтера. Однако перед этим приложение должно восстановить статус родительского окна и разрушить немодальный диалог:

```
EnableWindow(hWndParent, TRUE);
DestroyWindow(hDlgBreak);
DeleteDC(PrnDC);
```

На этом процедуру печати документа можно считать завершенной.

12. Процессы и потоки

В операционных системах на базе Win32 организована *вытесняющая многозадачность* (multitasking) – это способность операционной системы выполнять несколько программ одновременно. В основе этого принципа лежит использование операционной системой аппаратного таймера для выделения квантов времени для каждого из одновременно выполняемых процессов. Если эти отрезки времени достаточно малы, и машина не перегружена слишком большим числом программ, то пользователю кажется, что все эти программы выполняются параллельно.

Многозадачность, реализованная в Win16, не была вытесняющей. Она использовала системный таймер для периодического прерывания выполнения одной задачи и запуска другой. Как это было реализовано? Переключение между задачами происходило только в тот момент, когда одна программа завершала обработку сообщения и возвращала управление Windows. Такую невытесняющую многозадачность называют также *кооперативной многозадачностью* потому, что она требует некоторого согласования между приложениями – одна программа могла парализовать работу всей системы, если ей требовалось много времени для обработки сообщения.

32-х разрядные версии Windows кроме многозадачности поддерживают еще и *многопоточность* (multithreading). Многопоточность – это возможность программы самой быть многозадачной. Программа может быть разделена на отдельные потоки выполнения (threads), которые, выполняются параллельно.

Итак, многозадачная операционная система обеспечивает одновременное исполнение двух или более приложений за счет выделения каждой из них процессорного времени. Многопотоковость же подразумевает использование нескольких параллельных потоков вычислений, относящихся к одной прикладной программе.

Таким образом, истинная Win32 программа – это совокупность одного процесса (process) и нескольких потоков (threads). *Процесс* – это исполняемый модуль, которому Windows выделяет память и другие системные ресурсы. *Поток* – это последовательность исполняемых команд. Процесс может состоять из единственного потока, а может содержать их несколько. И на процессы, и на потоки распространяется вытесняющая многозадачность. В результате отпадает необходимость встраивания в каждую прикладную программу механизма уступки управления, как это было в Win16, обеспечивающего выделение процессорного времени другим программам. Вместо этого, в операционных системах Win32, реализован сложный механизм диспетчеризации, обеспечивающий принудительное прерывание, или, другими словами, вытеснение активного потока в тот момент, когда наступает

очередь другого. Это приводит к более четкой обработке многозадачности: при одновременном исполнении нескольких программ процессорное время распределяется между ними более последовательно и улучшается отклик программ на ввод. Чувствительность реакции программы является ее важным показателем, поскольку первое, что подмечает пользователь при неверной работе одной из программ, – это вялая реакция на манипуляции с мышью или клавиатурой.

Заметим, что распределяя процессорное время среди нескольких потоков, операционные системы Windows 95/98 или OS/2 создают иллюзию параллельно протекающих потоков. Только Windows NT при наличии нескольких процессоров действительно обрабатывает эти потоки одновременно, выделяя на обслуживание каждого из них свой собственный процессор. Подобное явление именуется симметричной мультипроцессорной обработкой (SMP).

В Win32 реализация фоновых действий, например, печать документа, не представляет особых трудностей. Создается рабочий поток, единственное назначение которого – это выполнение этих самых фоновых действий. При работе потока операционная система переключается с потока обработки ввода пользователя на фоновый поток и обратно. Более того, такая обработка происходит с максимальной эффективностью, поскольку когда поток для обслуживания ввода находится в ожидании, львиная доля процессорного времени выделяется фоновому потоку. Но, как только пользователь нажимает кнопку на клавиатуре или делает щелчок мышью, обработка фонового потока приостанавливается, и управление передается потоку ввода. Программа реагирует немедленно на все возникающие входящие сообщения.

12.1. Диспетчеризация потоков

Основным фактором, учитываемым при диспетчеризации, служит *приоритет выполнения* каждого потока. В Win32 каждому потоку присваивается свой приоритет - число от 0 до 31; чем больше число, тем выше приоритет. Приоритет 31 резервируется под особенно критичные операции, например, прием данных в реальном режиме времени. Приоритет 0 назначается операционной системой некоторым второстепенным задачам, выполнение которых происходит в то время, когда нет других задач. Большинство потоков работают с приоритетом в диапазоне от 7 до 11. Каждые несколько миллисекунд диспетчер операционной системы просматривает все работающие в системе потоки и передает управление в соответствии со следующими правилами:

1) процессорное время выделяется потоку с наивысшим приоритетом;

2) если потоков с одинаковым приоритетом несколько, то управление передается тому, который простаивает дольше других;

3) потоки с низким приоритетом никогда не вытесняют потоки с более высоким.

Здесь может возникнуть вопрос. Если в системе присутствует поток с приоритетом 10 и с приоритетом 9, то до выполнения потока с приоритетом 9 дело никогда не дойдет? Нет, это не так. Не забывайте, что Windows это событийная операционная система. Когда очередь сообщений потока с приоритетом 10 пуста, он переводится в состояние ожидания, до получения новых сообщений. При этом обрабатываются потоки с более низким приоритетом, а приостановленный поток не получает процессорное время до момента его активизации в ответ на какое-либо событие. Нужно сказать, что большинство потоков проводит длительное время как раз в состоянии ожидания ввода новых сообщений. В результате почти во всех случаях, кроме экстремальных, даже потоки с приоритетом 0 получают достаточно времени.

Отсюда вытекает вывод: не стоит стремиться поднять приоритет потоков собственного приложения с целью их более быстрой работы. Это, наверняка, приведет к обратному эффекту.

> Бездарно написанная программа может замедлить выполнение остальных работающих приложений и всей системы в целом.

12.2. Проблемы многопоточной технологии

Архитектура многопоточного приложения обычно включает главный поток программы, который создается автоматически при запуске исполняемого модуля. Он создает все окна и соответствующие им оконные процедуры, а также обрабатывает все сообщения для этих окон. Следовательно, главный поток должен иметь цикл обработки сообщений.

Все остальные потоки являются рабочими и служат для решения некоторых фоновых задач. Они не имеют оконных процедур и, значит, не обрабатывают сообщения операционной системы. Таким образом, рабочие потоки не выполняют задач, связанных с пользовательским интерфейсом.

Все потоки являются частями одного процесса, поэтому они разделяют все его ресурсы, такие как память, открытые файлы и прочее. Поскольку потоки разделяют память, отведенную программе, то они разделяют и статические переменные приложения. Однако каждый поток имеет свой собственный стек, т.е. автоматические переменные являются уникальными для каждого потока. Каждый поток также имеет свое собственное состояние процессора, которое сохраняется и восстанавливается при переключении между потоками.

Технология программирования для развитого многопоточного приложения – самая сложная задача, с которой приходится сталкиваться Windows программисту. Ее суть состоит в том, что в системе с вытесняющей много-

задачностью поток может быть прерван в любой момент для переключения на другой поток, следовательно, может произойти неконтролируемое приложением взаимодействие между двумя потоками, которое будет нежелательным.

12.3. Создание рабочего потока

В первую очередь нужно написать так называемую “функцию рабочего потока”, тело которой и содержит операции, выполняемые рабочим потоком. Прототип функции одинаков для всех потоков

```
DWORD WINAPI ThreadFunc (LPVOID pData);
```

Имя функции рабочего потока не имеет значения, поскольку, как и для оконной процедуры, Windows использует ее адрес, а не имя. Все функции рабочего потока имеют только один параметр типа **LPVOID**. Следовательно, чтобы передать в функцию несколько параметров из вызывающего процесса, нужно определить собственную структуру, заполнить поля и передать ее адрес через **pData**.

Особенность функции рабочего потока состоит в том, что при выходе из нее *поток автоматически завершается*.

> Используйте именно этот путь для окончания рабочего потока, а не принудительное завершение через специальные функции Win32 API. Это может быть источником трудноуловимых ошибок межпотокового взаимодействия.

Непосредственное создание рабочего потока происходит в момент вызова Win32 API функции **CreateThread()**, которая имеет прототип:

```
HANDLE CreateThread(
      SECURITY_ATTRIBUTES*  pThreadAttr,
      DWORD                 dwStackSize,
      THREAD_START_ROUTINE* pThreadFunc,
      LPVOID                pData,
      DWORD                 dwFlags,
      LPDWORD               pIdThread);
```

Первый параметр обычно не используется и равен **NULL**, второй – определяет размер стека для рабочего потока. Если он равен 0, Windows сама определяет необходимый размер. Третий параметр – это указатель на функцию рабочего потока, а четвертый – указатель блока памяти, который будет передан в **ThreadFunc()**. Флаг активизации рабочего потока обычно равен 0, что означает немедленный запуск потока. Через последний указатель буден возвращен идентификатор созданного потока. Однако больший интерес представляет дескриптор потока, возвращаемый функцией **CreateThread()**. Он позволяет вызывающему процессу проводить некоторые манипуляции с запущенными потоками.

Ниже представлен фрагмент кода, выполняющий запуск рабочего потока:

```
HANDLE       hThread;
DWORD        idThread;
THREAD_PARAM params;
// ....
hThread = ::CreateThread(NULL, 0, ThreadFunc,
                         &params, 0, &idThread);
if (!hThread) {
  // ошибки создания потока
}
```

Здесь **THREAD_PARAM** – некоторая структура, которая определяется в приложении. Она служит для передачи набора параметров в функцию потока.

Этот фрагмент содержит потенциальную ошибку, суть которой объяснена ниже.

При выходе из блока, запустившего поток, происходит разрушение локальных переменных **params** и **hThread**. Но уже запущенный поток не может "знать" об этом, он продолжает пользоваться данными из блока памяти, которого уже нет! Последствия этого – непредсказуемы.

Дескриптор **hThread** необходим для корректного освобождения ресурсов потока после его завершения следующим образом:

```
CloseHandle(hThread);
```

Выходом из сложившейся ситуации является объявление параметров **params** и **hThread** как глобальные или статические переменные.

12.4. Организация взаимодействия потоков

Как уже отмечалось, главная трудность технологии многопотокового программирования – это организация взаимодействия потоков. Приложение должно иметь возможность дуплексного обмена информацией. Это означает, что данные должны передаваться как от рабочего потока к процессу или, другими словами, в главный поток, так и в обратном направлении – от процесса в рабочий поток.

12.4.1. Рабочий поток – процесс

С первым направлением взаимодействия особых трудностей не возникает, поскольку здесь работает посылка сообщений. Напомним, что главный поток имеет функцию окна и, следовательно, может принимать сообщения от рабочего потока. Но какие сообщения? В заголовочном файле **windows.h** определена константа **WM_USER**, которая является верхней границей сообщений, используемых Windows для служебных целей. Это

означает, что все истинные Windows сообщения имеют номера, меньшие чем **WM_USER**. Следовательно, для своих внутренних целей приложение безопасно может использовать сообщения со значением **WM_USER+n**, где **n>0**. При этом операционной системой гарантируется, что это не приведет к нежелательному эффекту коллизии сообщений, т.е. наложению одного на другое.

Как отправлять сообщения – синхронно или асинхронно? Ответ едиственный. Следует использовать *только асинхронную* передачу через функцию **PostMessage()**, подразумевая, что в любой момент работа потока может быть прервана.

12.4.2. Процесс – рабочий поток

Рассмотрим типичные ситуации.

Рабочий поток должен начать обработку некоторых данных только после того, как эти данные будут подготовлены другим потоком. Как уведомить рабочий поток о том, что информация уже готова к обработке?

В силу каких-либо причин, рабочий поток должен немедленно завершить свою работу. Как сообщить ему об этом? Конечно, можно использовать принудительное снятие потока, но это, как правило, приводит к нежелательным побочным эффектам.

Подобные примеры можно продолжать, но уже ясно, что рабочий поток не может принимать сообщения, т.к. он не имеет функции окна.

1. Для координации действий потоков в такого рода ситуациях в Win32 предусмотрены специальные *формы синхронизации*. Одной из таких форм являются события (events). Объект событие может находиться только в двух состояниях: установлено и сброшено. Сам по себе объект событие не представляет интереса, он используется только в комбинации со специальными Win32 *функциями ожидания*.

Дескриптор на объект событие возвращает функция **CreateEvent()**:

```
HANDLE CreateEvent(
       SECURITY_ATTRIBUTES* pEventAttr,
       BOOL bManualReset,
       BOOL bInitialState,
       LPCTSTR lpName); //только для разных процессов
```

Здесь: указатель **pEventAttr**, как правило, равен NULL; переменная **bManualReset** определяет режим управления событием – вручную (**TRUE**) или автоматически (**FALSE**); переменная **bInitialState** задает стартовое состояние объекта – установлено (**TRUE**) или сброшено (**FALSE**).

Событие может быть именованным, тогда параметр **lpName** должен указывать на константную строку. Это имя используется в случае, когда

требуется синхронизировать два или более процессов. Для взаимодействия процесс – рабочий поток данное имя не требуется, поэтому можно использовать **NULL** указатель.

Напомним, что все созданные или открытые Windows объекты требуют закрытия функцией:

```
CloseHandle((HANDLE)hObject);
```

когда отпадает необходимость в их использовании.

2. Второй трудностью межпотокового взаимодействия является доступ к разделяемым данным. Допустим, один поток модифицирует какие-то данные, а в это же время второй поток их читает. Если первый поток не завершил свои изменения, то второй получит неверные значения. Наилучший способ предотвратить такую ситуацию – это использование критических секций (critical sections), которые представляют собой разделы кода, во время выполнения которого текущий поток не может быть прерван, а все остальные переводятся в состояние ожидания.

Критическая секция должна быть проинициализирова, после чего функция

```
EnterCriticalSection(&cs);
```

начинает критическую секцию **cs**, а функция

```
LeaveCriticalSection(&cs);
```

завершает ее. При этом Windows гарантирует, что код программы, заключенный между этими двумя вызовами будет выполняться как бы в монопольном режиме, т.е. тогда, когда остальные потоки данного приложения будут «заморожены».

3. Следует упомянуть о третьей трудности, которая связана с динамическим захватом памяти в рабочем потоке. Эта проблема проявляется тогда, когда, возникает необходимость иметь постоянную область памяти, *уникальную для каждого потока*. Win32 API предоставляет ряд функций, поддерживающих такую специфическую память, которая называется локальной памятью потока (thread local storage или TLS). Однако лучшим решением следует признать пересмотр алгоритма решения задачи, и полный отказ от такого захвата в потоке.

12.5. Общая схема взаимодействия потоков

Рассмотрим общую схему взаимодействия главного и рабочего потоков. Определим структуру **SWT**, которая будет передаваться в функцию рабочего потока как параметр:

```
struct SWT {
      HANDLE hevBeg;  // событие начала
      HANDLE hevEnd;  // событие завершения
      CRITICAL_SECTION cs;
      // дополнительные поля
};
```

Структура, помимо полей, определяемых решаемой задачей, содержит два дескриптора событий и критическую секцию.

В главном потоке объявляются две статические переменные **swt** и **hThread**, а также проводится инициализация критической секции и создание объектов событий для управления рабочим потоком. Оба события используют "ручное" изменение состояния и первоначально установлены в "сброшено".

```
static SWT    swt;
static HANDLE hThread;

swt.hevBeg = CreateEvent(NULL, TRUE, FALSE, NULL);
swt.hevEnd = CreateEvent(NULL, TRUE, FALSE, NULL);
InitializeCriticalSection(&swt.cs);
```

Затем запускается рабочий поток с обслуживающей функцией **ThreadFunc()**, которой в качестве параметра передается указатель на структуру **swt**.

```
hThread = CreateThread(NULL,0,
                  ThreadFunc,&swt,0,&idThread);
```

После этого, возможно в другом обработчике функции главного окна приложения, устанавливается событие **hevBeg**, что является сигналом рабочему потоку к началу исполнения своего кода.

```
SetEvent(swt.hevBeg);  // Событие установлено
```

Теперь рассмотрим функцию рабочего потока

```
DWORD WINAPI ThreadFunc (LPVOID pData)
{
  // преобразуем указатель
  SWT* pswt = (SWT*)pData;
  // бесконечно ожидаем наступления события
/*1*/   WaitForSingleObject(pswt->hevBeg, INFINITE);

  // продолжаем цикл пока не установлено
  // события завершения рабочего потока
/*2*/   while (WaitForSingleObject(pswt->hevEnd, 0)
                                    != WAIT_OBJECT_0) {
      // вводим критическую секцию и ждем когда
      // остальные потоки будут заблокированы
      EnterCriticalSection(&pswt->cs);
```

```
            // некие действия
            // в "монопольном" режиме

            // сбрасываем критическую секцию
            LeaveCriticalSection(&pswt->cs);
        }
        // сбрасываем событие
        ResetEvent(pswt->hEvEnd);
        // выход из функции автоматически
        // завершает рабочий поток
        return 0;
    }
```

В цикле, помеченном через /*1*/, функция рабочего потока будет находиться до тех пор, пока главный поток не установит событие **hevBeg**. Цикл, помеченный /*2*/, организован иным образом. Функция ожидания только проверяет состояние события **hevEnd**. Если оно не установлено, то выполняется тело цикла, в противном случае – цикл заканчивается, что ведет к выходу из функции рабочего потока и завершению самого потока.

Вернемся в главный поток. Для того, чтобы завершить рабочий поток устанавливается событие **hevEnd**, после чего процесс ожидает момента его сброса в рабочем потоке.

```
SetEvent(swt.hevEnd);
while (WaitForSingleObject(swt.hevEnd, 200) ==
                                    WAIT_OBJECT_0);
// в данный момент рабочий поток завершен
```

Затем выполняется закрытие объектов

```
CloseHandle(hThread);
CloseHandle(swt.hevBeg);
CloseHandle(swt.hevEnd);
DeleteCriticalSection(&swt.cs);
```

> Используете функцию ожидания **Sleep()**, только для рабочих потоков. При ее применении в главном потоке он блокируется и не обрабатывает поступающие сообщения.

13. Приложение "Тестер файлов"

В заключение рассмотрим исходный текст приложения, предназначенного для проверки целостности файлов на магнитном носителе, например, на дискете. Конечно, операционная система включает в свой состав служебную программу проверки диска. Однако она выполняет проверку для всей поверхности, что порой требует достаточно много времени. Данная утилита проверяет – путем чтения – только дисковые файлы, значит, в некоторых случаях, может быть более эффективной по затратам компьютерных ресурсов. С другой стороны, автор создавал данное приложение с учебными целями и не претендует, чтобы утилита считалась полностью системной.

Перечислим основные идеи и особенности приложения:

1. Для задания имени драйвера, на котором проводится тестирование файлов, используется командная строка приложения. Так как информация из командной строки потребуется в функции окна, определен указатель, объявленный как глобальная переменная.

2. Функция **`WinMain()`** обычна. Цикл обработки сообщений тоже не содержит каких-либо особенностей.

3. В обработчике **`WM_CREATE`** функции главного окна создается Win32 элемент управления ListView, который позволяет отображать различную информацию в колонках каждой строки. Приложение использует четыре колонки: имя файла, статус проверки, длина файла и время его создания. Здесь же создаются две кнопки для запуска процесса проверки и его остановки.

4. При нажатии кнопки начала проверки или нажатии клавиши "Enter", главный процесс запускает рабочий поток, который выбирает все файлы на указанном разделе и выполняет их проверку обычным чтением. Если встречается директория, то выполняется рекурсивный вызов функции проверки для этой директории. Результаты проверки отображаются в виде новой строки элемента ListView.

5. При нажатии кнопки останова, процесс сбрасывает дескриптор окна в нулевое значение, что является сигналом для рабочего потока, к прекращению своей работы. Одновременно в главном потоке запускается таймер, при обработке которого приложение проверяет, не закончился ли рабочий поток. Аналогичная проверка с использованием таймера выполняется при закрытии приложения.

Проект включает три файла:

`TestDisk.cpp`	исходный код приложения;
`Resource.h`	файл идентификаторов;
`TestDisk.rc`	файл ресурсов приложения.

Создайте файл `Resource.h` и определите в нем следующие идентификаторы (числовые значения могут быть иными):

```
IDS_ERROR            1
IDS_APP_NAME         2
ID_BUTT_BEG        101
ID_BUTT_END        102
ID_LISTVIEW        103
```

Файл `TestDisk.rc` для этого проекта содержит только один ресурс – строковый:

```
STRINGTABLE DISCARDABLE
BEGIN
      IDS_ERROR     "Ошибка"
      IDS_APP_NAME  "Тестер файлов на диске "
      ID_BUTT_BEG   "Проверить"
      ID_BUTT_END   "Прервать"
END
```

Далее приводится полный текст файла `TestDisk.cpp`.

13.1. Функция WinMain()

```
#define  STRICT
#define  WIN32_LEAN_AND_MEAN
#include <Windows.h>
// Заголовочный файл для Win32 элементов управления
#include <CommCtrl.h>

#include "Resource.h"


// Используем два таймера
#define  TIMER_CLOSE  4444
#define  TIMER_WAIT   4445

// Структура для отображения информации
// о тестируемом файле в ListView
struct NEW_FILE_DATA {
      TCHAR  szName [MAX_PATH];
      TCHAR  szLen  [16];
      TCHAR  szDate [32];
      TCHAR  szCheck[16];
};

// Структура для параметров
// функции рабочего потока
struct THREADPARAM {
      HWND   hWnd;
      HWND   hwndList;
      DWORD  idThread;
      TCHAR  szRoot[MAX_PATH];
};
```

```
//////////////////////////////////////////////////////
//     Глабальные переменные
HINSTANCE g_hInst;
LPCSTR    g_szCmd;

//////////////////////////////////////////////////////
//     Прототипы функций
int    InitApplication (LPCTSTR szClass);
void   ViewError       (HWND hWnd);
LPTSTR StrFromRC       (int IDstr);

int WINAPI WinMain (HINSTANCE hInstance,
             HINSTANCE hPrevInstance,
             LPSTR lpCmdLine, int nCmdShow)
{
    // Приложение использует командную строку,
    // через которую можно задать диск для проверки.
    // Например, "TestDisk.exe D:\"
    // В этом случае тестируются файлы на диске D:
    // Если параметр не задан, используется A:\

    // Сохраняем глобальные переменные
    g_hInst = hInstance;
    g_szCmd = lpCmdLine;
    if (!*g_szCmd)
      g_szCmd = TEXT("a:\\");

    // Подключаем библиотеку Win32 стандартных элементов,
    // но испотльзовать будем только ListView

    INITCOMMONCONTROLSEX commItems;
    commItems.dwSize = sizeof(commItems);
    commItems.dwICC  = ICC_LISTVIEW_CLASSES;
    InitCommonControlsEx(&commItems);

    // Регистрируем класса главного окна
    LPCTSTR szClass = TEXT("FileTester32");
    if ( !InitApplication(szClass) ) {
      // Возникла ошибка
      ViewError(NULL);
      return -1;
    }

    // Создание главного окна
    int proc = 20;//%
    int xW  = ::GetSystemMetrics(SM_CXSCREEN);
    int yW  = ::GetSystemMetrics(SM_CYSCREEN);
    int  x  = (proc*xW)/100;
    int  y  = (proc*yW)/100;

    // Шаблон имени окна хранится в ресурсах.
    // Добавляем имя тестируемого диска.
    TCHAR szTitle[128];
    lstrcpy(szTitle, StrFromRC(IDS_APP_NAME));
```

```
    lstrcat(szTitle, g_szCmd);
    HWND hWnd = ::CreateWindowEx(WS_EX_OVERLAPPEDWINDOW,
             szClass, szTitle,
             WS_OVERLAPPEDWINDOW,
             x, y,
             ((100-2*proc)*xW)/100,
             ((100-2*proc)*yW)/100,
             NULL, NULL, g_hInst, NULL);
    if (!hWnd) {
       // Возникла ошибка
       ViewError(NULL);
       return -1;
    }

    ::ShowWindow(hWnd, nCmdShow);
    ::UpdateWindow(hWnd);

    // Цикл обработки сообщений
    MSG msg;
    while (::GetMessage(&msg, NULL, 0, 0)) {
       ::TranslateMessage(&msg);
       ::DispatchMessage(&msg);
    }

    return 0;
}
```

13.2. Функция главного окна

```
// Прототипы функций, используемые в WndProc()
HWND  CreateListView (HWND hwndParent);
void  InitListView   (HWND hwndList);
void  ResizeListView (HWND hwndList, HWND hwndParent);
void  SwitchButton   (HWND hWnd, TCHAR chMode);

DWORD WINAPI ThreadFunc (LPVOID pData);

LRESULT CALLBACK WndProc(HWND hWnd,
        UINT message, WPARAM wParam, LPARAM lParam)
{
    // Используем две статические переменные
    // для дескриптора рабочего потока
    // и структуры параметров для него.
    static  HANDLE       hThread;
    static  THREADPARAM  parm;

    switch (message) {

    case WM_CREATE :
       // Создаем элемент управления ListView.
       parm.hwndList = CreateListView(hWnd);
       if (parm.hwndList) {
           // На одном месте создаем две кнопки
           // с разными идентификаторами.
```

```
        // Первая будет запускать процесс проверки,
        // а вторая - останавливать.
        // Пряча одну и показывая другую,
        // организуем эффект переключения.

        HWND hButt = ::CreateWindow(TEXT("BUTTON"),
                 StrFromRC(ID_BUTT_BEG),
                 WS_CHILD | WS_TABSTOP | WS_VISIBLE,
                 10, 6, 80, 24,
                 hWnd, (HMENU)ID_BUTT_BEG,
                 g_hInst, NULL);
        if (hButt) {
           // Заменяем шрифт для кнопки на тот,
           // который используется для ListView.
           WPARAM pFont = ::SendMessage(parm.hwndList,
                 WM_GETFONT, 0, 0);
           ::SendMessage(hButt, WM_SETFONT, pFont,
                 MAKELPARAM(TRUE, 0));
        }

        hButt = ::CreateWindow(TEXT("BUTTON"),
                 StrFromRC(ID_BUTT_END),
                 WS_CHILD | WS_TABSTOP,
                 10, 6, 80, 24,
                 hWnd, (HMENU)ID_BUTT_END,
                 g_hInst, NULL);

        if (hButt) {
           WPARAM pFont = ::SendMessage(parm.hwndList,
                 WM_GETFONT, 0, 0);
           SendMessage(hButt, WM_SETFONT, pFont,
                 MAKELPARAM(TRUE, 0));
        }
     }
     break;

  case WM_SIZE :
     // Изменяем размеры ListView.
     ResizeListView(parm.hwndList, hWnd);
     return 0;

  case WM_SETFOCUS :
     // Передаем фокус на элемент ListView.
     ::SetFocus(parm.hwndList);
     break;

  case WM_COMMAND :
     ::SetFocus(parm.hwndList);
     if (LOWORD(wParam) == ID_BUTT_BEG) {
         // Запускаем рабочий поток.
         InitListView(parm.hwndList);
         lstrcpy(parm.szRoot, g_szCmd);
         parm.hWnd = hWnd;
         if (hThread) ::CloseHandle(hThread);
```

```
        hThread = ::CreateThread(NULL, 0, ThreadFunc,
                  &parm, 0, &parm.idThread);
        SwitchButton(hWnd, TEXT('E'));
    }
    else if (LOWORD(wParam) == ID_BUTT_END) {
        // Останавливаем рабочий поток
        // и запускаем таймер ожидания.
        parm.hWnd = 0;
        ::SetTimer(hWnd, TIMER_WAIT, 100, NULL);
        SwitchButton(hWnd, TEXT('B'));
    }
    return 0;

case WM_NOTIFY : {
    NMHDR& nm = *(NMHDR*)lParam;
    if (nm.hwndFrom == parm.hwndList &&
        nm.code == LVN_KEYDOWN) {
        // Определяем какая клавиша нажата.
        LV_KEYDOWN& km = *(LV_KEYDOWN*)lParam;
        if (km.wVKey == VK_RETURN) {
            // Нажата клавиша "Enter".
            // Имитируем нажатие соответствующей кнопки.
            ::PostMessage(hWnd, WM_COMMAND,
                  MAKEWPARAM(((parm.idThread) ?
                  ID_BUTT_END : ID_BUTT_BEG),
                  BN_CLICKED), 0);
        }
        else if (km.wVKey == VK_ESCAPE)
            // Нажата клавиша "Esc".
            // Имитируем нажатие соответствующей кнопки.
            ::PostMessage(hWnd, WM_CLOSE, 0, 0);
        return 0;
    }
    break;  }

case WM_TIMER :
    ::KillTimer(hWnd, wParam);
    if (parm.idThread) {
        // Продолжаем ждать окончания рабочего потока.
        ::SetTimer(hWnd, wParam, 100, NULL);
    }
    else {
        // Поток завершен.
        ::CloseHandle(hThread);
        hThread = NULL;
        if (wParam == TIMER_CLOSE)
            ::PostMessage(hWnd, WM_CLOSE, 0, 0);
    }
    return 0;

case WM_CLOSE :
    if (parm.idThread) {
        // Рабочий поток еще активен.
```

```
            // Запускаем таймер ожидания окончания.
            parm.hWnd = 0;
            ::SetTimer(hWnd, TIMER_CLOSE, 100, NULL);
            return 0;
        }
        break;

    case WM_DESTROY :
        if (hThread)
            ::CloseHandle(hThread);
        ::PostQuitMessage(0);
        return 0;
    }

    return ::DefWindowProc(hWnd, message, wParam, lParam);
}
```

13.3. Вспомогательные функции

```
int InitApplication (LPCTSTR szClass)
// Регистрация класса главного окна
{
    WNDCLASS wc;
    ::ZeroMemory(&wc, sizeof(wc));
    wc.style         = 0;
    wc.lpfnWndProc   = WndProc;
    wc.hInstance     = g_hInst;
    wc.hIcon         = ::LoadIcon(NULL, IDI_APPLICATION);
    wc.hCursor       = ::LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = (HBRUSH)(COLOR_BTNFACE+1);
    wc.lpszClassName = szClass;

    return ::RegisterClass(&wc);
}

LPTSTR StrFromRC (int IDstr)
// Загружает строку из ресурсов приложения.
{
    static  TCHAR  szBuff[256];
    ::LoadString(g_hInst, (UINT)IDstr, szBuff,
                                       sizeof(szBuff));
    return szBuff;
}

void ViewError (HWND hWnd)
// Отображает сообщение о любой системной ошибке.
{
    TCHAR szErrs[16*1024];
    ::FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM,
             NULL, ::GetLastError(),
             MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
             szErrs, sizeof(szErrs), NULL);
```

```
    ::MessageBox((hWnd) ? hWnd : GetFocus(),
             szErrs, StrFromRC(IDS_ERROR),
             MB_OK | MB_ICONSTOP);
}

void SwitchButton (HWND hWnd, TCHAR chMode)
// "Переключает" кнопки,
// скрывая одну и показывая другую.
{
    HWND hButBeg = ::GetDlgItem(hWnd, ID_BUTT_BEG);
    HWND hButEnd = ::GetDlgItem(hWnd, ID_BUTT_END);
    if  (!hButBeg || !hButEnd) return;

    if (chMode == TEXT('B')) {
       ::ShowWindow(hButBeg, SW_SHOW);
       ::ShowWindow(hButEnd, SW_HIDE);
    }
    else {
       ::ShowWindow(hButBeg, SW_HIDE);
       ::ShowWindow(hButEnd, SW_SHOW);
    }
}

HWND CreateListView (HWND hwndParent)
// Создает элемент ListView
{
    DWORD dwStyle = WS_CHILD | WS_BORDER | WS_TABSTOP |
             WS_VISIBLE | LVS_REPORT | LVS_NOSORTHEADER;

    HWND hwndList = CreateWindowEx(WS_EX_CLIENTEDGE,
             WC_LISTVIEW, // class name
             TEXT(""), dwStyle,
             0, 0, 0, 0,
             hwndParent,
             (HMENU)ID_LISTVIEW, g_hInst, NULL);
    if(!hwndList)
       return NULL;

    ResizeListView(hwndList, hwndParent);

    // Создаем 4 колонки.
    LV_COLUMN lvColumn;
    int       col = 0;
    lvColumn.mask = LVCF_FMT | LVCF_WIDTH | LVCF_TEXT |
                                            LVCF_SUBITEM;
    lvColumn.fmt  = LVCFMT_LEFT;
    lvColumn.cx = 240;
    lvColumn.pszText = TEXT("Файл");
    ListView_InsertColumn(hwndList, col++, &lvColumn);

    lvColumn.fmt  = LVCFMT_LEFT;
    lvColumn.cx = 60;
    lvColumn.pszText = TEXT("Статус");
    ListView_InsertColumn(hwndList, col++, &lvColumn);
```

```
    lvColumn.fmt  = LVCFMT_RIGHT;
    lvColumn.cx = 80;
    lvColumn.pszText = TEXT("Длина");
    ListView_InsertColumn(hwndList, col++, &lvColumn);

    lvColumn.fmt  = LVCFMT_RIGHT;
    lvColumn.cx = 120;
    lvColumn.pszText = TEXT("Дата");
    ListView_InsertColumn(hwndList, col++, &lvColumn);

    return hwndList;
}

void ResizeListView (HWND hwndList, HWND hwndParent)
// Изменяет размеры элемента ListView
{
    if (!hwndList) return;

    RECT rc;
    int  lr_offs = 8, top_offs = 36;
    ::GetClientRect(hwndParent, &rc);
    ::SetWindowPos (hwndList, NULL,
             lr_offs, top_offs,
             rc.right - rc.left - 2*lr_offs,
             rc.bottom - rc.top - lr_offs - top_offs,
             SWP_NOZORDER);
}

void InitListView (HWND hwndList)
// Инициализация элемента ListView
{
    if (!hwndList) return;

    ListView_DeleteAllItems(hwndList);
    ListView_SetItemCount(hwndList, 256);
    ::UpdateWindow(hwndList);
}

int InsertListRow (HWND hwndList, NEW_FILE_DATA *pData)
// Добавляет одну строку для элемента ListView
{
    if (!hwndList) return -1;

    LVITEM item;
    ::ZeroMemory(&item, sizeof(item));
    item.mask    = LVIF_TEXT;
    item.iItem   = 0;
    item.pszText = pData->szName;

    item.iItem = ListView_InsertItem(hwndList, &item);

    if (item.iItem > -1) {
       item.iSubItem = 1;
       item.pszText  = pData->szCheck;
       ListView_SetItem(hwndList, &item);
```

```
        item.iSubItem = 2;
        item.pszText  = pData->szLen;
        ListView_SetItem(hwndList, &item);

        item.iSubItem = 3;
        item.pszText  = pData->szDate;
        ListView_SetItem(hwndList, &item);

        UINT mask = LVIS_FOCUSED | LVIS_SELECTED;
        ListView_SetItemState (hwndList, -1, 0, mask);
        ListView_SetItemState (hwndList, item.iItem,
                               mask, mask);
        ListView_EnsureVisible(hwndList, item.iItem, FALSE);
        ::UpdateWindow(hwndList);
    }

    return item.iItem;
}

void UpdateCheckRow (HWND hwndList, int nItem,
                     LPTSTR szCheck)
// Обновляет колонку статуса для "nItem"-той строки
{
    if (!hwndList || nItem < 0 ||
       ListView_GetItemCount(hwndList) <= nItem) return;

    LVITEM item;
    ::ZeroMemory(&item, sizeof(item));
    item.mask     = LVIF_TEXT;
    item.iItem    = nItem;
    item.iSubItem = 1;
    item.pszText  = szCheck;
    ListView_SetItem(hwndList, &item);
    ListView_EnsureVisible(hwndList, item.iItem, FALSE);
    ::UpdateWindow(hwndList);
}
```

13.4. Функция рабочего потока

```
DWORD WINAPI ThreadFunc (LPVOID pData)
// Функция рабочего потока.
{
    THREADPARAM& parm = *(THREADPARAM*)pData;
    // Истинная проверка выполняется
    // функцией CheckFiles()
    CheckFiles(parm, parm.szRoot);
    parm.idThread = 0;
    SwitchButton(parm.hWnd, TEXT('B'));
    return 0;
}

void CheckFiles (THREADPARAM& parm, LPCTSTR szRoot)
// Проверка файлов. Используется в потоке.
```

```
{
    WIN32_FIND_DATA df;
    SYSTEMTIME sysTime;
    TCHAR      tt[MAX_PATH];

    lstrcpy(tt, szRoot);
    lstrcat(tt, TEXT("*.*"));
    HANDLE hd = ::FindFirstFile(tt, &df);
    if (hd == INVALID_HANDLE_VALUE) {
      ViewError(parm.hWnd);
      return;
    }

    BOOL rb = TRUE;
    while (rb) {
      // Проверяем сигнал на остановку потока.
      if (!parm.hWnd)
          break;

      if (lstrcmp(df.cFileName, TEXT(".")) &&
          lstrcmp(df.cFileName, TEXT("..."))) {
          lstrcpy(tt, szRoot);
          lstrcat(tt, df.cFileName);

          if (df.dwFileAttributes &
                    FILE_ATTRIBUTE_DIRECTORY) {
            // Директория.
            // Делаем рекурсивный вызов этой же функции.
            if (tt[lstrlen(tt)-1] != TEXT('\\'))
                lstrcat(tt, TEXT("\\"));
            CheckFiles(parm, tt);
          }

          else {
            // Файл.
            // Добавляем новую строку в ListView.
            NEW_FILE_DATA arData;
            lstrcpy(arData.szCheck, TEXT("?"));
            lstrcpy (arData.szName, tt);
            wsprintf(arData.szLen, TEXT("%d"),
                        df.nFileSizeLow);
            ::FileTimeToSystemTime(&df.ftLastWriteTime,
                        &sysTime);
            wsprintf(arData.szDate, TEXT("%02d.%02d.%d"),
                        sysTime.wDay, sysTime.wMonth,
                        sysTime.wYear);

            WPARAM nItem = InsertListRow(parm.hwndList,
                        &arData);

            lstrcpy(arData.szCheck, TEXT("ERR"));
            HANDLE hFile = ::CreateFile(tt,
                  GENERIC_READ, FILE_SHARE_READ, NULL,
                  OPEN_EXISTING,
```

```
        FILE_ATTRIBUTE_NORMAL |
        FILE_FLAG_NO_BUFFERING,
        NULL);

    if (hFile != INVALID_HANDLE_VALUE) {
        // Читаем файл.
        BYTE  buff[8*512];
        DWORD nReaded = 0, len = 0, step = 19;
        do {
          if (!::ReadFile(hFile, buff,
                      sizeof(buff),
                      &nReaded, NULL)) {
            lstrcpy(arData.szCheck,
                      TEXT("ERR"));
            nReaded = (DWORD)(-1);
            break;
          }
          // Снова проверяем сигнал
          // на остановку потока.
          if (!parm.hWnd) {
            rb = 0;
            break;
          }

          // Если длина файла больше 40000 байт,
          // показываем процент выполнения
          // в колонке статуса.
          len += nReaded;

          if (parm.hWnd && df.nFileSizeLow >
                40000 &&
                100*len/df.nFileSizeLow > step) {
            wsprintf((LPTSTR)buff,
              TEXT("%d %c"),
              100*len/df.nFileSizeLow,
              TEXT('%'));
            step += 20;
            UpdateCheckRow(parm.hwndList,
                      nItem, (LPTSTR)buff);
          }

        } while (nReaded);

        if (!nReaded)
          lstrcpy(arData.szCheck, TEXT("ok"));

        ::CloseHandle(hFile);
    }
    // Показываем окончательный
    // результат проверки.
    if (parm.hWnd)
        UpdateCheckRow(parm.hwndList, nItem,
                      arData.szCheck);
```

```
            }
        }

        if (rb)
            rb = ::FindNextFile(hd, &df);

    }

    ::FindClose(hd);
}
```

Часть 3. Windows программирование на основе MFC

14. Введение в Visual C++

В связи с тем, что сегодня уровень сложности программного обеспечения очень высок, разработка Windows приложений с использованием только какого-либо языка программирования значительно затрудняется. Разработчик должен затратить массу времени на решение стандартных задач по созданию, например, многооконного интерфейса, а реализация технологии связывания и встраивания объектов – **OLE** – потребует от разработчика еще больших усилий.

Чтобы облегчить работу программиста практически все современные компиляторы языка C++ содержат специальные библиотеки классов. Такие библиотеки включают в себя практически весь программный интерфейс Windows и позволяют пользоваться при программировании средствами более высокого уровня, чем обычные вызовы функций. За счет этого значительно упрощается разработка приложений, имеющих сложный интерфейс пользователя.

Современные интегрированные системы разработки Windows приложений позволяют автоматизировать сам процесс создания приложения. Для этого, например, используются генераторы приложений. Основная идея заключается в том, что разработчик отвечает на вопросы генератора приложений и определяет свойства приложения, например, будет ли оно поддерживать многооконный интерфейс, будут ли использоваться трехмерные органы управления, нужна ли справочная система, а также ряд других. После этого генератор приложений создаст приложение, отвечающее указанным требованиям, и предоставит исходные тексты. Пользуясь этими текстами как шаблоном, разработчик завершает создание собственного приложения.

Подобные средства автоматизированного создания приложений включены в состав компилятора Microsoft Visual C++ и называются MFC AppWizard. Заполнив несколько диалоговых панелей, можно указать характеристики приложения и получить его тексты, снабженные поясняющими комментариями. MFC AppWizard позволяет создавать однооконные и многооконные приложения, а также приложения, не имеющие главного окна, когда вместо него используется диалоговая панель.

Конечно, MFC AppWizard не всесилен – прикладную часть приложения разработчику придется разрабатывать самостоятельно. Исходный текст приложения, созданный MFC AppWizard, станет только основой, к которой нужно подключить остальное. Но работающий шаблон приложения – это уже половина всей работы. Исходные тексты приложений, автоматически

полученные от MFC AppWizard, могут составлять сотни строк текста, набор которого вручную был бы весьма утомителен.

Нужно отметить, что MFC AppWizard создает тексты приложений только с использованием библиотеки классов MFC – Microsoft Foundation Classes. Поэтому только изучив язык C++ и библиотеку MFC, можно пользоваться средствами автоматизированной разработки для создания собственных приложения в кратчайшие сроки.

Итак, MFC – это библиотека или базовый набор классов, написанных на языке C++ и предназначенных для упрощения и ускорения процесса программирования под Windows. Перед изучением библиотеки MFC и ее использованием для создания Windows приложений, следует вспомнить, как работает сама Windows, каковы принципы взаимодействия программ с операционной системой и какова структура типичной Windows программы.

14.1. Интерфейс вызовов функций в Windows

Благодаря данному интерфейсу доступ к системным ресурсам осуществляется через целый ряд системных функций. Совокупность таких функций называется прикладным программным интерфейсом, или API – Application Programming Interface. Для взаимодействия с Windows приложение запрашивает функции API, с помощью которых реализуются все необходимые системные действия, такие как выделение памяти, вывод на экран, создание окон и проч.

Библиотека MFC инкапсулирует многие функции API. Хотя программам и разрешено обращаться к ним напрямую, все же чаще это будет выполняться через соответствующие методы классов.

14.2. Библиотеки динамической загрузки

Поскольку API состоит из большого числа функций, может сложиться впечатление, что при компиляции каждой программы, написанной для Windows, к ней подключается код довольно значительного объема. В действительности это не так. Функции API содержатся в библиотеках динамической компоновки – Dynamic Link Libraries, или DLL, – которые загружаются в память только в тот момент, когда к ним происходит обращение, т.е. при выполнении программы. Рассмотрим вкратце, как осуществляется механизм динамической загрузки.

Динамическая загрузка обеспечивает ряд существенных преимуществ. Во-первых, поскольку практически все программы используют API функции, то благодаря DLL библиотекам существенно экономится дисковое пространство, которое в противном случае занималось бы большим количеством повторяющегося кода, содержащегося в каждом из исполняемых модулей. Во-вторых, изменения и улучшения в Windows приложениях сводят-

ся к обновлению только содержимого DLL библиотек. Уже существующие тексты программ не требуют перекомпиляции.

14.3. Преимущества использования MFC

Библиотека содержит многоуровневую иерархию классов, насчитывающую более 200 объектов. Они дают возможность создавать Windows приложения на базе объектно-ориентированного подхода. С точки зрения разработчика, MFC представляет собой каркас, на основе которого можно создавать программы для Windows.

1) Библиотека MFC разрабатывалась для упрощения задач, стоящих перед разработчиками. Как известно, традиционный метод программирования под Windows требует написания достаточно длинных и сложных программ, имеющих ряд специфических особенностей. По мере же увеличения сложности программы ее код может достигать поистине невероятных размеров. Однако та же самая программа, написанная с использованием MFC, будет в несколько раз меньше по объему, поскольку большинство рутинных, частных деталей скрыто от разработчика.

2) Другим преимуществом работы с MFC является возможность многократного использования одного и того же кода. Так как библиотека содержит много элементов, общих для всех Windows приложений, нет необходимости каждый раз создавать их заново. Вместо этого их можно просто наследовать, говоря языком объектно-ориентированного программирования. Кроме того, интерфейс, обеспечиваемый библиотекой, практически независим от конкретных деталей, его реализующих. Поэтому программы, написанные на основе MFC, могут быть легко адаптированы к новым версиям Windows, в отличие от большинства программ, написанных обычными методами. Кроме того, библиотека MFC создана той же фирмой, что и сама операционная система.

3) Еще одним существенным преимуществом MFC является упрощение взаимодействия с прикладным программным интерфейсом Windows. Любое приложение взаимодействует с Windows через API, который содержит более шестисот функций. Внушительный размер API затрудняет попытки понять и изучить его целиком. Зачастую, даже сложно проследить, как отдельные части API связанны друг с другом. Но поскольку библиотека MFC объединяет путем инкапсуляции функции API в логически организованное множество классов, интерфейсом становится значительно легче управлять.

Не все классы MFC напрямую связаны с технологией программирования под Windows. Например, класс **`CString`** представляет собой объект “строка”, чего так не хватало в языке C. Класс **`CFile`** предназначен для

управления файлами и так далее. Эта группа объектов именуется *классами общего назначения*. Главная же часть библиотеки MFC состоит из классов, используемых для построения компонентов приложения.

14.4. Основные компоненты MFC приложения

Перечислим вкратце наиболее важные классы и объекты, участвующие в проектировании любого MFC приложения.

С каждым MFC приложением связывается определяющий его на верхнем уровне объект **theApp**, принадлежащий классу, производному от **CWinApp**.

Как правило, структура MFC приложения определяется архитектурой, именуемой "документ-вид"(Document-View). Это означает, что приложение состоит из одного, реже нескольких, документов – объектов, классы которых являются производными от класса **CDocument**. С каждым из документов связаны один или несколько его представлений или видов – объектов классов, производных от **CView**. Эти объекты определяют облик документа.

Класс **CFrameWnd** (окно-рамка) и производные от него определяют, так называемые окна-рамки приложения.

Все стандартные элементы управления, используемые для пользовательского интерфейса, имеют своих представителей в MFC. Это целое семейство классов, относящихся к элементам управления.

Не забыты, конечно, как модальные, так и немодальные диалоговые панели. Их обслуживает один класс **CDialog**.

Перечисленные классы **CView**, **CFrameWnd**, **CDialog** и все классы для элементов управления имеют один и тот же базовый класс – **CWnd**, который, по существу, определяет любое Windows окно. Этот класс, в свою очередь, является наследником своего базового класса **CObject**.

Основная и принципиальная трудность в понимании принципов устройства любого MFC приложения, заключается вы том, что объекты, из которых оно строится, наследуют свойства и поведение всех своих предков, поэтому необходимо твердо знать *все основные базовые классы* MFC.

14.5. Архитектура MFC приложения

Как известно, все Windows приложения имеют фиксированную структуру, которую определяет функция **WinMain**. Архитектура приложения, построенного из объектов классов библиотеки MFC, является еще более структурированной.

MFC приложение состоит из объекта **theApp**, функции **WinMain**, и некоторого количества других объектов. Сердцевина приложения – это объект **theApp**. Он отвечает за создание всех остальных объектов и обработку

очереди сообщений. Объект **theApp** является глобальным и создается *еще до начала* работы функции **WinMain**. Если рассматривать упрощенно, то работа функции **WinMain** заключается в последовательном вызове двух методов объекта **theApp**: **InitInstance** и **Run**.

При обработке метода **InitInstance**, объект **theApp** создает внутренние объекты приложения. Процесс создания выглядит как последовательное порождение одних объектов другими. Набор объектов, порождаемых в начале этой цепочки, определен структурой MFC практически однозначно – это шаблон, главная рамка, документ и вид документа. Их роли в работе приложения мы обсудим позже.

Следующий важный метод объекта **theApp** – это **Run**. Он, по существу, запускает работу приложения и начинает процесс обработки сообщений. Объект **theApp** циклически выбирает сообщения из очереди и инициирует обработку сообщений объектами приложения.

Некоторые объекты приложения остаются невидимыми, а часть их них имеют графический образ на экране. Именно с ними и взаимодействует пользователь. Эти интерфейсные объекты обычно связаны с Windows окнами. Среди них особенно важны главная рамка и объекты вид. Как раз им объект **theApp** прежде всего перенаправляет сообщения из очереди сообщений.

Когда пользователь выбирает команды меню окна главной рамки приложения, то возникают командные сообщения. Они отправляются сначала объектом **theApp** объекту главная рамка, а затем обходят по специальному маршруту целый ряд объектов, среди которых первыми являются документ и его вид, информируя эти объекты о пришедшей от пользователя команде.

При работе приложения возникают и обычные вызовы методов одного класса в объектах других классов. Такие перекрестные вызовы также будем именовать сообщениями. В Visual C++ некоторым методам приписан именно этот статус, например, методу **OnDraw**.

14.6. Наследование – главный принцип построения MFC приложений

Наследование – одна из фундаментальных идей объектно-ориентированного программирования. Именно этот механизм наследования позволяет разработчику дополнять и переопределять поведение базового класса, не вторгаясь в библиотеку MFC, которая остается неизменной. Все изменения делаются в производном классе.

Объекты, их которых состоит приложение, являются объектами классов, производных от классов библиотеки MFC. Разработка приложения состоит в следующем. Разработчик берет из библиотеки MFC классы

CWinApp, **CFrameWnd**, **CDocument**, **CView** и проч. и строит от них производные классы. Приложение создается как совокупность объектов этих производных классов. Каждый объект несет в себе как наследуемые черты, определяемые базовыми классами, так и новые черты, добавленные разработчиком. Наследуемые черты определяют общую схему поведения, свойственную таким приложениям. Новые же черты позволяют реализовать специфические особенности приложения, необходимые для решения стоящей перед разработчиком задачи.

При определении производного класса программист может: 1) переопределить некоторые методы базового класса, причем те методы, что не были переопределены, будут наследоваться в том виде, в каком они существуют в базовом классе; 2) добавить новые переменные и методы.

14.7. Каркас приложения

Приложение, построенное на основе библиотеки MFC – это “айсберг”, большая часть которого невидима, но является основой всего приложения. Часть приложения, лежащая в библиотеке MFC называется каркасом приложения – framework. Рассмотрим работу приложения как процесс взаимодействия между каркасом и частью приложения, разработанной программистом. Совершенно естественно, что в методах, определенных программистом, могут встречаться вызовы методов базового класса, что вполне можно рассматривать как вызов функции из библиотеки. Но метод производного класса, определенный разработчиком, может быть вызван из метода родительского класса. Другими словами, каркас и производный класс в этом смысле равноправны, поскольку их методы могут вызывать друг друга. Такое равноправие достигается благодаря виртуальным методам и полиморфизму, имеющимся в арсенале объектно-ориентированного программирования.

Пусть некоторый метод базового класса объявлен виртуальным. Разработчик переопределяет его в производном классе. Это значит, что при вызове данного метода в некоторой полиморфной функции базового класса *в момент исполнения* будет вызван метод производного класса и, следовательно, каркас вызывает метод, определенный программистом. Точнее говоря, обращение к этому методу должно производиться через ссылку на производный объект либо через объект, являющийся формальным параметром и получающий при вызове в качестве своего значения адрес объекта производного класса. Когда вызывается виртуальный метод M1, переопределенный разработчиком, то согласно терминологии Visual C++, каркас посылает сообщение M1 объекту производного класса, а метод M1 этого объекта обрабатывает это сообщение. Если сообщение M1 послано объекту производного класса, а обработчик этого сообщения не задан программи-

стом, объект наследует метод M1 ближайшего родительского класса, в котором определен этот метод. Если же обработчик такого сообщения создан программистом, он автоматически отменяет действия, предусмотренные родительским классом.

14.8. Проект приложения

Рассмотрим этапы создания приложения с помощью Visual C++. Сначала разберем такое важное понятие, как проект. До сих пор приложение рассматривалось в виде совокупности объектов базовых и производных классов. Но для обеспечения работы приложения требуется нечто большее – наряду с описанием классов необходимо описание ресурсов, связанных с приложением, нужна справочная система и проч. Термин “проект” используется именно в такой, более широкой интерпретации.

1. В среде Visual C++ можно строить различные типы проектов. Такие проекты после их создания можно компилировать и запускать на исполнение. Корпорация Microsoft разработала специальный инструментарий, облегчающий и ускоряющий создание проектов в среде Visual C++. Например, мастер MFC AppWizard (exe) позволяет создать проект Windows приложения, которое имеет однодокументный, многодокументный или диалоговый интерфейс и использует библиотеку MFC.

2. Создаваемый остов приложения составлен так, что в дальнейшей работе с проектом можно использовать другое инструментальное средство – ClassWizard (мастер классов), предназначенное для создания производных MFC классов. Еще одно назначение ClassWizard состоит в том, что он создает шаблоны для переопределяемых методов базового класса. Кроме того, он позволяет отследить сообщения, приходящие классу, и создать шаблон обработчика любого из этих сообщений. Рассматривайте ClassWizard не как панацею, но как толкового помощника, уберегающего вас от множества рутинных ошибок.

3. Для создания ресурсов приложения предназначен редактор ресурсов. Он позволяет быстро создавать новые меню, диалоговые панели, добавлять кнопки к панели управления и проч.

14.9. Типы мастеров проектов

Рассмотрим основные типы проектов, которые можно создавать при помощи мастеров проектов Microsoft Visual C++:

MFC AppWizard (exe) – при помощи мастера приложений можно создать проект Windows приложения, которое имеет однодокументный (SDI), многодокументный (MDI) или диалоговый интерфейс. Однодокументное приложение предоставляет пользователю возможность работать только с одним файлом. Многодокументное приложение, напротив, может одновре-

менно представлять несколько документов, каждый в собственном окне. Пользовательский интерфейс диалогового приложения представляет собой единственное диалоговое окно.

AppWizard (dll) – этот мастер приложений позволяет создать структуру DLL, основанную на MFC. При помощи него можно определить характеристики будущей DLL.

AppWizard ATL COM – это средство позволяет создать элемент управления ActiveX или сервер автоматизации, используя новую библиотеку шаблонов ActiveX (ActiveX Template Library - ATL). Опции этого мастера дают возможность выбрать активный сервер (DLL) или исполняемый внешний сервер (exe-файл).

Custom AppWizard – при помощи этого средства можно создать пользовательские мастера AppWizard. Пользовательский мастер может базироваться на стандартных мастерах для приложений MFC или DLL, а также на существующих проектах или содержать только определяемые разработчиком шаги.

DevStudio Add-in Wizard – мастер дополнений позволяет создавать дополнения к Visual Studio. Библиотека DLL расширений может поддерживать панели инструментов и реагировать на события Visual Studio.

ActiveX ControlWizard - мастер элементов управления реализует процесс создания проекта, содержащего один или несколько элементов управления ActiveX, основанных на элементах управления MFC.

Win32 Application – этот мастер позволяет создать проект обычного Window приложения. Проект создается незаполненным, файлы с исходным кодом в него следует добавлять вручную.

Win32 Console Application – мастер создания проекта консольного приложения. Консольное приложение – это программа, которая выполняется из командной строки Windows и не имеет графического интерфейса. Проект консольного приложения создается пустым, предполагая добавление файлов исходного текста в него вручную.

Win32 Dynamic-Link Library – создание пустого проекта динамически подключаемой библиотеки. Установки компилятора и компоновщика будут настроены на создание DLL. Исходные файлы следует добавлять вручную.

Win32 Static Library – это средство создает пустой проект, предназначенный для генерации объектной, статической библиотеки. Файлы с исходным кодом в него следует добавлять вручную.

14.10. Имена, используемые в MFC

Библиотека MFC содержит большое количество классов, структур, констант и проч. Для того, чтобы текст MFC приложений был более легким

для понимания, принято применять ряд соглашений для используемых имен и комментариев.

Названия всех классов и шаблонов классов библиотеки MFC начинаются с заглавной буквы **C**. При наследовании классов от классов MFC можно давать им любые имена. Рекомендуется начинать их названия также с заглавной буквы **C**. Это сделает исходный текст приложения более ясным для понимания.

Чтобы отличить элементы данных, входящих в класс, от простых переменных, их имена принято начинать с префикса **m_**. Названия методов классов, как правило, специально не выделяются, но обычно их начинают с заглавной буквы.

Библиотека MFC включает в себя, помимо классов, набор служебных функций. Названия этих функций начинаются с символов **Afx**, например **AfxGetApp**. Символы AFX являются аббревиатурой от Application Framework X, означающих основу приложения, его внутреннее устройство.

Символы AFX встречаются не только в названии функций MFC. Многие константы, макрокоманды и другие символы начинаются с этих символов. В общем случае AFX является признаком, по которому можно определить принадлежность того или иного объекта – функция, переменная, ключевое слово или символ – к библиотеке MFC.

Когда приложение разрабатывается средствами MFC AppWizard и ClassWizard, они размещают в исходном тексте приложения комментарии следующего вида:

```
//{{AFX_
...
//}}AFX_
```

Такие комментарии образуют блок кода программы, который управляется только средствами MFC AppWizard и ClassWizard. Пользователь может, но *не должен* вручную вносить изменения в этом блоке. Для этого необходимо употреблять средства ClassWizard.

В следующей таблице представлено краткое описание некоторых блоков:

Блок	Описание
//{{AFX_DATA **//}}AFX_DATA**	Включает объявление элементов данных класса. Используется в описании классов диалога.
//{{AFX_DATA_INIT **//}}AFX_DATA_INIT**	Включает инициализацию элементов данных класса. Используется в файле реализации классов диалога
//{{AFX_DATA_MAP **//}}AFX_DATA_MAP**	Включает макрокоманды DDX, предназначенные для связывания элементов данных класса и органов управления диалоговых панелей. Используется в

	файле реализации классов диалоговых панелей.
//{{AFX_MSG **//}}AFX_MSG**	Включает описание методов, которые предназначены для обработки сообщений. Этот блок используется при объявлении класса.
//{{AFX_MSG_MAP **//}}AFX_MSG_MAP**	Включает макрокоманды таблицы сообщений класса. Используются совместно с AFX_MSG.
//{{AFX_VIRTUAL **//}}AFX_VIRTUAL**	Включает описание переопределенных виртуальных методов класса. Блок AFX_VIRTUAL используется при объявлении класса.

MFC AppWizard и ClassWizard помогают разрабатывать приложения. Они создают все классы и методы, необходимые для его работы. Программисту остается дописать к ним свой код. В тех местах, где можно вставить свой код, MFC AppWizard и ClassWizard, как правило, помещают комментарий

```
//TODO:
```

15. Краткий обзор классов MFC

Библиотека классов MFC содержит большое количество разнообразных классов. Каждый класс, как правило, содержит несколько методов и элементов данных – свойств. Для детального изучения иерархии классов MFC можно воспользоваться справочной документацией или справочной системой среды Developer Studio. Здесь же мы проведем краткий обзор.

15.1. CObject – самый базовый класс библиотеки MFC

Подавляющее большинство классов MFC наследовано от базового класса **`CObject`**, лежащего в основе всей иерархии классов этой библиотеки. Методы и элементы данных класса **`CObject`** представляют наиболее общие свойства наследованных из него классов MFC.

Класс **`CObject`**, а также все классы, наследованные от него, обеспечивают возможность сохранения объектов класса в файлах на диске с их последующим восстановлением. Для объектов классов, наследованных от базового класса **`CObject`**, уже во время работы приложения можно получить разнообразную информацию о классе объекта.

Ряд методов класса **`CObject`** предназначен для получения дампа объектов класса во время отладки приложения. Эта особенность класса может ускорить процесс поиска ошибок в приложении.

15.2. CCmdTarget – основа структуры приложения MFC

Непосредственно от класса **`CObject`** наследуются ряд классов, которые сами являются базовыми для остальных классов MFC. В первую очередь это класс **`CCmdTarget`**, представляющий основу структуры любого приложения. Главной особенностью класса **`CCmdTarget`** и классов, наследованных от него, является то, что объекты этих классов могут получать от операционной системы сообщения и обрабатывать их. От класса **`CCmdTarget`** наследуется несколько классов, краткое описание которых следует ниже.

15.3. Классы CWinThread и CWinApp – подзадачи приложения

От класса **`CCmdTarget`** наследуется класс **`CWinThread`**, представляющий поток приложения. Простые приложения, которые будут рассматри-

ваться дальше, имеют только один поток. Эта подзадача, называемая главной, представляется классом **CWinApp**, наследованным от класса **CWinThread**.

15.4. CDocument – документ приложения

Большинство приложений работают с данными или документами, хранимыми на диске в отдельных файлах. Класс **CDocument**, наследованный от базового класса **CCmdTarget**, служит для представления документов приложения.

15.5. CDocTemplate, CSingleDocTemplate и CMultiDocTemplate – шаблоны документов

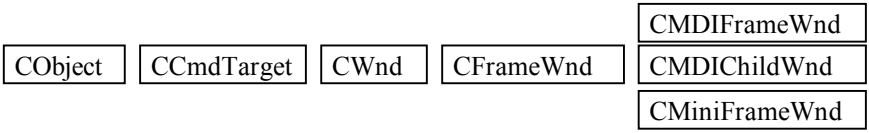
CDocTemplate – еще один важный класс, наследуемый от **CCmdTarget**. От этого класса наследуется два класса: **CSingleDocTemplate** и **CMultiDocTemplate**. Все три класса предназначены для синхронизации и управления основными объектами, представляющими приложение, – окнами, документами и используемыми ими ресурсами.

15.6. CWnd – базовый класс окон

Практически все приложения имеют пользовательский интерфейс, построенный на основе окон. Это может быть диалоговая панель, одно окно или несколько взаимосвязанных окон. Основные свойства окон представлены классом **CWnd**, наследованным от класса **CCmdTarget**.

Разработчики очень редко создают объекты класса **CWnd**. Этот класс сам является базовым классом для большого количества классов, представляющих разнообразные окна. Перечислим классы, наследованные от базового класса CWnd.

15.7. CFrameWnd – окно обрамляющей рамки

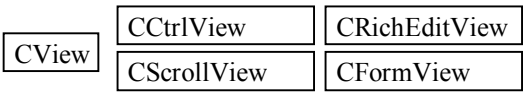


Класс **CFrameWnd** представляет окна, выступающие в роли обрамляющих рамок, в том числе главные окна приложений. От этого класса также наследуются классы **CMDIChildWnd** и **CMDIFrameWnd**, используемые для отображения окон многооконного интерфейса MDI. Класс **CMDIFrameWnd** представляет главное окно приложения MDI, а класс **CMDIChildWnd** - все дочерние окна MDI. Класс **CMiniFrameWnd** применяется для отображения окон уменьшенного размера. Такие окна обычно используются для отображения панелей управления.

15.8. CView – окна просмотра

Большой интерес представляет класс **CView** и классы, наследуемые от него. Эти классы являются окнами просмотра документов приложения. Именно окно просмотра используется для вывода на экран содержимого документа, с которым работает пользователь. Через это окно он может изменять документ.

Разрабатывая приложение, программисты наследуют собственные классы просмотра документов либо непосредственно от базового класса **CView**, либо от одного из порожденных классов, определенных в библиотеке MFC.



Так, классы, наследованные от **CCtrlView**, это, по существу, элементы управления, которые по размерам совпадают с окном отображения. Например, класс **CRichEditView** использует орган управления Rich Edit.

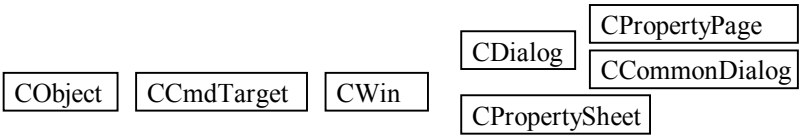
Класс **CScrollView** представляет окно просмотра, которое имеет полосы прокрутки. В классе определены специальные методы, управляющие полосами просмотра.

> В этом классе реализован пиксельный скроллиг содержимого окна, базирующийся на Win16 API функциях. Следовательно, объем документа не может превышать 2000 строк при 16-ти пиксельной высоте символов.

Класс **CFormView** позволяет создать окно просмотра документа, основанное на шаблоне диалоговой панели. Можно сказать, что это альтернатива обычному диалогу операционной системы.

> От класса **CFormView** наследуются два класса – **CRecordView** и **CdaoRecordView**, которые используются для просмотра записей баз данных.

15.9. CDialog – диалоговые панели



От базового класса наследуются классы, управляющие диалоговыми панелями. Все модальные и немодальные диалоговые панели пользователя выводятся от класса **CDialog**.

Вместе с диалоговыми панелями обычно используется класс **CDataExchange**, который обеспечивает работу функций обмена данными DDX (Dialog Data Exchange) и проверки данных DDV (Dialog Data Validation), используемых для диалоговых панелей.

> Класс **CDataExchange** не наследуется от какого-либо другого класса.

От класса **CDialog** наследуется **CCommonDialog**, а от него уже ряд классов, представляющих собой стандартные диалоговые панели. Например, для выбора файла используются объекты класса **CFileDialog**, для выбора шрифта – **CFontDialog**, цвета – **CColorDialog**, вывода документа на печать – **CPrintDialog**, настройки параметров страницы для печати – **CPageSetupDialog**.

Чтобы создать стандартный диалог, нужно просто определить объект соответствующего класса, а дальнейшее управление такой панелью осуществляется методами данного класса.

От класса **CDialog** наследуется также диалог **CPropertyPage**, который является реализацией объекта “страница свойств”. Эти страницы, в свою очередь, помещаются в объект класс **CPropertySheet**, который можно рассматривать как контейнер для страниц свойств.

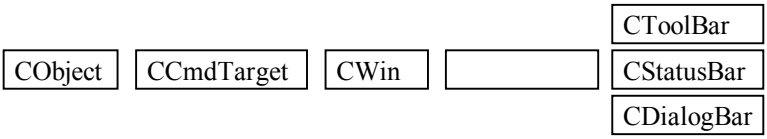
15.10. Семейство классов - элементов управления

Для работы с элементами управления – кнопки, полосы прокрутки, редакторы текста и проч. – в библиотеке MFC предусмотрены специальные классы, наследованные непосредственно от класса **CWnd**. Перечислим эти классы:

Класс органа управления	Описание
CButton	кнопка
CComboBox	список с окном редактирования
CEdit	поле редактирования
CListBox	список
CScrollBar	полоса просмотра
CStatic	статический элемент управления
CAnimate	отображение видеоинформации
CBitmapButton	кнопка с рисунком
CHeaderCtrl	заголовок для таблицы
CHotKeyCtrl	работа с клавишами акселераторов
CListCtrl	отображение списка пиктограмм
CProgressCtrl	линейный индикатор
CRichEditControl	окно редактирования с форматированием текста
CSliderCtrl	движок
CSpinButtonCtrl	увеличивает или уменьшает значение параметра
CTreeCtrl	иерархическая структура данных
CToolTipCtrl	окно, содержащее строку текста подсказки

В представленном списке, выше черты располагаются “старые” элементы управления, пришедшие из Win16, а ниже – “новые” элементы, появившиеся в Win32. Конечно, “старые” элементы частично адаптированы для 32-х разрядной операционной системы, но эта модификация не затронула основные ограничения, присущие 16-ти разрядной Windows.

15.11. Управляющие панели



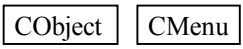
Класс **CControlBar** и классы, наследуемые от него, предназначены для создания управляющих панелей. Такие панели могут содержать различные элементы управления и отображаются, как правило, в верхней части главного окна приложения.

Класс **CToolBar** предназначен для создания панели управления. Эта панель обычно содержит ряд кнопок, дублирующих действие меню приложения.

Класс **CStatusBar** является отображением панели состояния в виде полосы в нижней части окна. В этой панели приложение может размещать всевозможную информацию, например, краткую подсказку о выбранной строке меню.

Большие возможности представляет управляющая панель, созданная на основе класса **CDialogBar**. Такая панель использует обычный шаблон диалоговой панели, которую можно разработать в редакторе ресурсов.

15.12. CMenu – класс меню приложений



Практически каждое приложение имеет собственное меню. Для управления меню в состав MFC включен специальный класс **CMenu**, наследованный непосредственно от базового класса **CObject**. Методы класса **CMenu** позволяют выполнять различные операции по изменению состава пунктов меню, а для модификации существующих пунктов используется другой класс – **CCmdUI**. Заметим, что этот класс *не наследуется* от базового класса **CObject**.

Объекты класса **CCmdUI** создаются динамически, когда пользователь выбирает строку меню или нажимает кнопку панели управления. Методы класса **CCmdUI** позволяют управлять состоянием строк меню или кнопок панели управления. Например, существует метод, который делает строку меню неактивной.

15.13. Массивы, списки, словари

В состав MFC включен целый набор вспомогательных, но весьма полезных классов, предназначенных для хранения информации в массивах, списках и словарях. Все эти классы наследованы от базового класса **CObject**.

Массивы

В языке С дано классическое понятие массива со множеством ограничений. Классы MFC обеспечивают более широкую трактовку и дают дополнительные возможности при работе с массивами. Так, MFC массивы обеспечивают динамичность, т.е. возможность изменения размерности массива.

Для представления массивов предназначены следующие классы:

MFC класс	Элементы массива
CByteArray	**unsigned char**
CDWordArray	**unsigned long**
CObArray	указатели на объекты класса **Cobject**
CPtrArray	указатели типа **void**
CStringArray	объекты класса **CString**
CUIntArray	**unsigned int**
CWordArray	**unsigned short**

Основная схема использования MFC массивов состоит в следующем:

1) объявляется объект класса массив, например,

```
CByteArray arByte;
```

2) указывается на сколько элементов будет увеличиваться блок памяти, выделенный под весь массив, при его динамическом расширении.

Эту операцию можно опустить, но она позволяет существенно уменьшить общую фрагментацию памяти, и следовательно, повышает эффективность вашего приложения.

```
arByte.SetSize(0, 4); // на 4 элемента
```

3) массив заполняется элементами

```
arByte.Add(0x17);
arByte.Add(0x255);
```

4) доступ к элементам массива осуществляется через перегруженную операцию индексирования

```
arByte[k] = 20;
```

или метод **GetAt()**

```
BYTE bTest = arByte.GetAt(k);
```

5) освобождать память, выделенную под массив, не требуется, т.к. это происходит автоматически при вызове деструктора класса.

Списки

Для решения многих задач применяются такие структуры хранения данных, как списки. MFC включает ряд классов, наследованных от базового класса **CObject**, которые предоставляют разработчику готовое решение для создания собственных списков. В этих классах определены все методы, необходимые при работе со списками: добавление нового элемента, вставка нового элемента, определение следующего или предыдущего элемента в списке, удаление элемента и проч.

Ниже перечислены классы, позволяющие построить списки из элементов любых типов:

MFC класс	Элементы списка
CObList	указатели **CObject**
CPtrList	указатели типа **void**
CStringList	объекты класса **CString**

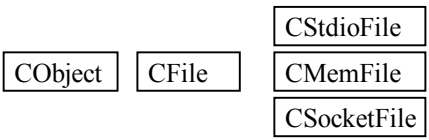
Словари

Словарь – это информационная структура, которая представляет собой таблицу из двух колонок, устанавливающих соответствие между парами значений. Первая величина представляет собой ключевое значение и записывается в первую колонку, а вторая – связанное с ней значение, хранящееся во второй колонке. Словарь позволяет добавлять в него пары связанных величин и осуществлять выборку значений по ключевому слову.

Для работы со словарями используются следующие MFC классы:

Класс	Ключевое слово	Связанное значение
CMapPtrToPtr	указатель типа void	указатель типа void.
CMapPtrToWord	указатель типа void	WORD
CMapStringToOb	объекты класса CString	указатель класса CObject
CMapStringToPtr	объекты класса CString	указатель типа void
CMapStringToString	объекты класса CString	указатель класса CObject
CMapWordToOb	WORD	указатель класса CObject
CMapWordToPtr	WORD	указатель типа void

15.14. Файловая система



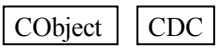
Библиотека MFC включает класс **CFile,** предназначенный для работы с файловой системой Windows, и также наследуемый от базового класса **CObject**. Класс **CFile** инкапсулирует все необходимые операции для работы с файлом, как последовательностью байт. Если же файл содержит текстовую информацию и строки разделены сепараторами **\r\n**, то удобнее воспользоваться методами класса **CStdioFile**, специально предназначенного для манипулирования такими файлами.

Кроме того, от класса **CFile** наследуются еще несколько классов, среди которых упомянем **CMemFile** и **CSocketFile**. Первый представляет собой файл, спроецированный в оперативную память, а второй – файл, передаваемый через сокеты.

При работе с файловой системой может потребоваться различная информация о некотором файле, например, дата создания, его размер и проч. Для получения этих данных предназначен специальный класс **CFileStatus**. Заметим, что этот класс *не наследуются* от базового класса **CObject**.

> Класс **CFileStatus** используется также для решения задачи об определении существования файла с заданным именем.

15.15. Контекст отображения



Для отображения информации в окне или на любом другом устройстве приложение должно получать контекст отображения. Основные свойства контекста отображения определены в классе **CDC**. От него наследуется еще четыре класса, представляющие контекст отображения различных устройств:

CClientDC контекст отображения, связанный с клиентской областью окна. Для получения контекста конструктор класса вызывает функцию **GetDC** из Win32 API, а деструктор - функцию **ReleaseDC**.

CPaintDC конструктор класса **CPaintDC** для получения контекста отображения вызывает метод **CWnd::BeginPaint**, а деструктор - метод **CWnd::EndPaint**. Объекты данного класса можно использовать только при обработке сообщения **WM_PAINT** в обработчике **OnPaint**.

CWindowDC контекст отображения, связанный со всем окном. Для полу-

чения контекста конструктор класса вызывает функцию **GetWindowDC** из Win32 API, а деструктор – функцию **ReleaseDC**.

CMetaFileDC класс предназначен для работы с метафайлами.

15.16. Объекты графического интерфейса

Для отображения информации используются различные объекты графического интерфейса – GDI-объекты. Для каждого из этих объектов библиотека MFC содержит соответствующий класс, наследованный от базового класса **CGdiObject:**

CBitmap	изображение bitmap
CBbrush	кисть
CFont	шрифт
CPalette	палитра цветов
CPen	перо
CRgn	произвольная область внутри окна

15.17. Исключения

Напомним, что стандарт языка C++ включает такую конструкцию, как **try** блок. Любые ошибки, возникающие в коде, охваченном **try** блоком, генерируют исключения посредством **throw** операторов. При возникновении ошибки, выполнение кода прекращается, а управление передается на соответствующие **catch** обработчики, расположенные за пределами **try** блока.

MFC, не меняя самой идеи перехвата ошибок, проводит структуризацию исключений, так что каждая ошибочная ситуация имеет собственный **catch** обработчик. Коллекция, насчитывающая 11 классов, обслуживает все возможные исключительные ситуации, среди которых отметим:

Класс **CException**, является базовым для этой группы, но сам практически не используется.

CMemoryException класс, объекты которого локализует все исключительные ситуации, возникающие в памяти.

CFileException класс, объекты которого локализует все исключительные ситуации, возникающие при работе с файлами.

CDBException исключения, возникающие при работе MFC объектов, основанных на ODBC технологии.

CUserException исключения, которые может генерировать разработчик, вызывая MFC функцию AfxThrowUserException().

Здесь представлен псевдокод, который дает представление, как разработчик может использовать указанные выше возможности.

```
      TRY {

(1)         // код приложения, который может
            // вызвать исключение или исключения

      }

      // в эту точку попадем только тогда,
      // когда исключений не было

      CATCH ( CUserException, e ) {
            // обработка исключений
            // разработчика
            ViewUserError(e);
      }
      AND_CATCH ( CDBException, e ) {
            // обработка DB исключений
            ViewDBError(e);
      }
      AND_CATCH ( CFileException, e ) {
            // обработка файловых исключений
(2)         ViewFileEerror(e);
      }
      END_CATCH
```

Из схемы видно, что MFC заменяет ключевые слова языка C++ **`try`** и **`catch`** аналогичными макросами. Объясняется это необходимостью передачи информацию из точки возникновение исключения (1) в точку его обработки (2). Достичь этого можно динамическим созданием объекта, например класса **`CFileException`**, в точке возникновение исключения. Однако это потребует динамического разрушения объекта в точке (2), что и выполняют MFC макросы, выводя данную операцию из зоны ответственности разработчика.

Как правило, обработчики исключений только отображают текстовое сообщение о характере возникшей ошибки. Классы исключений несут информацию об ошибке, но в виде целочисленного кода, поэтому возникает задача преобразования этого кода в его текстуальное представление. Наиболее просто данная задача решается с помощью вызова Win32 API функции **FormatMessage**, так что обработчик файловых исключений мог бы иметь вид:

```
void ViewFileEerror (CFileException* e)
{
    // e->m_lOsError     системной код ошибки
    // e->m_strFileName  имя файла, вызвавшего ошибку

    // буфер для текстового представления ошибки
    TCHAR szErrs[4*1024];

    // получение текста ошибки по ее коду
    ::FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM,
             NULL, e->m_lOsError,
             MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
             szErrs, sizeof(szErrs), NULL);
    .
    .
    .
}
```

15.18. Классы, не имеющие базового класса

Кроме классов, наследованных от базового класса **CObject**, библиотека MFC включает ряд самостоятельных классов. У них нет общего базового класса, и все они имеют различное назначение. Несколько классов, которые не наследуются от базового класса **CObject**, уже упоминались – это классы **CCmdUI**, **CFileStatus** и **CDataExchange**. Среди других укажем наиболее интересные.

Простые геометрические фигуры.

CPoint объекты класса описывают точку.

CRect объекты класса описывают прямоугольник.

CSize объекты класса определяют размер прямоугольной области.

Дату и время определяют два класса:

CTime объекты класса служат для хранения даты и времени. Большое количество методов класса позволяет выполнять различные преобразования.

CTimeSpan объекты класса определяют период времени.

Строка. Объекты класса **CString** представляют собой текстовые строки переменной длины. Этот класс интересен еще и тем, что имеет метод **GetBuffer**, который дает возможность захватывать блоки памяти произвольной длины. Данный путь предпочтительнее, поскольку не приводит к столь сильной фрагментации CRT "кучи", как многократное использование оператора **new**.

Архивный класс. Класс **CArchive** используется для сохранения и восстановления состояния объектов в файлах на магнитном носителе. Перед использованием объекта класса **CArchive** он должен быть привязан к объекту класса **CFile**.

Информация о классе объекта. Во многих случаях бывает необходимо уже во время выполнения приложения получить информацию о классе и его базовом классе. Для этого любой класс, наследованный от базового класса **CObject**, связан со структурой **CRuntimeClass**. Она позволяет определить имя класса объекта, размер объекта в байтах, получить указатель на конструктор по умолчанию и деструктор класса. Можно также узнать подобную информацию о базовом классе и некоторые дополнительные сведения.

Отладка приложения. В отладочной версии приложения можно использовать класс **CDumpContext**. Он позволяет выдавать состояние различных объектов в текстовом виде. Класс **CMemoryState** позволяет локализовать проблемы, связанные с динамическим выделением оперативной памяти. Такие проблемы обычно возникают, когда пользователь выделяет память оператором **new**, а затем забывает вернуть эту память операционной системе.

Печать документа. Класс **CPrintInfo** предназначен для управления печатью документов на принтере. Когда пользователь отправляет документ на печать или выполняет предварительный просмотр документа перед печатью, создается объект класса **CPrintInfo**. Он содержит различную информацию о диапазоне печати, о том, какие страницы документа печатаются в данный момент и проч.

Итак, в данном разделе проведен обзор основных классов, с помощью которых строятся типичные MFC приложения. Информацию о других MFC классах вы можете найти в справочной литературе.

16. Структура простейших MFC приложений

Самые простейшие приложения с использованием библиотеки классов MFC можно создавать в Microsoft Developer Studio без применения автоматизированных средств разработки приложений MFC AppWizard. Необходимо только создать проект типа Win32 Application и включить в его установках поддержку библиотеки MFC.

16.1. Приложение без главного окна

С целью изучения внутренней структуры типичного MFC приложения, создадим приложение **Simple1**, отображающее на экране диалоговую панель, которая содержит строку текста. В этом приложении используется единственный класс, наследованный от базового класса **CWinApp**. Приведем исходный текст приложения:

```
        // включаем поддержку MFC
(1)     #include <afxwin.h>
        class CSimple1App: public CWinApp {
        public:
        // переопределяем метод InitInstance
(3)     virtual BOOL InitInstance();
        };

        // создаем объект приложения
(2)     CSimple1App theApp;

        // Метод InitInstance
        BOOL CSimple1App::InitInstance()
        {
                AfxMessageBox("Simple1 MFC application");
(4)             return FALSE;
        }
```

Рассмотрим, как работает это приложение.

(1) В текст приложения включается файл **afxwin.h**. В этом файле определены классы, методы, константы и другие структуры для библиотеки классов MFC. Кроме того, этот включаемый файл автоматически подключает другой файл – **windows.h**, необходимый для вызовов функций стандартного программного интерфейса Windows.

(2) В исходном тексте программы нет хорошо знакомой функции **WinMain**. Не видно также переменных, представляющих параметры этой функции. В приложениях, основанных на классах MFC, функция **WinMain** скрыта от программиста в определении класса **CWinApp**. В каждом приложении определяется главный класс приложения, наследуемый от базового класса **CWinApp**. Приложение должно иметь *только один объект* главного

класса приложения, наследованного от класса **CWinApp**. После создания глобального объекта **theApp** вызывается функция **WinMain**, определенная в классе **CWinApp**. Она выполняет свои обычные функции: регистрирует классы окон, создает главное окно приложения и проч.

Класс **CWinApp** выполняет все действия, которые обычно выполняет функция **WinMain**, – инициализирует приложение, обрабатывает сообщения и завершает приложение. Для этого класс **CWinApp** включает виртуальные методы **InitApplication**, **InitInstance**, **Run** и **ExitInstance**. Именно в такой последовательности они и вызываются каркасом приложения.

Метод **InitApplication** выполняет инициализацию на уровне приложения. Хотя вы и имеете возможность переопределить этот метод в своем приложении, но, как правило, он остается без изменения.

(3) Метод **InitInstance** выполняет инициализацию каждой копии приложения. Этот метод создает главное окно приложения. В отличие от остальных методов класса **CWinApp**, разработчик *обязан переопределить* этот метод в своем приложении.

(4) Если Метод **InitInstance** возвращает значение **FALSE**, приложение сразу же завершается; если же метод **InitInsatnce** вернет значение **TRUE**, приложение продолжит свою работу и приступит к обработке очереди сообщений.

Метод **Run** вызывается каркасом для обработки цикла сообщений. Если в этом есть необходимость, вы можете переопределить данный метод, чтобы реализовать свой собственный цикл обработки сообщений.

Метод **ExitInstance** вызывается каркасом, когда приложение заканчивает работу, то есть при завершении цикла обработки сообщений. Вы может переопределить этот метод, чтобы выполнить какие-либо действия перед завершением приложения.

16.2. Приложение с главным окном

Приложение **Simple1** не имело главного окна. Рассмотрим другое приложение **Simple2**, которое при запуске отображает на экране компьютера обычное окно, имеющее заголовок, системное меню и кнопки управления.

В приложении **Simple2** также используется класс **CWinApp** в качестве главного класса приложения. Для представления главного окна приложения создается еще один класс, наследуемый от базового класса **CFrameWnd**.

```
// поддержка MFC
#include <afxwin.h>
```

```
class CSimple2App: public CWinApp {
public:
virtual BOOL InitInstance();
};

// Класс главного окна приложения
class CMainWindow : public CFrameWnd {
public:
      CMainWindow();

};
CMainWindow::CMainWindow()
{
      // Создание окна приложения
(1)   Create(NULL,"Hello");
}

BOOL CSimple2App::InitInstance()
{
      // Создание объекта класса CMainWindow
(2)   m_pMainWnd= new CMainWindow;
      // Отображение окна на экране
      m_pMainWnd->ShowWindow(m_nCmdShow);
      // Обновление содержимого окна
(3)   m_pMainWnd->UpdateWindow();
      return TRUE;
}

CSimple2App theApp;
```

Приложение **Simple2** состоит из одного главного окна и не содержит ни меню, ни каких-либо других органов управления. Тем не менее, главное окно приложения обладает всеми возможностями Windows окон. Оно имеет заголовок, системное меню и кнопки управления. Можно изменить размер этого окна, увеличить его на весь экран и уменьшить до размера пиктограммы.

(1) В нашем примере главное окно приложения создается в конструкторе класса **CMainWindow**. Имейте ввиду, что этот прием нельзя назвать типичным, здесь он применяется только в демонстрационных целях.

(2) В данном приложении метод **InitInstance** используется для создания и отображения на экране окна приложения. Для этого динамически создается объект класса **CMainWindow**, а указатель на этот объект сохраняется в элементе данных **m_pMainWnd** класса **CWinThread**. Напомним, что этот класс является базовым для класса **CWinApp**.

> Таким образом, производится увязка объекта приложения и объекта главного окна приложения.

Для создания объекта класса **CMainWindow** используется оператор **new**. Он создает объект указанного класса, отводит память и возвращает

указатель на него. Следует напомнить, что при создании нового объекта с помощью оператора **new** для него автоматически вызывается конструктор.

> В приведенном коде отсутствует разрушение объекта класса **CMainWindow**. Поразмышляете, как происходит данное разрушение, если с помощью оператора **delete**, то где он должен располагаться?

(3) Само окно появится на экране только после того, как будет вызван метод **ShowWindow**. В качестве параметра методу **ShowWindow** передается значение **m_nCmdShow**. Переменная **m_nCmdShow** является элементом класса **CWinApp**. Этот параметр соответствует параметру функции **WinMain**, который определяет, как должно отображаться главное окно приложения сразу после его запуска. После появления окна на экране ему передается сообщение **WM_PAINT** вызовом метода **UpdateWindow**, что требует обновления содержимого главного окна.

Метод **InitInstance** возвращает значение **TRUE**; это означает, что инициализация приложения завершилась успешно и можно приступать к обработке очереди сообщений.

> Для создания окна приложения создается объект класса **CMainWindow**. Этот объект *не есть окно*, которое пользователь видит на экране компьютера. Это внутреннее представление окна с помощью MFC. Для создания окна предназначается метод **Create**, определенный в классе **CFrameWnd**. Этот метод создает Windows окно и *связывает* его с объектом C++, в нашем случае – с объектом класса **CMainWindow**.

16.3. Обработка сообщений

Как известно, работа операционной системы Windows основана на обработке сообщений. Сообщения сначала попадают в системную очередь сообщений операционной системы. Из нее сообщения передаются приложениям, которым они предназначены, и записываются в очередь сообщений приложений. Каждое приложение имеет собственную очередь сообщений.

Приложение в цикле обработки сообщений получает их из очереди приложения и направляет соответствующей функции окна, которая и выполняет обработку сообщения.

Цикл обработки сообщений в традиционном Windows приложении обычно состоит из оператора **while**, в котором циклически вызываются функции **GetMessage** и **DispatchMessage**. Для более сложных приложений цикл обработки сообщений содержит вызовы других функций,

например **TranslateMessage**, которые обеспечивают предварительную обработку сообщений.

Каждое окно приложения имеет собственную функцию окна. В процессе обработки сообщения операционная система вызывает функцию окна и передает ей структуру, описывающую очередное сообщение. Функция обработки сообщения проверяет, какое именно сообщение поступило для обработки, и выполняет соответствующие действия.

В MFC приложениях за обработку сообщений отвечают классы. Любой класс, наследованный от базового класса **CCmdTarget**, может обрабатывать сообщения. Чтобы класс смог обрабатывать сообщения, необходимо, чтобы он имел таблицу сообщений класса. В этой таблице для каждого сообщения указан метод класса, предназначенный для его обработки.

Прежде чем рассматривать эту таблицу укажем группы MFC сообщений.

Сообщения, которые могут обрабатываться MFC приложением делятся на три группы.

16.3.1. Оконные сообщения

Эта группа включает сообщения, предназначенные для обработки функцией окна. Практически все сообщения, идентификаторы которых начинаются префиксом **WM_**, за исключением **WM_COMMAND**, относятся к этой группе.

Оконные сообщения предназначаются для обработки объектами, представляющими окна. Это могут быть любые объекты, наследованные от класса **CWnd**. Характерной чертой этих классов является то, что они включают идентификатор окна.

16.3.2. Сообщения от элементов управления

Эта группа включает в себя сообщения **WM_COMMAND**, передаваемые от дочерних окон своему родительскому окну. Сообщения от элементов управления обрабатываются точно таким же образом, что и оконные сообщения.

Исключение составляет сообщение **WM_COMMAND** с кодом извещения **BN_CLICKED**. Напомним, что это сообщение передается при нажатии кнопки. Обработка сообщений с кодом извещения **BN_CLICKED** от элементов управления происходит аналогично обработке командных сообщений.

16.3.3. Командные сообщения

Это сообщения **WM_COMMAND** от меню, кнопок панели управления и клавиш акселераторов. В отличие от оконных сообщений и сообщений от элементов управления, командные сообщения могут быть обработаны более

широким спектром объектов. Эти сообщения обрабатывают не только объекты, представляющие окна, но также объекты классов, представляющих приложение, документы или шаблон документов.

Характерной особенностью командных сообщений является идентификатор. Идентификатор командного сообщения определяет объект, который вырабатывает данное сообщение.

16.3.4. Таблица сообщений

В библиотеке классов MFC для обработки сообщений используется специальный механизм, который имеет название Message Map – таблица сообщений.

Таблица сообщений состоит из набора специальных макрокоманд, ограниченных макрокомандами **BEGIN_MESSAGE_MAP** и **END_MESSAGE_MAP**. Между ними расположены макрокоманды, отвечающие за обработку отдельных сообщений:

```
BEGIN_MESSAGE_MAP(ИмяКласса, ИмяБазовогоКласса)
// макросы
END_MESSAGE_MAP()
```

> Встает вопрос, зачем нужно использовать такие громоздкие макросы, когда в языке C++ существуют виртуальные функции? Действительно, применение виртуальных функций решает ту же задачу – однозначного определения методов в цепочке производных классов. Однако хранение в коде больших таблиц виртуальных функций привело бы как к неразумному “раздуванию” объема приложения, так и к снижению скорости работы MFC приложений. Но, как мы увидим позже, идея использования виртуальных функций для обработки сообщений не была полностью отброшена. Она применяется, например, в диалогах.

Макрокоманда **BEGIN_MESSAGE_MAP** представляет собой заголовок таблицы сообщений. Она имеет два параметра. Первый параметр содержит имя класса таблицы сообщений. Второй – указывает его базовый класс.

Если в таблице сообщений класса отсутствует обработчик для сообщения, оно передается для обработки базовому классу. Если таблица сообщений базового класса также не содержит обработчик этого сообщения, оно передается следующему базовому классу и так далее.

Если ни один из базовых классов не может обработать сообщение, выполняется обработка по умолчанию, которая зависит от типа сообщения:

1) стандартные сообщения Windows обрабатываются функцией обработки по умолчанию **DefWindowProc**;
2) командные сообщения передаются по цепочке следующему объекту, который может обработать командное сообщение.

Можно определить таблицу сообщений класса вручную, однако более удобной путь – это воспользоваться средствами **ClassWizard**, который может сгенерировать необходимый макрос для выбранного сообщения, а также создать шаблон функции обработчика. Разработчику остается только дополнить этот шаблон необходимым кодом.

16.3.5. Обработчики сообщений

Рассмотрим макрокоманды, отвечающие за обработку различных типов сообщений.

ON_WM_<name>

Макрокоманда обрабатывает стандартные сообщения операционной системы Windows. Вместо **<name>** указывается имя сообщения без префикса **WM_**. Например, для сообщения **WM_CREATE** в карте сообщений появится строка

```
ON_WM_CREATE(),
```

что соответствует стандартному обработчику **OnCreate()**.

Для обработки сообщений, определенных в таблице макрокомандой **ON_WM_<name>**, вызываются одноименные методы. Имя метода обработчика соответствует названию сообщения, без учета префикса **WM_**. В классе **CWnd** определены обработчики для большинства стандартных сообщений. Эти обработчики будут использоваться по умолчанию, если вы не определите их в своем производном классе.

Макрокоманды **ON_WM_<name>** не имеют параметров. Однако методы, которые вызываются для обработки соответствующих сообщений, могут иметь параметры, количество и назначение которых зависит от обрабатываемого сообщения.

Если определить обработчик стандартного сообщения Window в своем классе, то он будет вызываться вместо обработчика, определенного в классе **CWnd** или другом базовом классе. В любом случае можно вызвать метод-обработчик базового класса из своего метода обработчика, используя операцию разрешения видимости, например:

```
void MyClass::OnCreate()
{
    .
    .
    CWnd::OnCreate();
    .
    .
}
```

ON_COMMAND

Эта макрокоманда предназначена для обработки командных сообщений. Командные сообщения поступают от меню, кнопок панели управления и клавиш акселераторов. Характерной особенностью командных сообщений является то, что с ними связан идентификатор сообщения.

Макрокоманда **ON_COMMAND** имеет два параметра. Первый параметр соответствует идентификатору командного сообщения, а второй – имени метода, предназначенного для обработки этого сообщения. Таблица сообщений должна содержать не больше одной макрокоманды для одного и того же командного сообщения, например, такая ситуация недопустима:

```
ON_COMMAND(ID_ITEM, OnItemOne)
ON_COMMAND(ID_ITEM, OnItemTwo) // неверно!
```

Обычно командные сообщения не имеют обработчиков, используемых по умолчанию. Существует только небольшая группа стандартных командных сообщений, имеющих методы обработчики, вызываемые по умолчанию. Эти сообщения соответствуют стандартным строкам меню приложение. Так, например, если строке Open меню File присвоить идентификатор **ID_FILE_OPEN**, то для его обработки будет вызван метод **OnFileOpen**, определенный в классе **CWinApp**.

ON_COMMAND_RANGE

Данная макрокоманда применяется в тех случаях, когда один и тот же метод класса применяется для обработки сразу нескольких командных сообщений с различными идентификаторами. В этом случае идентификаторы должны находится в диапазоне от **id1** до **id2**, где **id2>id1**.

Эта макрокоманда не поддерживается **ClassWizard**, поэтому все действия по ее применению выполняются вручную. Предположим, что метод **OnItemAll** должен обрабатывать все командные сообщения с идентификаторами от **ID_01** до **ID_20**, тогда:

1) в карту сообщений добавляется строка вида

```
ON_COMMAND_RANGE(ID_01, ID_20, OnItemAll)
```

2) создается обработчик вида

```
void MyClass::OnItemAll(UINT nID)
{
    .
    // значение формального параметра
    // nID находится в диапазоне
    // ID_01 <= nID <= ID_20
    .
}
```

ON_UPDATE_COMMAND_UI

Данная макрокоманда обрабатывает сообщения, предназначенные для обновления пользовательского интерфейса – пунктов меню и кнопок панели управления. В общем случае макрос имеет вид:

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

Как и для других макрокоманд, параметр **id** указывает идентификатор сообщения, а параметр **memberFn** – имя метода для его обработки:

Методы, предназначенные для обработки данного класса сообщений, должны быть определены с ключевым словом **afx_msg** и иметь один параметр – указатель на объект класса **CCmdUI**. Как правило, имена методов, предназначенных для обновления пользовательского интерфейса, начинаются с символов **OnUpdate..**, например:

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

В качестве параметра **pCmdUI** методу передается указатель на объект класса **CCmdUI**, который содержит информацию о требующем обновления объекте пользовательского интерфейса. Класс **CCmdUI** включает методы, позволяющие изменить внешний вид представленного им объекта пользовательского интерфейса. Так, например, показанный ниже обработчик делает доступным пункт меню с идентификатором **ID_TEST** только тогда, когда массив **m_aItem** имеет один или более элементов:

```
void MyClass::OnUpdateTest(CCmdUI* pCmdUI)
{
    pCmdUI->Enable(m_aItem.GetSize());
}
```

Сообщения, предназначенные для обновления пользовательского интерфейса, передаются в двух случаях:

1) перед тем, как **POPUP** меню появляется на экране;

2) во время цикла простоя, т.е. когда очередь сообщений приложения становится пуста. В этом случае сообщения **OnUpdate** вызываются для каждого пункта меню и для каждой кнопки панели управления, если она не дублирует какой-либо пункт меню.

> Если не определить метод для обновления какой-либо строки меню или кнопки панели управления, то выполняется обработка по умолчанию. Она заключается в том, что каркас приложения выполняет поиск обработчика соответствующего командного сообщения во всех классах, и, если таковой не обнаружен, выбор строки запрещается.

ON_UPDATE_COMMAND_UI_RANGE

Эта макрокоманда является разновидностью предыдущей и обеспечивает обработку сообщений, предназначенных для обновления пользователь-

ского интерфейса, идентификаторы которых лежат в интервале от **id1** до **id2**. Параметр **memberFn** указывает метод, используемый для обработки:

```
ON_UPDATE_COMMAND_UI_RANGE(id1,id2,memberFn)
```

ON_<name>

Данные макрокоманды предназначены для обработки сообщений от элементов управления. Такие сообщения могут передаваться органами управления диалоговой панели.

> Сообщения от элементов управления не имеют обработчиков, используемых по умолчанию.

Все макрокоманды **ON_<name>** включают два параметра. В первом параметре **id** указывается идентификатор элемента управления. Сообщения от этого элемента будут обрабатываться методом, указанным вторым параметром. Например, если карта сообщений содержит строку

```
ON_BN_CLICKED(IDC_CLEAR,OnClear)
```

это означает, что при нажатии кнопки с идентификатором **IDC_CLEAR** будет вызван обработчик **OnClear()**.

ON_CONTROL_RANGE

Эта макрокоманда является расширенным вариантом предыдущего макроса и обрабатывает сообщения от элементов управления, идентификаторы которых находятся в интервале от **id1** до **id2**, кроме того, макрокоманда имеет дополнительный параметр **wNotifyCode**, который несет код извещения:

```
ON_CONTROL_RANGE(wNotifyCode,id1,id2,memberFn)
```

Обработчик данного сообщения **memberFn** имеет один формальный параметр, который определяет идентификатор элемента управления во время вызова. Так строка карты сообщений

```
ON_CONTROL_RANGE(BN_CLICKED,IDC_01,IDC_20,OnClicked)
```

означает, что нажатие кнопок (код – **BN_CLICKED**) с идентификаторами от **IDC_01** до **IDC_20** обрабатывается одним методом **OnClicked** вида:

```
void MyClass::OnClicked(UINT nID)
{
    .
    // значение формального параметра
    // nID находится в диапазоне
    // ID_01 <= nID <= ID_20
    .
}
```

ON_MESSAGE

Данная макрокоманда обрабатывает сообщения, определенные пользователем. Идентификатор сообщения, который должен превышать значение **WM_USER**, указывается параметром **idMessage**, а метод, который вызывается для обработки сообщения, указывается параметром **memberFn**:

```
ON_MESSAGE(idMessage,memberFn)
```

Эта макрокоманда не поддерживается **ClassWizard**, поэтому все действия по ее применению выполняются вручную по следующей схеме:

```
        // Заголовочный файл.

        // Идентификатор собственного сообщения
(1)     #define  um_NEED_DATA  (WM_USER+100)

        // Объявляем класс CDlgList,
        // наследуемый от CDialog.

        //{{AFX_MSG(CDlgList)

        // Прототипы, сгенерированные ClassWizard

        //}}AFX_MSG
        // Прототипы для собственных обработчиков
(2)     afx_msg LONG OnNeedData (WPARAM, LPARAM);
        DECLARE_MESSAGE_MAP()

        // Файл реализации.
        BEGIN_MESSAGE_MAP(CDlgList, CDialog)
              //{{AFX_MSG_MAP(CDlgList)
              // Макросы ClassWizard
              //}}AFX_MSG_MAP
(3)           ON_MESSAGE(um_NEED_DATA, OnNeedData)
        END_MESSAGE_MAP()

(4)     LONG CDlgList::OnNeedData(WPARAM wParam,
                                  LPARAM lParam)
        {
            // ваш код
            return 0;
        }
```

(1) В заголовочном файле через **#define** определяется идентификатор вашего сообщения, например, **um_NEED_DATA**.

(2) Здесь же, после прототипов **ClassWizard**, нужно указать прототип для обработчика сообщения **um_NEED_DATA**. Обратите внимание на список формальных параметров и сравните с параметрами обычной функции окна.

Интересен и тип возвращаемого значения в сравнении с остальными обработчиками. Почему **LONG**, а не **void**, как для других методов?

(3) В таблице указывается сообщение и обработчик для него, также после всех макросов **ClassWizard**.

(4) И наконец, определяется тело самого обработчика.

ON_REGISTERED_MESSAGE

Эта макрокоманда похожа на предыдущую, с той разницей, что она обслуживает сообщения операционной системы Windows, зарегистрированные с помощью Win32 API функции **RegisterWindowMessage**. Параметр **idMessage** этой макрокоманды указывает на полученный идентификатор сообщения, для которого будет вызываться метод **memberFn**:

```
ON_REGISTERED_MESSAGE(idMessage, memberFn)
```

Макрокоманда также не поддерживается **ClassWizard**, поэтому все действия по ее применению выполняются вручную по схеме, аналогичной для сообщения **ON_MESSAGE**.

Итак, сформулируем основные выводы данного раздела:

1. Чтобы объекты производного класса могли обрабатывать сообщения, в объявлении такого класса необходимо поместить макрокоманду **DECLARE_MESSAGE_MAP**.

2. Необходимо определить таблицу сообщений. Таблица начинается макрокомандой **BEGIN_MESSAGE_MAP** и заканчивается макрокомандой **END_MESSAGE_MAP**. Между этими макросами расположены строки таблицы, определяющие те сообщения, которые будут обработаны данным классом. Здесь же указываются методы, выполняющие такую обработку.

3. Приложение может содержать несколько классов, обладающих собственными таблицами сообщений. Чтобы однозначно определить класс, к которому относится таблица сообщений, имя этого класса записывается первым параметром макрокоманды **BEGIN_MESSAGE_MAP**.

4. Классу может поступать гораздо больше сообщений, чем перечислено в его карте. Необработанные сообщения передаются базовому классу, который указывается вторым параметром макроса **BEGIN_MESSAGE_MAP**.

5. Таблицу сообщений может иметь не только класс окна, но и класс приложения (документа). Следовательно, возможна ситуация, когда некоторые командные сообщения могут обрабатываться как окном, так и приложением (документом). В этом случае реализуется следующая очередность обработки. Те команды, которые не имеют обработчика в таблице сообщений класса окна, передаются для обработки в класс приложения (документа).

Если же команда может быть обработана и в классе окна, и в классе приложения (документа), она обрабатывается только один раз в классе окна.

17. Диалоговые панели в MFC приложениях

Диалоговые панели весьма популярны у Windows разработчиков, поскольку они являются наиболее удобным средством организации взаимодействия с пользователем. Некоторые приложения могут успешно работать даже без главного окна, базируясь только на диалоговых окнах.

Библиотека классов MFC содержит класс **`CDialog`**, специально предназначенный для управления диалоговыми панелями. Напомним, что диалоговые панели бывают двух типов – модальные и немодальные.

Активизация модальных диалоговых панелей приводит к блокировке как родительского окна, так и всех его дочерних окон. Пользователь не может продолжить работу с приложением, пока не закроет модальную диалоговую панель.

Немодальные диалоговые панели не блокируют работу остальных окон приложения. Поэтому, открыв такую панель, можно продолжать работать с приложением, в частности, использовать пункты меню, открывать другие дочерние окна и другие немодальные диалоговые панели.

И модальные, и немодальные диалоговые панели обслуживаются одним MFC классом **`CDialog`**, который наследуется от класса **`CWnd`**.

17.1. Этапы создания диалоговых панелей

Для создания и отображения необходимо произвести следующие действия.

1) В файл ресурсов приложения необходимо добавить шаблон новой диалоговой панели и при помощи редактора ресурсов изменить его, согласно поставленной задачи.

2) Нужно создать класс для управления диалоговой панелью. Этот класс наследуется непосредственно от базового класса **`CDialog`** и может быть сгенерирован с помощью **`ClassWizard`**.

3) Каждая диалоговая панель обычно содержит несколько элементов управления. Работая с диалоговой панелью, пользователь взаимодействует с этими элементами управления: нажимает кнопки, вводит текст, выбирает элементы списков. В результате генерируются соответствующие сообщения, которые должны быть обработаны классом диалоговой панели. Следовательно, класс диалоговой панели должен содержать таблицу сообщений и соответствующие обработчики для этих сообщений.

4) Последним шагом является отображение модальных диалоговых панелей. Напомним, что способ отображения модальных и немодальных диалоговых панелей различен.

17.2. Модальная диалоговая панель

17.2.1. Шаблон диалога

Как было отмечено, первым этапом работы с диалоговыми панелями является создание в файле ресурсов приложения шаблона новой диалоговой панели с каким-либо идентификатором, например **IDD_DIALOG1**. Исходя из ознакомительных целей, добавим только хорошо известное поле редактирования с идентификатором **IDC_EDIT1** и кнопку с идентификатором **IDC_CLEAR**. Заметим. что кнопки **OK** и **Cancel** присутствуют на панели по умолчанию.

Затем при помощи **ClassWizard** создадим класс диалога, например **CDlg**, производный от базового класса **Cdialog**. **ClassWizard** предложит сохранить объявление и реализацию этого класса в файлах с расширениями CDlg.cpp и CDlg.h.

При помощи инструментария ClassWizard добавим в класс диалоговой панели заготовки обработчиков оконных сообщений и сообщений от элементов управления. Например, обработчики нажатий кнопок **OK**, **Cancel** и **Clear**. Кроме того, для обеспечения процесса обмена данными нужно связать элементы управления с переменными класса, например поле редактирования (идентификатор **IDC_EDIT1**) с переменной **m_szEdit** класса **CString**.

17.2.2. Класс диалоговой панели и его реализация

В итоге выполнения перечисленных действий по созданию класса диалога **ClassWizard** сгенерирует файл CDlg.h вида:

```
      #include "resource.h"
(1)   class CDlg : public CDialog {
      public:
(2)   CDlg(CWnd* pParent = NULL);
      //{{AFX_DATA(CDlg)
(3)   enum { IDD = IDD_DIALOG1};
      CString m_szEdit;
      //}}AFX_DATA
      protected:
(3)   virtual void DoDataExchange(CDataExchange* pDX);
      //{{AFX_MSG(CDlg)
(4)   virtual void OnCancel();
      virtual void OnOK();
      virtual BOOL OnInitDialog();
      afx_msg void OnClear();
      //}}AFX_MSG
      DECLARE_MESSAGE_MAP()
};
```

(1) Класс **CDlg** наследуется от базового класса **CDialog**. Конструктор класса (2) имеет один необязательный параметр **pParent**, используемый для передачи указателя родительского окна, которое будет заблокировано при активизации диалога. Непосредственно после объявления конструктора класса следует объявление элементов данных класса (3), которые добавляются **ClassWizard**'ом, поэтому не рекомендуется вручную изменять код приложения, расположенный между комментариями **AFX_DATA**.

> Все ваши изменения будут затерты при следующем использовании средств **ClassWizard** для данного класса.

(4) Добавляются также объявления переопределенных виртуальных функций базового класса, а именно, методы **DoDataExchange**, **OnOK**, **OnCancel** и **OnInitDialog**.

Теперь рассмотрим файл CDlg.cpp:

```
#include <afxwin.h>
#include "resource.h"
#include "Cdlg.h"
BEGIN_MESSAGE_MAP(CDlg, Cdialog)
//{{AFX_MSG_MAP(CDlg)
  ON_BN_CLICKED(IDC_CLEAR,OnClear)
//}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

Класс диалоговой панели обрабатывает сообщения от своих элементов управления, поэтому он должен иметь таблицу сообщений. В заголовке таблицы указывается имя класса **CDlg** и имя базового класса **CDialog**. Карта сообщений класса **CDlg** содержит только одну строку, в которой указан макрос для обработки нажатия кнопки **IDC_CLEAR**.

Две другие кнопки панели **OK** и **Cancel**, точнее кнопки с идентификаторами **IDOK** и **IDCANCEL**, являются стандартными для MFC. В классе диалога **CDlg** определены *виртуальные, а не обычные методы* **OnOK** и **OnCancel**, которые вызываются при нажатии данных кнопок, поэтому обработчики для них не представлены в таблице сообщений. Виртуальные методы **OnOK** и **OnCancel** полностью определены в базовом классе **CDialog**; поэтому для модального диалога нет особой необходимости создавать собственные обработчики, если только вам не потребуется какая-то специализированная обработка информации.

> Для немодального диалога виртуальные методы **OnOK** и **OnCancel** должны быть переопределены *всегда*.

Конструктор класса **CDlg**.

```
CDlg::CDlg(CWnd* pParent /*=NULL*/) :
       CDialog(CDlg::IDD, pParent)
{
    //{{AFX_DATA_INIT(CDlg)
    m_szEdit = "";
    //}}AFX_DATA_INIT
}
```

Основное назначение конструктора **CDlg** – вызвать конструктор базового класса. Именно конструктор класса **CDialog** выполняет создание диалоговой панели по заданному шаблону. После названия конструктора стоит двоеточие и название конструктора базового класса. При этом в качестве параметров ему передается идентификатор диалоговой панели и указатель на родительское окно.

> Заметим, что этот указатель не используется в коде конструктора базового класса, поэтому его можно положить равным **NULL**.

В теле конструктора расположен блок **AFX_DATA_INIT**. Он предназначен для добавления **ClassWizard**'ом кода инициализации элементов данных класса **CDlg**. В нашем случае пустой строкой инициализируется переменная **m_szEdit**, входящая в класс **CDlg**. Напомним, что с этой переменной связано поле редактирования с идентификатором **IDC_EDIT1**. При помощи такой связи реализуется процесс обмена данными между элементами управления и переменными класса.

Обработка сообщения **WM_INITDIALOG**

```
CDlg::OnInitDialog()
{
    CDialog::OnInitDialog();
    // ваша инициализация
    return TRUE;
}
```

Как вы уже заметили, у разработчика нет непосредственного доступа к функции диалога, эта функция скрыта в недрах каркаса приложения. Таблица сообщений класса **CDlg** также не содержит макрокоманд для сообщения **WM_INITDIALOG**, но в классе **CDlg** (как и в базовом классе **CDialog**) объявлен виртуальный метод **OnInitDialog**. Именно он вызывается каркасом приложения в ответ на сообщение **WM_INITDIALOG**, непосредственно перед выводом панели на экран.

> Обратите внимание на то, что собственная инициализация элементов управления должна быть выполнена *после* вызова метода базового класса.

Метод **OnInitDialog** возвращает значение **TRUE**. Это означает, что фокус ввода будет установлен на первый элемент управления диалоговой панели, имеющий стиль **WS_TABSTOP**. Определить такой элемент на диалоговой панели можно, используя средства редактора ресурсов Visual C++.

Если во время инициализации диалоговой панели метод **OnInitDialog** устанавливает фокус ввода на другой элемент управления, метод должен вернуть значение **FALSE**.

> В большинстве случаев переопределять метод **OnInitDialog** не потребуется, если в полной мере используются возможности механизма автоматического обмена данными (см. далее).

Механизм автоматического обмена данными в диалоге

```
void CDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CDlg)
    DDX_Text(pDX, IDC_EDIT1, m_szEdit);
    //}}AFX_DATA_MAP
}
```

Виртуальный метод **DoDataExchange**, который также переопределяется в классе диалоговой панели **CDlg**, первоначально определен в классе **CWnd**, а не **CDialog**. Этот метод служит для реализации механизмов автоматического обмена данными Dialog Data Exchange (**DDX**) и автоматической проверки данных Dialog Data Validation (**DDV**).

Механизм автоматического обмена данными позволяет привязать к элементам управления диалоговой панели переменные или элементы данных класса диалоговой панели. Существует ряд специальных функций, определенных в библиотеке MFC, которые вызываются только методом **DoDataExchange** и выполняют обмен данными между элементами управления диалоговой панели и соответствующими элементами данных класса этой же диалоговой панели. Такой обмен работает в обоих направлениях. Информация из элементов управления диалоговой панели сохраняется в элементах данных класса, и в обратном направлении, информация из элементов данных класса переносится в элементы управления диалоговой панели.

Название всех функций, обеспечивающих такой дуплексный обмен данными, начинаются с префикса **DDX_**. Практически каждый тип элементов управления диалоговой панели имеет собственную функцию для выполнения процедуры обмена данными.

Все **DDX_** функции имеют три параметра. Первый параметр содержит указатель на объект класса **CDataExchange**. Этот объект определяет параметры обмена, в том числе направление, в котором надо выполнить обмен данными. Второй параметр определяет идентификатор элемента управления, с которым выполняется обмен данными. Третий параметр содержит ссылку на элемент данных класса диалоговой панели, связанный с данным органом управления.

Метод **DoDataExchange** позволяет выполнять проверку данных, которые пользователь вводит в диалоговой панели. Для этого предназначен ряд **DDV_** функций. Эти функции позволяют гарантировать, что данные, введенные пользователем в диалоговой панели, соответствуют определенным критериям.

> Если **DDV_** функция используется для проверки ввода для некоторого элемента управления диалоговой панели, то ее необходимо вызвать *сразу же* после вызова **DDX_** функции этого же элемента управления.

Если функция **DDV_** обнаруживает ошибку пользователя при вводе информации в элементе управления, она отображает сообщение и передает фокус ввода на этот элемент.

В отличие от функций **DDX_**, функции **DDV_** в зависимости от их предназначения имеют различное количество параметров. Первый параметр, как и в случае **DDX_**, содержит указатель на объект класса **CDataExchange**. Остальные параметры имеют различное назначение в зависимости от типа функции проверки.

Приложение не должно напрямую вызывать метод **DoDataExhange**. Для этого существует метод **UpdateData** класса **CWnd**. Параметр этой функции определяет в каком направлении будет происходить обмен данными. Если метод **UpdateData** вызывается с параметром **FALSE**, то выполняется обновление полей диалоговой панели. Другими словами, информация из элементов-данных класса переносится в элементы управления диалоговой панели. Если метод **UpdateData** вызван с параметром **TRUE**, данные перемещаются в обратном направлении: из элементов управления диалоговой панели они копируются в соответствующие элементы данных класса диалоговой панели.

Метод **UpdateData** возвращает ненулевое значение, если обмен данными прошел успешно, и нуль в противном случае. Ошибка при обмене данными может произойти, если данные копируются из полей управления диалоговой панели в элементы класса, и пользователь ввел неправильные значения, отвергнутые процедурой автоматической проверки данных.

Как уже было указано, при создании модальной диалоговой панели вызывается виртуальный метод **OnInitDialog** класса **CDialog**. По

умолчанию **OnInitDialog** вызывает метод **UpdateData** с параметром **FALSE** и выполняет инициализацию элементов управления. Если метод **OnInitDialog** переопределяется в классе диалоговой панели, то сначала необходимо вызвать метод **OnInitDialog** базового класса **CDialog**.

Методу **DoDataExchange**, который служит для реализации механизмов автоматического обмена данными, передается указатель **pDX** на объект класса **CDataExchange**. Этот объект создается, когда инициируется процесс обмена данными вызовом функции **UpdateData**. Элементы данных класса **CDataExchange** определяют процедуру обмена данными, в том числе определяют, в каком направлении будет происходить этот обмен. Следует обратить внимание на то, что указатель **pDX** передается как **DDX_**, так и **DDV_** функциям.

Если к диалоговой панели добавить новые поля управления и связать их средствами **ClassWizard** с элементами данных класса **CDlg**, то в блоке **AFX_DATA_MAP** будут размещены вызовы и других **DDX_** и **DDV_** функций, которые необходимые для корректного выполнения процедуры обмена данными.

Методы-обработчики

```
void CDlg::OnClickedClear()
{
    m_szEdit = "";
    CWnd::UpdateData(FALSE);
}
```

Особенность этого обработчика состоит в том, что после изменения значения одного элемента данных класса **CDlg**, вызывается метод **UpdateData**, который обновляет информацию *во всех* элементах управления диалога, хотя это и не требуется. В этом случае, для повышения эффективности вы можете воспользоваться методом **SetDlgItemText**, также определенном в классе **CWnd**.

```
void CDlg::OnCancel()
{
    // отменяющие действия
    CDialog::OnCancel();
}
void CDlg::OnOK()
{
    // если требуется самостоятельная
    // обработка, то она выполняется здесь
    CDialog::OnOK();
}
```

Метод **OnOK**, определенный в классе **CDialog**, копирует данные из полей диалоговой панели в связанные с ними переменные. Для этого вызывается метод **UpdateData** с параметром **TRUE**. Затем выполняется вызов метода **EndDialog**, который закрывает диалоговую панель и возвращает значение **IDOK**. Именно это значение будет возвращать метод **DoModal**, который используется для создания диалоговой панели.

Метод **OnCancel**, определенный в классе **CDialog**, закрывает диалоговую панель и возвращает значение **IDCANCEL**. При этом копирования данных из элементов управления не происходит.

Поскольку методы **OnOK** и **OnCancel** определены в классе **CDialog** как виртуальные и они переопределяются в классе **CDlg**, то управление получат переопределенные методы, а не методы класса **CDialog**. Для вызова методов базового класса нужно явно указывать имя класса **CDialog**.

17.2.3. Отображение модальной диалоговой панели

Для отображения модальной диалоговой панели сначала создается объект класса **CDlg**, который будет представлять диалоговую панель. Когда объект создан, диалоговая панель еще не появляется на экране, для этого нужно воспользоваться методом **DoModal**, который определен в классе **CDialog**. Возвращение из метода **DoModal** не происходит до тех пор, пока пользователь не закроет диалоговую панель.

Для отображения модальной диалоговой панели должен использоваться один из методов родительского объекта – класса приложения или класса главного окна, а сам код может быть аналогичен следующему:

```
void CMainFrame::OnTest()
{
    // создание объекта класса диалога,
    // вызов конструктора класса диалога
    CDlg dlg;
    // инициализация переменных класса,
    // диалогового окна еще нет
    dlg.m_szEdit = "Просто текст";
    // запуск диалога
    int rc = dlg.DoModal();
    if (rc == IDOK) {
       // пользователь нажал кнопку OK,
       // разрушено диалоговое окно,
       // но не объект dlg класса CDlg
       AfxMessageBox(dlg.m_szEdit);
    }
    else {
       // пользователь нажал кнопку Cancel,
       // разрушено диалоговое окно
    }
```

```
// объект dlg класса CDlg сейчас будет разрушен
}
```

17.3. Немодальная диалоговая панель

Процедура создания немодальной диалоговой панели незначительно отличается от процедуры создания модальной диалоговой панели. В первую очередь создается шаблон диалоговой панели, затем – класс, например **CDlg**, управляющий диалоговой панелью. Этот класс также как и для модальной панели наследуется от базового класса **CDialog**. Обработчики сообщений остаются без изменений, за исключением **OnOK** и **OnCancel**.

1. В теле этих функций нужно убрать вызовы одноименных методов базового класса **CDialog::OnOK()** и **CDialog::OnCancel()**. Дело в том, что они, в свою очередь, обращаются к методу **EndDialog** для завершения модального диалога, что *недопустимо* для немодальной панели. Таким образом, в простейшем случае обработчик **OnOK** будет иметь вид:

```
void CDlg::OnOK()
{
    CWnd::DestroyWindow();
}
```

> Уточняющее имя класса **CWnd** в данном случае не требуется. Здесь подчеркнуто то обстоятельство, что метод **DestroyWindow** размещается в классе **CWnd**, а не в классе **CDialog**.

2. Следующее принципиальное отличие относится к схеме использования немодального диалога. Нельзя конструировать объекты класса **CDlg** на стеке, как это было в случае модального диалога (см. пункт 4.2.3). Вы должны либо использовать динамическое создание объекта, либо, что предпочтительнее, объявить объект **m_dlg** класса **CDlg** элементом данных родительского класса, например класса **CMainFrame**. После этого, метод обработчик **OnTest** для командного сообщения **ID_TEST** мог бы иметь вид:

```
void CMainFrame::OnTest()
{
    if( !m_dlg.GetSafeHwnd() ) {
       m_dlg.Create(m_dlg::IDD, this);
       m_dlg.ShowWindow();
       m_dlg.UpdateWindow();
    }
}
```

Метод **Create** класса **CDialog** создает, а метод **ShowWindow** класса **CWnd** (заметим, что свойство **VISIBLE** в шаблоне диалога *не устанавливается*) отображает окно немодального диалога и немедленно возвращает управление (сравните с методом **DoModal**). Следовательно, если объект

был бы создан на стеке, выход из функции **OnTest** разрушил бы этот объект, тогда как окно только появилось на экране. Такая ситуация, когда есть окно, но нет ассоциированного с ним объекта, в MFC не допустима.

Здесь же показан простой, но эффективный прием, который позволяет блокировать повторное создание немодального диалога, если он уже отображен на экране. Метод **GetSafeHwnd** класса **CWnd** возвращает дескриптор окна, ассоциированного с объектом **m_dlg**, либо **NULL**, если такого окна нет.

3. Последняя особенность использования немодальных диалоговых панелей заключается в организации обмена данными между диалогом и родительским объектом.

Первый путь состоит в передаче диалогу указателя на родительский объект, после чего появляется возможность вызова методов родительского объекта в методах немодального диалога. Однако здесь требуется приведение указателя, поскольку диалог получает указатель на объект класса **CWnd**, а не **CMainFrame**, как в нашем случае.

Обычно используется более универсальная технология, когда родительскому объекту посылается пользовательское сообщение при нажатии в диалоге кнопки **OK** или **Cancel**.

```
#define  um_FROM_DIALOG  WM_USER+144

void CDlg::OnOK()
{
    UpdateData(TRUE);
    GetParent()->PostMessage(um_FROM_DIALOG, IDOK);
}
```

Родительское окно уведомляется о том, что пользователь закончил работу с диалогом, поскольку в своей таблице сообщений имеет макрокоманду (см. пункт 3.3.5)

```
ON_MESSAGE(um_FROM_DIALOG, FromDialog)
```

и обработчик сообщения **um_FROM_DIALOG**:

```
LONG CMainFrame::FromDialog(WPARAM wParam,
                            LPARAM lParam)
{
    // имеем обновленные данные из диалога
    // m_dlg.m_szText

    // закрываем диалог
    m_dlg.DestroyWindow();
    return 0;
}
```

Как легко заметить, в этом подходе родительский объект и создает, и разрушает окно немодального диалога.

18. Поддержка MFC новых элементов управления

MFC поддерживает все элементы управления, пришедшие из Win16, через такие классы, как **CButton**, **CComboBox**, **CEdit**, **CListBox** и **CStatic**. Мы довольно подробно разбирали особенности работы с этими элементами в работе [6]. В MFC учитывается, что все перечисленные элементы управления "общаются" с родительским окном посредством посылки сообщения WM_COMMAND с соответствующим нотификацияным кодом. Эти элементы по-прежнему имеют ряд жестких ограничений, унаследованных из Win16. Так, элемент управления **EditBox**, и соответственно класс **CEdit**, не позволяют разместить в данном элементе текст, превосходящий 32К для Windows 95/98/ME. Вы по-прежнему можете использовать эти элементы, однако в арсенале Win32 появились новые органы управления, а MFC осуществляет их полную поддержку. Заметим, что все элементы являются окнами и, следовательно, наследуются от **CWnd** класса.

Элемент управления	Краткое описание
CRichEditCtrl	расширяет возможности **Edit**
CListCtrl	расширяет возможности **ListBox**
CTreeCtrl	окно для отображения иерархической структуры
CImageList	массив изображений icon или bitmap; обычно используется совместно с **CListCtrl** и **CTreeCtrl**
CHeaderCtrl	заголовок изменяемого размера; может иметь вид кнопок; используется совместно с **CListCtrl**
CSpinButtonCtrl	наборный счетчик; используется совместно с **CEdit**
CSliderCtrl	ползунок для выбора фиксированных значений параметра
CToolTipCtrl	окно подсказки
CProgressCtrl	окно-индикатор для визуализации протекания длительных процессов
CAnimateCtrl	"проигрывает" в своем окне файл AVI формата

Все новые элементы управления посылают родительскому окну уведомляющее сообщение **WM_NOTIFY**. В общем случае **WPARAM** данного сообщения является идентификатором элемента управления, а **LPARAM** – это указатель на структуру **NMHDR**, которая имеет следующие поля:

```
HWND hwndFrom;     дескриптор окна
UINT idFrom;       идентификатор
UINT code;         код уведомления
```

Однако каждый элемент посылает родительскому окну расширенную или дополнительную информацию. В этом случае параметр **LPARAM** указывает на другие структуры, зависящие от конкретного элемента управления и от типа конкретного сообщения. Таких уведомляющих структур в Win32 достаточно много, но все они объединены тем, что первым полем для всех является структура **NMHDR**.

Работа с элементами управления **CSliderCtrl** – ползунок для выбора фиксированных значений параметра; **CToolTipCtrl –** окно подсказки по месту; **CProgressCtrl** – индикатор протекания длительных процессов и **CAnimateCtrl** – "проигрыватель" AVI файлов не вызывает особых затруднений, поэтому они выносится на самостоятельное изучение.

Ознакомление с остальными элементами управления будет проводиться на примерах решения практических задач.

18.1. Элемент CRichEditCtrl

Этот элемент управления – весьма мощное средство, появившееся в арсенале разработчика с приходом Win32. RichEdit позволяет проводить форматирование текста в широком диапазоне, отслеживает местоположение строк-ссылок, выделяя их цветом и меняя форму курсора и проч. В качестве недостатков следует отметить не всегда корректную вставку текста из буфера обмена под Windows 2000. Видимо, это связано с проблемами UNICOD'а для национальных шрифтов и будет исправлено в следующих версиях Windows.

18.1.1. Создание элемента RichEdit

Наиболее широко данный элемент используется для отображения и редактирования информации достаточно большого объема. Данное обстоятельство должно быть учтено выбором соответствующих флагов при создании элемента. Так, типичным набором является следующий:

```
DWORD dwStyle = ES_AUTOVSCROLL | ES_AUTOHSCROLL |
                ES_MULTILINE |
                WS_CHILD | WS_VISIBLE |
                WS_VSCROLL | WS_HSCROLL;

m_rich.Create(dwStyle, rc, pParentWnd, nID);
```

Остальные параметры не вызывают затруднений и соответственно равны: **rc** – прямоугольная область, в которой создается элемент управления; **pParentWnd** – указатель на родительский объект; **nID** – идентификатор RichEdit.

Установка свойства “только для чтения” для RichEdit выполняется динамически вызовом метода **SetReadOnly(fReadOnly)**. Если параметр **fReadOnly** принимает значение **TRUE**, RichEdit переводится в режим, когда редактирование текста становится невозможным.

18.1.2. Загрузка текста в RichEdit

Загрузка текста в элемент управления может выполняться несколькими способами. Здесь мы рассмотрим простейший путь, аналогичный тому, который применяется для EditBox.

В первую очередь следует установить максимально возможную длину текста, который может быть загружен в элемент управления. Эта величина равна максимально возможному значению числа типа **long** за вычетом длины одного параграфа:

```
m_rich.LimitText(0xFFFFFFFF/2-16);
```

Затем выделяется буфер, равный длине считываемого файла, проводится считывание информации и заполнение элемента управления по следующей схеме:

```
// пусть len - длина текстового файла
CString tt;
LPTSTR temp = tt.GetBuffer(len);

// читаем len байт из файла
f.Read(temp, len);

// помещаем текст в RichEdit
m_rich.SetWindowText(temp);
```

Недостаток данного подхода состоит в том, что метод **SetWindowText** класса **CWnd** имеет внутреннее ограничение: он не может обработать текст, если его длина превышает 64К. В этом случае можно воспользоваться методом **StreamIn** класса **CRichEditCtrl**, который осуществляет потоковую загрузку текста произвольной длины.

18.1.3. Форматирование текста

Форматирование текста в RichEdit – это новая возможность в сравнении с элементом EditBox. Непосредственно форматирование выполняется вызовом метода **SetDefaultCharFormat** класса **CRichEditCtrl**, однако перед этим нужно корректно заполнить поля структуры форматирования, например:

```
CHARFORMAT cf;
// устанавливаем длину структуры
// и необходимые маски
cf.cbSize = sizeof(cf);
```

```
cf.dwMask = CFM_CHARSET | CFM_SIZE | CFM_FACE;
// устанавливаем высоту шрифта
cf.yHeight = 10*20;
// используем только русские шрифты
cf.bCharSet = RUSSIAN_CHARSET;
// указываем предпочтительный шрифт
lstrcpy(cf.szFaceName, "Courier New");

m_rich.SetDefaultCharFormat(cf);
```

Единицей измерения для элемента RichEdit является twips, поэтому *все* величины при форматировании задаются именно в этих единицах.

18.1.4. Форматирование параграфов

Форматирование параграфов текста выполняется с помощью метода **SetParaFormat** класса **CRichEditCtrl** и позволяет изменять ряд характеристик параграфа, например, величину полей, выравнивание текста и проч. Для упрощения работы с отдельными полями структуры форматирования, можно предложить следующий прием:

```
// получаем текущие параметры форматирования
PARAFORMAT pf = {0};
pf.cbSize = sizeof(pf);
m_rich.GetParaFormat(pf);
// изменяем только нужные параметры
pf.dwMask = PFM_TABSTOPS;
pf.cTabCount = 10;

m_rich.SetParaFormat(pf);
```

18.1.5. Сохранение в файле текста

Сохранение в файле текста из элемента RichEdit может базироваться на нескольких подходах.

Вначале воспользуемся простейшим методом:

```
// определяем полную длину текста в RichEdit
int len = m_rich.GetTextLength();
// выделяем буфер требуемой длины
CString tt;
LPTSTR temp = tt.GetBuffer(len+1);
// получаем текст из RichEdit
m_rich.GetWindowText(temp, len+1);
// сохраняем len байт в файле
f.Write(temp, len);
```

Недостаток данного метода заключается в том, что функция **GetWindowText** класса **CWnd** также не может обработать текст, если его длина превышает 64К.

Второй подход лишен указанного недостатка:

```
// выделяем весь текст в RichEdit
m_rich.SetSel(0, -1);
// и получаем его
CString tt = m_rich.GetSelText();
// снимаем выделение
m_rich.SetSel(0, 0);
// находим длину текста
len = tt.GetLength();
// сохраняем len байт в файле
f.Write(tt, len);
```

Хотя предложенный способ неограничен по длине текста в элементе RichEdit, однако пользователь будет видеть как текст в поле ввода дважды инвертирует свой цвет.

Если вас, как разработчика, это не устраивает, следует воспользоваться методом **StreamOut** класса **CRichEditCtrl**, который осуществляет потоковую выгрузку текста произвольной длины из данного элемента управления.

18.2. Элемент управления CListCtrl

По всей видимости, данный элемент управления является самым распространенным для Windows платформ. Вы имеете дело с ListCtrl, когда открываете “Панель управления” или “Мой компьютер”, “Windows проводник” или любимый вами ReGet и так далее, и так далее… Смело можно утверждать, что ни одно серьезное приложение не обходится без этого элемента управления. К недостаткам следует отнести то обстоятельство, что заполнение элемента большими объемами информации, например, 1000 и более строк, может выполняться весьма долго, даже на мощных компьютерах. В данном случае сортировка и поиск работают также медленно. Видимо это подтолкнуло Windows разработчиков на создание так называемого виртуального списка, который появился в последних версиях Windows (точнее – после установки IExplorer версии 5 и выше). Идея состоит в том, что вы только указываете количество строк в списке, например, 100000, а ListCtrl уведомляет родительское окно, например диалог, о том, какая порция строк требует перерисовки в каждый момент времени. Конечно, все это требует от вас дополнительных затрат времени на разработку, однако с использованием виртуального списка ваше приложение выглядит более профессионально, когда конечный пользователь работает с большими объемами информации.

18.2.1. Предварительные замечания

Элемент управления ListCtrl может находиться в одном из четырех режимов отображения:

1. Многоколоночный список. В этом режиме список имеет заголовок. Его первая колонка индексируется нулем и именуется **item**. Все остальные

колонки именуются **subitem** и индексируются 1,2,…. Каждая строка списка однозначно определяется уникальным значением **long** типа, которое обозначается **param**.

2. Набор крупных изображений размером 32×32 пикселя. Каждый элемент списка имеет подпись, а его расположение на поверхности родительского окна произвольно. В этом и последующих режимах вы можете отобразить только текст, содержащийся в **item**, но не в **subitem**.
3. Набор изображений размером 16×16 пикселей. Каждый элемент списка имеет подпись. Обычно пункты выровнены по верхней границе окна, но их расположение на поверхности родительского окна также может быть произвольно.
4. Набор изображений размером 16×16 пикселей. Каждый элемент списка имеет подпись, но, в отличие от предыдущего случая, все пункты располагаются строго сверху вниз и выровнены по левой границе родительского окна.

18.2.2. Создание элемента ListCtrl

Создание элемента управления ListCtrl осуществляется вызовом метода **Create** класса **CListCtrl**. Однако перед этим вы должны выбрать желаемый режим отображения и указать его в стиле окна. Также в стиле создаваемого окна задаются дополнительные параметры, которые зависят от выбранного режима:

Стиль	Описание	Режим
LVS_REPORT	Многоколоночный список	1
LVS_ICON	Набор крупных icon с подписью	2
LVS_SMALLICON	Набор мелких icon с подписью	3
LVS_LIST	Набор мелких icon с подписью	4
LVS_EDITLABELS	Позволяет редактировать **item** или подпись	1,2,3,4
LVS_SINGLESEL	Позволяет выбирать только один пункт списка	1,2,3,4
LVS_SHOWSELALWAYS	Выбранный пункт показан всегда	1,2,3,4
LVS_NOLABELWRAP	Блокирует перенос слов в подписи	2
LVS_ALIGNLEFT	Устанавливает выравнивание	2,3
LVS_ALIGNTOP	Устанавливает выравнивание	2,3
LVS_AUTOARRANGE	Устанавливает автовыравнивание	2,3
LVS_NOSORTHEADER	Заголовок не имеет кнопочного стиля	1
LVS_NOCOLUMNHEADER	Список не имеет заголовка	1

LVS_SORTASCENDING	Устанавливает сортировку по возрастанию	1,2,3,4
LVS_SORTDESCENDING	Устанавливает сортировку по убыванию	1,2,3,4

18.2.3. Инициализация списка

После создания элемента **m_List**, он должен быть заполнен информацией. Проведем такую инициализацию на примере многоколоночного списка в режиме **LVS_REPORT**.

В общем случае вам потребуется элемент данных **m_Images** класса **CImageList**, который является контейнером для изображений размером 16×16 пикселей. Эти изображения размещаются в поле **item**, в левой его части. Заметим, что этот шаг инициализации не является обязательным, если вы не планируете использовать графические данные в списке.

1. В графическом редакторе нужно создать **BMP** файл с изображениями размером 16×16 пикселей. Для определенности будем считать, что этот ресурс имеет идентификатор **IDB_IMAGELIST** и содержит 8 изображений. После этого необходимо создать вспомогательный элемент-массив изображений **m_Images** класса **CImageList** и прикрепить его к списку:

```
// создаем массив изображений
COLORREF clr = ::GetSysColor(COLOR_WINDOW);
m_Images.Create(IDB_IMAGELIST, 16, 8, clr);
m_Images.SetBkColor(clr);
// прикрепляем массив к списку
m_List.SetImageList(&m_Images, LVSIL_SMALL);
```

2. Затем нужно создать заголовок списка, указав требуемое количество колонок. Хотя впоследствии вы можете изменить это количество, добавляя новые колонки или удаляя существующие, но лучше это сделать на этапе инициализации. В нашем случае список будет содержать две колонки:

```
// добавляем первую колонку
CString str = _T("Первая колонка");
int col  = 0;
// рассчитываем ширину колонки в пикселях
int pixW = m_List.GetStringWidth(str);
// указываем выравнивание для всей колонки
int alig = LVCFMT_RIGHT;
// добавляем колонку
m_List.InsertColumn(col++, str, alig, pixW);

// задаем параметры второй колонки
str = _T("Вторая колонка");
// и добавляем ее в список
m_List.InsertColumn(col++, str, alig, pixW);
```

3. Остается оформить строки списка. Однако первым действием на данном шаге нужно указать примерное количество строк, которые будут помещены в список. Задача не совсем тривиальная, так как это количество не всегда заранее известно. В этом случае вы указываете такую порцию строк, на которую будет прирастать список, когда потребуется его расширение. Например, определим, что прирост списка должен выполняться по 64 строки:

```
m_List.SetItemCount(64);
```

Операция эта необязательная, но весьма желательная, поскольку уменьшает фрагментацию памяти. Кроме того, замечено, что это слегка ускоряет работу элемента ListCtrl при большом количестве строк.

Затем к списку добавляется строка по следующей схеме:

```
// индекс колонки
int col = 0;
// индекс изображения в элементе m_Images
int locImage = 1;
// уникальное значение для param
int uniq = 0;
// отображаемый текст
CString str = "в первую колонку";
// добавляем item
int row = m_List.InsertItem(
      LVIF_TEXT | LVIF_IMAGE | LVIF_PARAM,
      col++, str, 0, 0, locImage, uniq++);
str = "во вторую колонку";
// заполняем subitem
m_List.SetItem(row, col++, LVIF_TEXT, str,
                   -1, 0, 0, 0);
```

На представленной схеме обработана одна строка списка, аналогичные действия выполняются и для других строк.

18.2.4. Выборка из списка

Выборка из списка является одной из самых распространенных задач. Ее решение проведем для общего случая – списка, допускающего множественный выбор строк. Для определенности будем считать, что диалоговая панель класса **DlgTest**, производного от класса **CDialog**, содержит элемент ListCtrl с идентификатором **IDC_LIST**. Пользователь отмечает несколько строк в списке и нажимает клавишу **Delete**, желая убрать их из списка.

В таблицу сообщений класса **DlgTest** должна быть помещена макрокоманда, определяющая обработчик нажатий клавиш в поле списка:

```
ON_NOTIFY(LVN_KEYDOWN, IDC_LIST, OnListKeydown)
```

Шаблон обработчика **OnListKeydown**, созданного ClassWizard, следует дополнить несложным кодом, который просто вызывает внутренний метод класса **DlgTest**, в ответ на нажатие пользователем клавиши **Delete**:

```
void DlgTest::OnListKeydown(
             NMHDR* pNMHDR, LRESULT* pResult)
{
    // выполняем преобразование указателя
    LV_KEYDOWN* pLVKeyDow = (LV_KEYDOWN*)pNMHDR;
    // нажата ли нужная клавиша?
    if (pLVKeyDow->wVKey == VK_DELETE)
       DeleteFromList();

    // стандартное действие для WM_NOTIFY обработчика
    *pResult = 0;
}
```

Внутренний метод **DeleteFromList** класса **DlgTest**, выполняет все действия, необходимые для уничтожения отобранных элементов списка:

```
void DlgTest::DeleteFromList()
{
    // определяем, есть ли отмеченные строки
    int k = m_List.GetSelectedCount();
    if (k == 0) {
       MessageBox("Ничего не выбрано", "", MB_OK);
       return;
    }

    // переменная nLast будет содержать номер
    // строки, которая уничтожена последней
    int nLast = 0;
    // отключаем перерисовку элемента управления
    m_List.SetRedraw(FALSE);
    // начиная с конца списка, ищем строки,
    // которые имеют состояние «выбрано»
    for (k = m_List.GetItemCount()-1; 0 <= k; k--) {
       if (m_List.GetItemState(k, LVIS_SELECTED) ==
                                  LVIS_SELECTED) {
          // уничтожаем строку и
          m_List.DeleteItem(k);
          // запоминаем ее номер
          nLast = k;
       }
    }

    // устанавливаем фокус на строку, которая
    // расположена рядом с последней уничтоженной
    UINT mask = LVIS_FOCUSED | LVIS_SELECTED;
    m_List.SetItemState(nLast, mask, mask);
```

```
        // включаем перерисовку и
        // обновляем содержимое списка
        m_List.SetRedraw(TRUE);
        m_List.UpdateWindow();
    }
```

Представленный способ является достаточно универсальным и работает для всех режимов отображения элемента ListCtrl, а не только для режима **LVS_REPORT**.

18.2.5. Пересортировка списка

Пересортировка списка относится к классу более сложных задач. Схема ее решения базируется на том обстоятельстве, что пользователь может щелкнуть манипулятором мышь по любому заголовку, в то время как элемент ListCtrl находится в режиме многоколоночного отображения информации.

> В этом случае объект класса **CHeaderCtrl** автоматически создается и прикрепляется к верхней части списка как дочернее окно.

После щелчка, элемент управления ListCtrl, а не HeaderCtrl, только уведомляет родительское окно об этом событии; сам же процесс пересортировки должен организовать разработчик. Существенным подспорьем является функция **SortItems** класса **CListCtrl**, которая реализует один из методов быстрой сортировки. Однако данная функция требует задания **CALLBACK** функции сравнения элементов списка, что входит уже в вашу зону ответственности.

Итак, согласно схеме, таблица сообщений класса **DlgTest** должна содержать макрокоманду, которая определяет обработчик нажатий левой клавиши мыши на заголовке списка:

```
ON_NOTIFY(LVN_COLUMNCLICK, IDC_LIST, OnColumnClick)
```

Основная задача этого обработчика заключается в вызове метода **SortItems**:

```
void DlgTest::OnColumnClick(
                    NMHDR* pNMHDR, LRESULT* pResult)

{
    // преобразуем указатель
    NM_LISTVIEW* pClick = (NM_LISTVIEW*)pNMHDR;

    // в элементе-данных класса DlgTest
    // сохраняем номер колонки, по которой
    // будет проводиться сортировка
    m_nSort = pClick->iSubItem;

    // вызываем метод сортировки списка и передаем:
    // 1) указатель на функцию сравнения
```

```
        // 2) указатель на объект текущего класса
        m_List.SortItems(CompareFunc, (LPARAM)this);
        *pResult = 0;
    }
```

Ядро системы Windows вызывает функцию **CompareFunc** всякий раз, когда алгоритму сортировки требуется сравнить два элемента ListCtrl. Список формальных параметров функции сравнения имеет предопределенную структуру.

Так, два первых параметра соответствуют уникальным номерам **param** сравниваемых строк.

> Вот почему нужно было побеспокоиться об однозначной идентификации строк еще на этапе заполнения списка. Вернитесь к пункту «Инициализация списка» данного параграфа и еще раз проанализируйте использование идентификатора **uniq**.

Третьим параметром ядро передает то значение, которое мы определили во втором параметре метода **SortItems**. Поскольку мы используем адрес объекта **DlgTest**, то в теле функции **CompareFunc** будем иметь все необходимые данные. Следовательно, использовать какие-либо дополнительные глобальные параметры для связи обработчика **OnColumnClick** и функции сравнения **CompareFunc** не требуется.

Возвращаемое значение функции сравнения также предопределено. Если функция **CompareFunc** возвращает 0, то сравниваемые строки равны между собой; если 1, то первая строка будет размещена за второй, а если –1, то наоборот, первая строка будет размещена перед второй.

Итак, функция сравнения **CompareFunc** выглядит следующим образом:

```
    int CALLBACK CompareFunc (LPARAM lParam1,
                              LPARAM lParam2,
                              LPARAM pObject)
    {
        // преобразуем указатель
        DlgTest* dlg = (DlgTest*)pObject;
        CString str1, str2;
        int     itm1, itm2;
        LV_FINDINFO find;
        // заполняем данные для поиска строки в списке
        // по значению lParam1
        find.flags  = LVFI_PARAM;
        find.lParam = lParam1;
        // получаем позицию строки в списке и
        itm1 = dlg->m_List.FindItem(&find);
        // текст из колонки сортировки
        str1 = dlg->m_List.GetItemText(itm1,dlg-
>m_nSort);
```

```
            // заполняем данные для второй строки и
            find.lParam = lParam2;
            // повторяем операцию
            itm2 = dlg->m_List.FindItem(&find);
            str2 = dlg->m_List.GetItemText(itm2,dlg-
>m_nSort);

            // возвращаем результат сравнения строк
            return lstrcmp(str1, str2);
      }
```

Данный вариант функции **CompareFunc** использует только лексикографическое сравнение для строк всех колонок посредством вызова API функции **lstrcmp**. Если же колонка, по которой проводится сортировка, содержит, например, целочисленные данные или какие-либо даты, то, конечно же, алгоритм сравнения должен быть изменен соответствующим образом.

18.3. Элемент CTreeCtrl

Элемент управления TreeCtrl обычно используется для отображения иерархических структур информации, например, дерева каталогов файловой системы, поэтому в дальнейшей так и будем его именовать – элемент дерево. Этот элемент можно отнести к категории наиболее продуманных Windows объектов, поскольку он практически не имеет недочетов.

18.3.1. Предварительные замечания

Каждый элемент окна дерева имеет уникальный дескриптор типа **HTREEITEM**. Именно через эти дескрипторы выполняются все манипуляции в элементе TreeCtrl.

В основе каждого дерева лежит корневой объект, от которого производится навигация.

Каждый элемент дерева имеет текстовую строку, при этом TreeCtrl предоставляет возможность редактирования данной строки по месту.

Элемент дерева может иметь подчиненные, дочерние элементы. В этом случае слева появляется небольшая кнопка, нажатие на которую распахивает дочерние элементы.

Для описания элемента дерева используется структура **TV_ITEM** со следующими полями:

HTREEITEM hItem дескриптор элемента;

UINT mask набор битовых значений, которые указывают, какие поля структуры задействованы в данный момент: TVIF_HANDLE TVIF_IMAGE
TVIF_PARAM TVIF_SELECTEDIMAGE
TVIF_STATE TVIF_TEXT

UINT state	указывает текущее состояние элемента через набор битовых значений: TVIS_BOLD TVIS_DROPHILITED TVIS_EXPANDED TVIS_SELECTED
LPSTR pszText	текст, связанный с элементом;
int cchTextMax	длина текста;
int iImage	если для TreeCtrl используются графические изображения, то данное поле содержит номер иконки в списке ImageList для обычного, невыбранного состояния;
int iSelectedImage	позиция в списке изображений для состояния “выделено”;
LPARAM lParam	32-х битовое значение, связанное с текущим элементом;
UINT stateMask	практически не используется;
int cChildren	практически не используется.

Как вы заметили, для каждого пункта элемента TreeCtrl вводится в рассмотрение 32-битовое значение **lParam**, которое может быть использовано для хранения дополнительной информации об элементе. Так, например, здесь вы можете запомнить указатель на экземпляр некоторой структуры данных, сопровождающих каждый элемент дерева. Конечно же, в этом случае время жизни таких структур должно быть не меньше, чем время жизни самого элемента управления TreeCtrl.

18.3.2. Создание элемента

Создание элемента дерево осуществляется вызовом метода **Create** класса **CTreeCtrl**. Первым параметром следует указать набор стилей, которые определяют внешний вид элемента TreeCtrl:

TVS_HASLINES	присутствуют линии, связывающие дочерние и родительские объекты;
TVS_LINESATROOT	присутствуют линии, связывающие дочерние и корневые объекты;
TVS_HASBUTTONS	присутствуют кнопки распахивания;
TVS_EDITLABELS	включает возможность редактирования наименования пунктов;
TVS_SHOWSELALWAYS	выделенный объект показан всегда;
TVS_DISABLEDRAGDROP	запрещает операцию drag and drop.

18.3.3. Инициализация элемента

Инициализацию элемента управления дерево **m_tree** начнем с создания списка изображений **m_Images** класса **CImageList**, который является контейнером для изображений размером 16×16 пикселей. По-прежнему будем считать, что **BMP** файл имеет идентификатор **IDB_IMAGELIST** и содержит 8 изображений. Тогда:

```
COLORREF clr = ::GetSysColor(COLOR_WINDOW);
m_Images.Create(IDB_IMAGELIST, 16, 8, clr);
m_Images.SetBkColor(clr);
// прикрепляем список изображений к дереву
m_tree.SetImageList(&m_Images, LVSIL_NORMAL);
```

Следующим шагом создается корневой объект дерева, который является основой для всех остальных элементов. Операция вставки выполняется посредством метода **InsertItem** класса **CTreeCtrl**, однако перед этим необходимо корректно заполнить поля структуры **TV_INSERTSTRUCT**, при этом поле **lParam** в нашем примере будет соответствовать уровню заполнения: 0 помечает корневой элемент, 1 – элементы первого слоя и так далее.

```
TV_INSERTSTRUCT tvIns;
// заполняем структуру для корневого элемента,
tvIns.hParent      = TVI_ROOT;
// который располагается первым
tvIns.hInsertAfter = TVI_FIRST;
tvIns.item.mask    = TVIF_TEXT | TVIF_IMAGE |
                     TVIF_SELECTEDIMAGE | TVIF_PARAM;
long level = 0;
tvIns.item.lParam   = level++;
tvIns.item.pszText  = _T("Мой компьютер");
tvIns.item.cchTextMax = lstrlen(tvIns.item.pszText);
tvIns.item.iImage         = I_IMAGECALLBACK;
tvIns.item.iSelectedImage = I_IMAGECALLBACK;

// заносим корневой элемент в дерево,
// запоминая его дескриптор:
HTREEITEM root = m_tree.InsertItem(&tvIns);
```

В нашем примере поле **hInsertAfter** принимает значение **TVI_FIRST**, поскольку мы заносим самый первый элемент в список. Кроме этого, можно использовать и другие мнемонические константы: **TVI_LAST** – расположить вставляемый объект последним в текущей ветке; **TVI_SORT** – вставка с сортировкой в пределах данной ветки. Или, используя дескриптор существующего в TreeCtrl элемента, можно точно указать местоположение для нового объекта.

Обратите внимание на константу **I_IMAGECALLBACK**, которая использована вместо индекса графического списка **m_Images** для обоих со-

стояний корневого элемента. Эта константа сообщает элементу дерево о том, что родительский объект (а не TreeCtrl) будет самостоятельно изменять графический образ корневого элемента, когда пользователь попытается его распахнуть.

Основная причина применения такой технологии – эффективность работы с элементом дерево. Конечно же, TreeCtrl может сам изменить графическое изображение для любого элемента, но если эту операцию проделывает родительский объект, то само изменение проходит более плавно, практически незаметно для человеческого глаза.

18.3.4. Вставка объектов

Вставка объектов *n*-того уровня выполняется аналогично, с той лишь разницей, что в качестве родительского дескриптора следует указать объект предыдущего (n-1)-го уровня

```
// заполняем структуру для n-того уровня
tvIns.hParent = hPrev;
// используем вставку с сортировкой
tvIns.hInsertAfter = TVI_SORT;
tvIns.item.mask    = TVIF_TEXT | TVIF_IMAGE |
                     TVIF_SELECTEDIMAGE | TVIF_PARAM;
tvIns.item.lParam   = level;
tvIns.item.pszText  = _T("n-й уровень");
tvIns.item.cchTextMax = lstrlen(tvIns.item.pszText);
tvIns.item.iImage         = I_IMAGECALLBACK;
tvIns.item.iSelectedImage = I_IMAGECALLBACK;

// заносим элемент в дерево
m_tree.InsertItem(&tvIns);
```

Нередко возникает необходимость динамической вставки элементов в дерево. Класс **`CTreeCtrl`** включает ряд методов, которые существенно облегчают решение этой задачи. Так, метод **`GetRootItem()`** возвращает дескриптор корневого элемента (в нашем примере – нулевого уровня), что избавляет вас от необходимости запоминания его в статической переменной. Метод **`GetChildItem(hItem)`** возвращает дескриптор первого дочернего объекта для **`hItem`**, а метод **`GetNextSiblingItem(hItem)`** позволяет проитерировать все элементы одного уровня. Наконец, метод **`GetParentItem(hItem)`** вернет дескриптор родительского объекта [в нашем примере – (*n*-1)-го уровня] для текущего элемента.

18.3.5. Позиционирование

Установка фокуса на заданный пункт и перерисовка объекта в элементе управления TreeCtrl не представляют труда. Так, строка программного кода

```
m_tree.EnsureVisible(hItem);
```

не только выполнит обновление элемента, заданного дескриптором **hItem**, но и осуществит необходимый скроллинг изображения в клиентской части TreeCtrl, если в результате навигации этот объект невидим в данный момент.

Напомним, что в элементе управления TreeCtrl в каждый момент времени может быть отселектирован только один пункт, дескриптор которого можно получить с помощью метода **GetSelectedItem()**

```
hItem = m_tree.GetSelectedItem();
```

По известному дескриптору, установить фокус ввода на произвольный пункт дерева можно, используя метод **Select()**

```
m_tree.Select(hItem, TVGN_CARET);
```

18.3.6. Динамическое изменение образа объекта

Для решения данной задачи вы должны указать мнемоническую константу **I_IMAGECALLBACK** для позиции графического образа некоторого пункта дерева (см. п.5.3.3). Кроме того, в карте сообщений родительского объекта – в нашем случае диалога **Dlg** – должен присутствовать макрос следующего вида:

```
ON_NOTIFY_REFLECT(TVN_GETDISPINFO, OnGetdispinfo)
```

тогда ядро операционной системы будет посылать соответствующее сообщение диалогу **Dlg** всякий раз, когда образ данного пункта должен быть перерисован. Учитывая это, тело обработчика **OnGetdispinfo()** может выглядеть следующим образом:

```
void Dlg::OnGetdispinfo(NMHDR* pNMHDR, LRESULT* pRes)
{
   // преобразовываем указатель
   TV_DISPINFO* pdi = (TV_DISPINFO*)pNMHDR;
   // указатель pdi включает структуру TV_ITEM,
   // поля которой мы должны изменить для адекватного
   // отображения текущего элемента дерева.
   // Кроме того, поле state укажет нам текущее
   // состояние элемента, а номер графического
   // образа будет зависеть от этого состояния.
   int k = (pdi->item.state & TVIS_EXPANDED) ? 5 : 4;

   // изменяем номер графического образа
```

```
    pdi->item.iImage =
    pdi->item.iSelectedImage = k+pdi->item.lParam;
}
```

18.3.7. Динамический отклик на пользовательский выбор

Данная задача возникает тогда, когда необходимо отслеживать изменение пользователем выбора в элементе управления. В этом случае в карте сообщений диалога **Dlg** должен присутствовать макрос вида:

```
ON_NOTIFY_REFLECT(TVN_SELCHANGED, OnSelchanged)
```

Обработчик данного сообщения, конечно же, зависит от характера решаемой задачи, стоящей перед вами, в нашем случае мы просто перешлем сообщение **um_TREE_CHANGE** главному окну приложения о том, что пользователь изменил выбор в элементе TreeCtrl.

```
void Dlg::OnSelchanged(NMHDR* pNMHDR, LRESULT* pRes)
{
   // преобразовываем указатель
   NM_TREEVIEW* pt = (NM_TREEVIEW*)pNMHDR;
   // указатель pt включает поля itemOld и
   // itemNew, соответствующие старому и новому
   // выбору пользователя, причем в каждой из этих
   // структур определены только поля
   // mask, hItem, state, lParam.

   // посылаем сообщение главному окну
   AfxGetMainWnd()->SendMessage(um_TREE_CHANGE);
}
```

18.3.8. Отслеживание нажатий кнопок мыши

Данное событие мы проиллюстрируем отображением контекстного меню, если пользователь нажал левую клавишу манипулятора мышь, но только тогда, когда щелчок пришелся точно на какой-либо элемент дерева (а не рядом с ним!), причем пункт меню “Уничтожить” следует заблокировать, если тестируемый объект имеет дочерние элементы.

С учетом сказанного, обработчик сообщения нажатия правой кнопки мыши имеет вид

```
void OnRButtonDown(UINT nFlags, CPoint point)
{
   // по координатам точки point определяем
   // местоположение нажатия через значение flag
   UINT flag;
   HTREEITEM hItem = tree.HitTest(point, &flag);

   // если нажатие произведено на элементе дерева,
   // то определяем имеет ли он дочерние объекты,
   // одновременно устанавливаем фокус на
```

```
        // этот элемент. В противном случае,
        // сбрасываем флаг в нулевое значение.
        if (flag & TVHT_ONITEM) {
            flag = !m_tree.ItemHasChildren(hItem);
            m_tree.Select(hItem, TVGN_CARET);
        }
        else
            flag = 0;

        // загружаем меню из ресурсов приложения
        CMenu popm, *ptm;
        popm.LoadMenu(IDR_TEMP_MENU);

        // получаем указатель на первое popup меню
        ptm = popm.GetSubMenu(0);

        // блокируем пункт "уничтожить", если требуется
        if (!flag)
            ptm->EnableMenuItem(ID_TREE_DELETE,
                        MF_GRAYED | MF_BYCOMMAND);

        // отображаем временное меню
        CRect rc;
        GetWindowRect(rc);
        ptm->TrackPopupMenu(TPM_CENTERALIGN |
TPM_RIGHTBUTTON,
                    rc.left+point.x, rc.top+point.y, this);
    }
```

18.3.9. Редактирование меток объектов

Последней в данном разделе рассмотрим задачу об изменении текста, сопровождающего каждый элемент дерева TreeCtrl. Допустим, пользователь нажимает клавишу **Insert**, после чего включается возможность редактирования "по месту" текста выделенного пункта дерева. Нажатие клавиши **Enter** или щелчок по другому пункту дерева являются сигналом для окончания редактирования. Ваша реакция на это событие заключается в проверке правильности ввода и прием пользовательских изменений посредством пересортировки текущей ветки дерева.

Согласно изложенному, карта сообщений диалога **Dlg** должна содержать макросы вида:

```
    ON_NOTIFY_REFLECT(TVN_KEYDOWN, OnKeyDown)
    ON_NOTIFY_REFLECT(TVN_ENDLABELEDIT, OnEndEdit)
```

Обработчик нажатия клавиш отслеживает нажатие **Insert** и включает редактирование выделенного пункта дерева:

```
void Dlg::OnKeyDown(NMHDR* pNMHDR, LRESULT* pRes)
{
    // преобразовываем указатель
```

```
    TV_KEYDOWN* pkd = (TV_KEYDOWN*)lParam;

    // проверяем нажатие нужной клавиши
    if (pkd->wVKey == VK_INSERT) {
        // определяем выделенный пункт
        HTREEITEM hItem = m_tree.GetSelectedItem();

        // включаем редактирование метки
        CEdit* ed = m_tree.EditLabel(hItem);
        // если требуется, здесь можно выполнить
        // дополнительные действия с EditBox
    }
    *pRes = 0;
}
```

После того, как пользователь закончит редактирование текста метки элемент управления TreeCtrl посылает родительскому объекту сообщение **WM_NOTIFY** с кодом извещения **TVN_ENDLABELEDIT**, обработчик которого обычно имеет вид:

```
void Dlg::OnEndEdit(NMHDR* pNMHDR, LRESULT* pRes)
{
    // преобразовываем указатель
    TV_DISPINFO* pdi = (TV_DISPINFO*)lParam;

    // проверяем действительность указателя
    if (pdi->item.pszText) {
        // контроль введенного текста

        // подготавливаем поля структуры
        // для обновления информации
        TV_ITEM ns;
        ns.mask   = TVIF_HANDLE | TVIF_TEXT;
        ns.hItem   = pdi->item.hItem;
        ns.pszText = pdi->item.pszText;
        ns.cchTextMax = lstrlen(ns.pszText);

        // обновляем информацию
        m_tree.SetItem(&ns);

        // пересортировываем текущую ветку дерева
        HTREEITEM hParen =
m_tree.GetParentItem(ns.hItem);
        m_tree.SortChildren(hParen);
    }
    *pRes = 0;
}
```

18.4. Элемент CSpinButtonCtrl

Наборный счетчик обычно используется совместно с полем ввода. В этом случае поле ввода должно быть расположено слева от наборного счетчика как визуально, так и в порядке навигации с помощью клавиши **Tab** между элементами управления. Поле ввода является как бы партнером – "buddy" – наборного счетчика для этого случая. При использовании графического редактора необходимо установить следующие атрибуты на закладках свойств наборного счетчика:

Auto buddy; **Set buddy integer** и **Arrow key**

В качестве примера управления элементом CSpinButtonCtrl рассмотрим задачу изменения шага наборного счетчика до величины 0.5. Дело в том, что по умолчанию счетчик может работать только с целыми значениями с шагом 1. Это объясняется тем, что разработчики Windows не предусмотрели иного поведения для данного элемента управления. Поэтому в обработчике **WM_INITDIALOG** применим несложный трюк: установим верхнюю и нижнюю границы изменения наборного счетчика в два раза большими, чем требуется. Например, если мы хотим, чтобы размер шрифта изменялся от 2 до 20 пунктов с шагом 1/2, а стартовое значение было бы 4 пункта, то метод **OnInitDialog()** класса Dlg, производного от CDialog, может иметь вид:

```
BOOL Dlg::OnInitDialog()
{
       // получаем указатель на наборнй счетчик
       CSpinButtonCtrl* pSpin =
      (CSpinButtonCtrl*)GetDlgItem(IDC_SPIN);

       // устанавливаем требуемые значения
       pSpin->SetRange(4, 40);
       pSpin->SetPos(8);

       // инициализация других элементов
}
```

Теперь все изменения в наборном счетчике будут поступать диалогу через сообщение **WM_VSCROLL**, обработчик которого может иметь вид:

```
void Dlg::OnVScroll(UINT nSBCode, UINT nPos,
                    CScrollBar* pScrBar)
{
   // проверяем от наборного ли счетчика
   // поступило сообщение
   if (GetDlgItem(IDC_SPIN) == pScrBar) {
       CString tt;

       // уменьшаем текущее значение счетчика
```

```
        // в два раза
        tt.Format("%.1f", 0.5*double(nPos));
        // и отображаем в EditBox
        CSpinButtonCtrl* ps = (CSpinButtonCtrl*) pScrBar;
        ps->GetBuddy()->SetWindowText(tt);
    }
    else
        CDialog::OnVScroll(nSBCode, nPos, pScrBar);
}
```

Итак, мы рассмотрели наиболее интересные моменты при работе с Win32 элементами управления, а также особенностями MFC реализации для поддержки данных элементов. В завершении этой темы заметим, что MFC инкапсулирует многие, но не все возможности новых элементов управления. Однако не забывайте, что, получив дескриптор окна элемента управления, вы, по-прежнему, можете послать окну любое допустимое сообщение, так же как это вы делали средствами API.

19. Архитектура Document-View

В предыдущих лекциях рассматривались основные положения, касающиеся создания MFC приложений, базирующихся на диалогах. Это типичный подход для Windows программирования. Однако для Visual C++ такие приложения не являются основными. В Visual C++ модель визуального программирования получила дальнейшее развитие в виде новой архитектуры приложения, называемой **`Document-View`**. Средство AppWizard позволяет создавать приложения, основанные на документах: приложения с однодокументным интерфейсом (Single Document Interface - SDI) и приложения с многодокументным интерфейсом (Multiple Document Interface - MDI). Они то и являются основными для Visual C++. Нужно напомнить, что любое MFC приложение функционирует как набор взаимодействующих объектов. В этом его основная суть. Так вот, в Visual C++ эти объекты организованы в систему со вполне четкой архитектурой **`Document-View`**.

19.1. Пользовательский интерфейс

Правильно организованный пользовательский интерфейс должен обеспечивать первостепенную роль пользователя, позволяя ему управлять действиями приложения, а не наоборот. Самым подходящим для реализации такого требования является интерфейс, сфокусированный на данных. Суть его в том, что пользователь, работая с приложением, не начинает очередное действие с поиска команд, программ и проч. Прежде всего, он должен видеть перед собой нужные данные, которые в свою очередь автоматически сопровождаются необходимыми для работы с ними командами и инструментарием.

Особая роль здесь принадлежит неким упорядоченным, логически увязанным данных, которые мы и будем именовать **документом**. Простейший пример документа – это любой текст или таблица. Однако документ может быть и каким-либо изображением, звуковым или видео файлом, в конце концов, просто числом, другими словами – любым объектом.

Приложение должно строиться так, чтобы, видя один или несколько документов, пользователь сосредоточил внимание именно на них самих, а не на средствах работы с документами. Архитектура **`Document-View`** как раз предоставляет, и в некотором смысле *навязывает*, разработчику систему объектов, позволяющих строить приложения, сфокусированные на данных, а точнее – на документах.

19.2. Документ и его представления

Центральными объектами в архитектуре приложения являются один или несколько объектов – документов. Они ориентированы на хранение

информации и имеют хорошо развитые методы загрузки, сохранения и управления данными. Документы создаются как объекты классов, производных от класса **CDocument** библиотеки MFC.

Каждый документ сопровождается одним или несколькими объектами, которые называются вид, т.е. внешнее представление документа, его облик. Через эти представления происходит взаимодействие конечного пользователя с документами. Вид является объектом, предназначенным для выполнения двух функций: 1) отображения документа на экране; 2) распознавания команд пользователя по управлению документом.

При этом виды используют методы документа для его изменения.

Виды создаются как объекты классов, производных от класса **CView** библиотеки классов MFC. Кроме класса **CView**, вы можете использовать целый ряд классов, производных от **CView**, которые также можно применять для создания собственных классов отображения. Так, например, если документ большой, то, вид будет отображать на экране только часть документа и, конечно же, должен предоставлять пользователю возможность перемещаться по документу. В этом случае вы можете использовать класс **CScrollView**.

Запомните два правила:

1. Каждый вид должен быть сопоставлен некоторому документу.

2. Один документ может иметь несколько видов, и в таком случае они обеспечивают различное представление на экране одного документа. Например, некие статистические данные вы одновременно можете показывать пользователю как текстовую таблицу, как график и как столбчатую диаграмму.

19.3. Создание документов и его отображений

Документы и их представления являются одними из наиболее важных составляющих архитектуры **Document-View**. Их создание выполняется каркасом приложения. Процесс создания довольно сложен, здесь же остановимся лишь на наиболее существенных моментах, важных для понимания этого процесса.

1. Помимо уже указанных объектов **CDocument** и **CView** рассматриваемая архитектура включает еще один специальный объект – шаблон документа (document template). Понятие шаблона документа инкапсулирует класс **CDocTemplate** и производные от него. Шаблоны задают описание наиболее существенной части архитектуры **Document-View**, включая взаимосвязи класса документа, класса представления документа и класса окна-рамки документа. В MFC существуют два документных шаблона – для

однодокументного интерфейса (SDI) и для многодокументного интерфейса (MDI).

2. Создание документа и связанных с ним объектов выполняется при открытии пользователем документа – существующего или нового. Это происходит в начале работы приложения, а также когда пользователь выбирает команды Open и New в меню File.

3. Создание начинается с выбора шаблона документа. В соответствии с описанным в шаблоне классом документа создается сам документ. Если шаблон задает MDI-интерфейс, то каждый раз создается новый объект-документ. Если же шаблон задает SDI-интерфейс, то новый объект создается только один раз, а при выборе пользователем команд Open и New выполняется "очистка" уже имеющегося объекта документа.

4. Для инициализации созданного документа каркас приложения вызывает метод **`OnNewDocument()`** при открытии нового документа, а при открытии уже имеющегося – **`OnOpenDocument()`**. Оба метода принадлежат базовому классу **`CDocument`**.

Заметим, что указанные методы являются виртуальными, и конечно же, вы, как разработчик, можете их переопределить, например, задавая какие-то необходимые действия по инициализации документа. Однако эти методы и сами по себе выполняют ряд действий, связанных с инициализацией, в частности, проверяют корректность создания объекта документа. Поэтому ваши переопределенные методы **должны** содержать вызовы методов базового класса. Кстати, заготовки методов **`OnNewDocument()`** и **`OnOpenDocument()`**, созданные AppWizard, уже содержат такие вызовы, так что не удаляйте их.

Заметим также, что знатоки C++ могли бы использовать конструктор документа для его инициализации. Однако если используется шаблон SDI-интерфейса, то при повторном создании или открытии документа **конструктор не вызывается**, а производится очистка документа. Поэтому конструктор документа можно использовать для инициализации лишь в случае приложений с MDI-интерфейсом.

5. Создание окон приложения. После создания документа создается объект главного окна-рамки, который является Windows-окном приложения. Внутри этого окна будут располагаться окна-рамки документов. Объект окно-рамка документа создается главным окном-рамкой. Затем окно-рамка документа создает объект-облик, которого и отображает содержимое документа.

6. Если открывается уже существующий документ, каркас приложения вызывает метод **`Serialize()`**, который производит загрузку документа. Заметим, что этот же метод вызывается при закрытии документа, тогда он выполняет функции сохранения документа.

7. Заключительный этап создания документа и связанных с ним объектов – инициализация вида и отображение содержимого документа.

19.4. Взаимодействие документа и его видов

Как уже отмечалось, виды логически связаны со своим документом. У классов **CDocument** и **CView** конечно же имеются механизмы реализации этой связи. Так, класс **CDocument** инкапсулирует список, хранящий все облики данного документа. Имеются методы, с помощью которых документ добавляет новый вид к списку или удаляет его из списка, вы можете осуществить просмотр списка и получить доступ к любому виду.

Метод **GetDocument()** класса **CView** возвращает указатель на объект документа и позволяет виду получить доступ к открытым полям объекта документа.

Рассмотрим еще одну задачу. Пусть документ имеет несколько видов, один из которых изменяет документ. Тогда остальные виды должны синхронизировать свое отображение с измененным содержанием этого документа. Такую синхронизацию обеспечивает метод **UpdateAllViews()** класса **CDocument**.

Классы объектов-обликов должны быть устроены так, чтобы при получении сообщения **Update()** обрабатывающий их метод производил синхронизацию облика с документом. Обычно это означает синхронизацию данных облика и синхронизацию изображения в окне облика. Простейший вариант такой синхронизации дает метод **OnUpdate()** класса **CView**. Он состоит в отправке объектом сообщения **OnDraw()** самому себе. Если разработчик не предусмотрел своего метода обработки сообщения **Update()**, то при получении этого сообщения облик выполняет метод **OnUpdate()** базового класса **Cview**.

19.5. SDI-приложение

В первую очередь приведем последовательность действий MFC каркаса в архитектуре документ-вид при запуске SDI приложения:

1) создается документный шаблон – объект класса, производного от **CDocTemplate**;
2) динамически создается документ – объект класса, производного от **CDocument**;
3) динамически создается MFC объект главного окна приложения – окна рамки, которое является объектом класса, производного от **CFrameWnd**;

4) динамически создается MFC объект окно-вид класса, производного от **CView**;
5) создаются Windows окна соответствующими вызовами метода **OnCreate()** для **CFrameWnd** и **CView** классов;
6) вызывается метод **CDocument::OnNewDocument()**. Напомним, что здесь вы должны проводить инициализацию полей объекта документа, но *обязательно* после вызова метода базового класса;
7) вызывается метод **CView::OnInitialUpdate()**. Здесь вы можете провести инициализацию полей объекта вид, точно также как это делалось в обработчике **OnInitDialog()** класса **CDialog**;
8) клиентская область объекта вид объявляется недействительной, следовательно, вызывается метод **OnDraw()** класса **CView**.

Теперь, вкратце, рассмотрим проект однооконного приложения, базирующегося на однодокументном интерфейсе, которое создано с использованием средств MFC AppWizard.

19.5.1. Ресурсы приложения

Легко заметить, что приложение с однооконным интерфейсом имеет гораздо больше ресурсов, чем приложение, использующее в качестве главного окна приложения диалоговую панель. В нем определены не только диалоговые панели и таблица текстовых строк, но и пиктограмма, меню, панель управления и таблица акселераторов.

Шаблон меню. В ресурсах приложения определен только один шаблон меню, имеющий идентификатор **IDR_MAINFRAME**. Когда пользователь выбирает пункт меню, операционная система Windows передает командное сообщение главному окну приложения. Большая часть строк меню **IDR_MAINFRAME** имеет стандартные идентификаторы, описанные в библиотеке MFC.

Панель управления. Кнопки панели управления, как правило, имеют одинаковые идентификаторы с пунктами меню, и следовательно, дублируют функции меню.

Пиктограмма. В файле ресурсов приложения определена пиктограмма, которая используется всеми приложениями, построенными на основе MFC.

Таблица текстовых строк. Одним из самых объемных ресурсов приложения является таблица текстовых строк. В ней определены названия главного окна приложения, строки, отображаемые в панели состояния и проч.

Наибольший интерес представляет текстовая строка с идентификатором **IDR_MAINFRAME**. В этой строке собрана различная информация, относящаяся к типу документов приложения. Формирование этой строки выполняется MFC, однако полезно знать структуру этой строки. Итак, строка состоит из 7 частей, разделенных символом ‘\n’.

Класс **CDocTemplate** содержит метод **GetDocString()**, который производит разбор строки **IDR_MAINFRAME** и выдает каждую ее часть.

```
BOOL GetDocString(CString& str, int index);
```

Параметр **index** должен быть одним из следующих значений, входящих в **enum CDocTemplate::DocStringIndex**.

CDocTemplate::windowTitle	Имя приложения, например, “Microsoft Excel”
CDocTemplate::docName	Имя документа по умолчанию, например “Sheet”
CDocTemplate::fileNewName	Имя типа документа, например, “Worksheet”. Если приложение обслуживает более одного типа документа, это имя появляется в диалоге открытия документа
CDocTemplate::filterName	Имя документа, для заполнения ComboBox в диалоге открытия файлов, например, “Worksheets (*.xls)”
CDocTemplate::filterExt	Расширение для файлов документа, например, “.xls”
CDocTemplate::regFileTypeId	Имя, под которым данный документ будет зарегистрирован в базе данных Windows, например, “ExcelWorksheet”. После регистрации, Explorer будет запускать ваше приложение при обработке документов с этим расширением
CDocTemplate::regFileTypeName	Имя типа документа, которое будет занесено в файл реестра, например, “Microsoft Excel Worksheet”

Если вы не желаете регистрировать документ в файле реестра, поскольку это засоряет базу данных Windows лишней информацией, уберите все разделы, кроме первого, у текстовой строки с идентификатором **IDR_MAINFRAME**.

19.5.2. Динамическая замена одного объекта вид другим

Как было указано в п.6.5, последовательность действий MFC каркаса при запуске SDI приложения строго предопределена и, обычно, не требует вмешательства разработчика.

Если же перед вами стоит задача соответствующего изменения объекта вид в зависимости, например, от типа загружаемого файла (текстовый или графический), то стоит рассмотреть последовательность вызовов при открытии нового документа SDI приложении:

1) вызывается метод **CWinApp::OnFileOpen()**. Отображается диалог открытия файлов;

2) после того, как пользователь выберет нужный файл, вызывается метод **OpenDocumentFile()** класса **CWinApp**. Последний, в свою очередь, вызывает метод **OpenDocumentFile()** класса **CDocTemplate**;
3) вызывается метод **CDocument::Serialize()**. Вы переопределяете этот виртуальный метод для выполнения собственной загрузки информации из файла в соответствующие поля документа;
4) вызывается метод **CDocument::OnOpenDocument()**, переопределяя который вы можете обновить представление документа, например, вызовом метода **CDocument::UpdateAllViews()**.

Таким образом, наш способ решения задачи динамической подмены одного представления документа другим в SDI приложении сводится к вмешательству в предопределенный MFC каркасом порядок действий. Сделать это можно только между шагом 1 и 2 при открытии нового документа. Итак, определим в нашем приложении два класса **CTextView** и **CGraphView**, наследованные от **CView**. Объект первого класса потребуется тогда, когда пользователь выберет некий текстовый файл с расширением **blan**, а второй – когда пользовательский файл имеет расширение **grph**.

Полностью заместим метод **OnFileOpen()** класса, производного от **CWinApp**, например:

```
void CTestApp::OnFileOpen()
{
   // задаем фильтр требуемых файлов
   CString szFilter = "Текст (*.blan)|*.blan|"
                      "Графика (*.grph)|*.grph|"
                      "Все файлы (*.*)|*.*||";
   CFileDialog dlgFile(TRUE, 0, 0,
         OFN_HIDEREADONLY |OFN_FILEMUSTEXIST,
         szFilter, 0);

   // отображаем диалог открытия файлов
   if (dlgFile.DoModal() == IDOK) {
       szFilter = dlgFile.GetPathName();
       NewFile(szFilter);
   }
}
```

Внутренний метод NewFile класса CTestApp выполняет требуемую замену объектов-видов, но только если новый файл имеет расширение, отличное от расширения текущего, обрабатываемого файла. Для учета этого, введем поле **m_fType** в классе **CTestApp**, которое будет хранить тип текущего файла. Тогда:

```
void CTestApp::NewFile(LPCTSTR szFullPath)
{
   // выделяем расширение нового файла
   CString szFilter = szFullPath;
   int k = szFilter.Find(_T("."));
   if (k == -1) return;
   szFilter = szFilter.Mid(k+1);
   // и переводим его в верхний регистр
   szFilter.MakeUpper();

   // запоминаем тип нового файла
   int old_flag = m_fType;
   m_fType = (szFilter == _T("BLAN")) ? 0 : 1;

   // делаем замену только при несовпадении типов
   if (old_flag != m_fType) {

        // сложность в том, что мы не можем
        // напрямую создавать объекты CView,
        // т.к. класс имеет закрытый конструктор,
        // поэтому используем обходной путь -
        // через вызов метода CreateView()

        // получаем указатели на объекты окна-рамки
        // и текущего окна-вида
        CFrameWnd* pFrame = (CFrameWnd*)m_pMainWnd;
        CView*   pOldView = pFrame->GetActiveView();

        // выставляем указатель на новый вид
        CRuntimeClass* pNewViewClass;
        if (m_fType)
          pNewViewClass = RUNTIME_CLASS(CTextView);
        else
          pNewViewClass = RUNTIME_CLASS(CGraphView);

        // заполняем поля структуры
        CCreateContext context;
        context.m_pNewViewClass = pNewViewClass;
        context.m_pCurrentDoc =
                       pFrame->GetActiveDocument();
        // и создаем новый объект вид
        CView* pNewView = STATIC_DOWNCAST(CView,
                       pFrame->CreateView(&context));

        if (pNewView) {
          // отображаем новый объект вид
          pNewView->ShowWindow(SW_SHOW);
          pNewView->OnInitialUpdate();
          // прикрепляем объект вид к окну-рамке
          pFrame->SetActiveView(pNewView);
          pFrame->RecalcLayout();
          // только теперь уничтожаем
```

```
            // старый объект вид
            pOldView->DestroyWindow();
        }
    }
    // восстанавливаем последовательность
    // действий, предопределенную каркасом
    AfxGetApp()->OpenDocumentFile(szFullPath);
}
```

19.5.3. Расщепление главного окна приложения

Технология второго способа множественного представления документа в SDI приложениях принципиально иная, она основана на использовании как называемого расщепления главного окна приложения на базе объектов класса **CSplitterWnd**. Главное окно разделяется на несколько частей вертикальными и/или горизонтальными "расщепителями" с последующей загрузкой в каждую область отдельного представления одного и того же документа. Заметим, что такие области расщепления какого-либо окна именуются английским словом "pane".

Допустим, нам требуется представить документ в двух видах. Необходимые действия в этом случае заключаются в следующем:

1. Добавим к полям класса **CTestFrame**, который наследован от класса **CFrameWnd**, объект **m_wndSplitVert** класса **CSplitterWnd**. Именно он будет выполнить вертикальное расщепление главного окна приложения на две независимые области.

2. Создадим два класса **COrderView** и **CGridView**, производные от **CView**. Объекты этих классов будут загружены в области расщепления главного окна.

3. Переопределим метод **CFrameWnd::OnCreateClient()**, который вызывается перед **OnCreate()**.

> Несмотря на то, что в функцию **OnCreateClient()** передаются указатели, *не меняйте их содержимое*. Используйте их только для просмотра значений полей структур.

С учетом сказанного, данный метод может иметь вид:

```
BOOL CTestFrame::OnCreateClient(LPCREATESTRUCT lpcs,
                                CCreateContext* pContext)
{
    // создаем вертикальный splitter с
    // одной строкой и двумя колонками
    if (!m_wndSplitVert.CreateStatic(this, 1, 2,
                WS_CHILD | WS_VISIBLE | WS_BORDER)) {
        return FALSE;
    }
```

```
    // создаем объект класса COrderView и
    // загружаем его в левый "pane"
    if (!m_wndSplitVert.CreateView(0, 0,
                RUNTIME_CLASS(COrderView),
                CSize(0, 0), pContext)) {
        return FALSE;
    }

    // создаем объект класса CGridView и
    // загружаем его в правый "pane"
    if (!m_wndSplitVert.CreateView(0, 1,
                RUNTIME_CLASS(CGridView),
                CSize(0, 0), pContext)) {
        return FALSE;
    }

    // обратите внимание, мы создали два объекта вид
    // нулевого размера. Если так и оставить,
    // то левый "pane" отобразится схлопнутым,
    // а правый - займет всю клиентскую область
    // главного окна приложения, поэтому
    // откорректируем размеры левого "pane"

    // зададим "идеальную" и минимальную
    // ширину в пикселях
    int cxIdeal = 100;
    int cxMin   = 0;

    m_wndSplitVert.SetColumnInfo(0, cxIdeal, cxMin);
    return TRUE;
}
```

4. Необходимо добавить новый объект вид к списку обликов объекта документ. Для этого используется метод **`CDocument::AddView(CView* pView)`**. По умолчанию документ "знает" только о классе **`COrderView`**, который зарегистрирован в шаблоне документа. Указатель на объект класса **`CGridView`** можно получить при помощи вызова метода класса **`CSplitterWnd`**, который возвращает указатель на **`CWnd`** класс:

```
CWnd* pView = m_wndSplitVert.GetPane(0, 1);
```

Не забудьте о преобразовании указателя при вызове **`AddView()`**.

19.6. MDI-приложение

Приложения с многооконным интерфейсом (MDI) создаются для одновременной работы с несколькими документами. В таких приложениях каждому документу будет отведено собственное окно просмотра, но, тем не менее, все окна просмотра документов будут расположены внутри главного

окна приложения, будут иметь общее меню, а также одну панель управления и панель состояния.

Пусть при помощи AppWizard создан проект приложения с многооконным интерфейсом с именем **Multi**. Сразу после запуска открывается дочернее окно, предназначенное для просмотра документа, которое получает название **Multi1**. При помощи пунктов New и Open меню File можно создавать новые дочерние окна. Если одновременно открыто несколько окон, то можно упорядочить расположение этих окон и пиктограмм, представляющих минимизированные окна. Для этого предназначен пункт меню Window.

Рассмотрим характерные отличия в MDI архитектуре в сравнении с однооконным интерфейсом.

19.6.1. Ресурсы приложения

В ресурсах многооконного приложения определены два меню, две пиктограммы, а таблица текстовых строк имеет существенно больший размер.

Так, для многооконного приложения **Multi** в будут определены два меню с идентификаторами **`IDR_MAINFRAME`** и **`IDR_MULTITYPE`**. Меню с идентификатором **`IDR_MAINFRAME`** используется, если в приложении не открыт ни один документ. Причем его пункты отличаются от пунктов меню однооконного приложения. Меню с идентификатором **`IDR_MULTITYPE`** отображается после создания пользователем нового документа или после открытия уже существующего документа. В этом меню определен пункт Window, строки которого служат для управления окнами MDI приложения.

Кстати, несмотря на то, что приложение имеет два меню, для него определена только одна панель управления.

В файле ресурсов приложения **Multi** определены две пиктограммы **`IDR_MAINFRAME`** и **`IDR_MULTITYPE`**. Первая представляет приложение, а вторая используется для представления документа, с которым работает приложение.

19.6.2. Классы приложения

Основное отличие состоит в изменении базового класса для главного окна приложения и появление нового класса для дочерних окон.

Класс приложения	Базовый класс	Описание
`CMultiApp`	**`CWinApp`**	Главный класс приложения
`CMainFrame`	**`CMDIFrameWnd`**	Класс главного окна приложения
`CChildFrame`	**`CMDIChildWnd`**	Класс дочернего окна MDI
`CMultiDoc`	**`CDocument`**	Класс документа приложения
`CMultiVeiw`	**`CView`**	Класс окна просмотра документа

Главный класс приложения **CMultiApp**, также как и в однооконном приложении, управляет работой всего приложения. Методы этого класса выполняют инициализацию приложения, обработку цикла сообщений и вызываются при завершении приложения. По-прежнему, определен только один объект главного класса приложения – **theApp**.

Метод **InitInstance()** несколько отличается от такового для однооконных приложений. Так, при создании шаблона документа определяется указатель на объект соответствующего класса. Но для однооконных приложений это класс **CSingleDocTemplate**, а для многооконных – **CMultiDocTemplate**. Новый объект класса также создается динамически при помощи оператора **new**.

Конструктору класса **CMultiDocTemplate** передается 4 параметра. Первый параметр определяет идентификатор ресурсов, используемый совместно с типом документов, управляемых шаблоном. К таким ресурсам относятся меню, пиктограмма, строковый ресурс, таблица акселераторов. Для многооконного приложения в этом параметре указан идентификатор **IDR_MULTITYPE**.

Остальные три параметра – **pDocClass**, **pFrameClass**, **pViewClass** – содержат указатели на структуры **CRuntimeClass**, полученные с помощью макрокоманд **RUNTIME_CLASS** из классов документа **CMultiDoc**, дочернего окна MDI **CChildFrame** и окна просмотра **CMultiView**. Таким образом, шаблон документа объединяет всю информацию, относящуюся к *одному* типу документа.

Созданный шаблон документа заносится в список шаблонов, с которыми работает приложение. Для этого указатель на созданный шаблон документа передается методу **AddDocTemplate** из класса **CWinApp**.

> Если разрабатывается приложение, основанное на многооконном интерфейсе, то объект главного класса приложения может управлять несколькими объектами класса шаблона документа. Они, в свою очередь, управляют созданием документов. При этом *один* шаблон используется для *всех документов* данного типа.

После создания шаблона документа создается главное окно приложения как объект класса **CMainFrame**. Вызывается метод **LoadFrame** класса **CFrameWnd**. Он создает окно, загружает ресурсы, указанные параметром, и связывает их с объектом класса **CMainFrame**. Параметр метода **LoadFrame** определяет меню, пиктограмму, таблицу акселераторов и таблицу строк главного окна приложения.

Указатель на главное окно приложения, которым является главное окно MDI, записывается в элемент данных **m_pMainWnd**, определенного в классе **CWinThread**.

> Заметим, что метод **LoadFrame()** *не отображает* главного окна приложения на экране, для этого, как прежде, нужно вызвать методы **ShowWindow()** и **UpdateWindow()**.

Затем проверяется информация из командной строки приложения. Для этого создается объект **cmdInfo** класса **CCommandLineInfo**. Этот объект передается методу **ParseCommandLine()** класса **CWinApp**. Он заполняет объект **cmdInfo** данными, взятыми из командной строки приложения.

Подготовленный объект передается методу **ProccessShellCommand()** класса **CWinApp** для обработки. Стандартная обработка заключается в том, что происходит создание пустого документа и, соответственно, открытие "пустого" окна просмотра документа.

Для блокирования такого поведения каркаса приложения ваш заключительный код метода **InitInstance()** может выглядеть следующим образом:

```
// создаем главное окно MDI приложения
CMainFrame* pMainFrame = new CMainFrame;

// напрямую загружаем ресурсы главного окна
if (!pMainFrame->LoadFrame(IDR_MAINFRAME))
     return FALSE;
m_pMainWnd = pMainFrame;

// отображаем главное окно
pMainFrame->ShowWindow(m_nCmdShow);
pMainFrame->UpdateWindow();
return TRUE;
```

Класс главного окна **CMainFrame** многооконного приложения практически полностью соответствует классу главного окна однооконного приложения. Следует только обратить внимание, что класс **CMainFrame** наследуется от базового класса **CMDIFrameWnd**, а не от **CFrameWnd**, как это было для однооконного приложения.

Класс дочернего окна MDI.

Многооконное приложение строится с использованием бо́льшего числа классов, чем однооконное приложение. Помимо классов главного окна и классов окон просмотра документов в нем определен еще один класс, непосредственно связанный с отображением дочерних окон MDI. В нашем при-

ложении этот класс называется **CChildFrame**, и он наследуется от базового класса **CMDIChildWnd**, определенного в библиотеке MFC.

Объекты класса **CChildFrame** представляют дочерние окна MDI главного окна приложения. Внутри этих окон отображаются окна просмотра документов. Как правило, вносить изменения методы класса дочернего MDI окна не требуется.

Класс документа приложения **CMultiDoc** наследуется от базового класса **CDocument** библиотеки MFC, при этом класс документа приложения определяется одинаково как для однооконных, так и для многооконных приложений.

> Следует иметь в виду, что для приложений, построенных на основе многооконного интерфейса, объект класса документ создается всякий раз, когда пользователь создает новый или открывает существующий документ. Когда пользователь создает новый документ в однооконном приложении, то на самом деле используется один и тот же объект класса документа. В этом случае вы должны удалить содержимое старого документа и выполнить его повторную инициализацию в методе **OnNewDocument()**.

Если просмотреть исходные тексты приложения, то нигде нельзя обнаружить кода, который бы явно создавал объекты этого класса. Напомним, что объект класса **CMultiDoc** создается шаблоном документа динамически вместе с объектом класса дочернего MDI окна и объектом класса окна просмотра.

Класс окна просмотра документа наследуется от базового класса **CView** библиотеки MFC однотипно как для однооконных, так и для многооконных приложений.

Скорее всего, вам придется переопределить только метод **OnDraw()**, который вызывается при необходимости обновления отображение документа в окне. В качестве единственного параметра методу **OnDraw()** передается указатель на контекст устройства **CDC*pDC**.

> Имея только указатель на объект класса `CDC` нельзя определить координаты недействительного прямоугольника. На самом же деле, `pDC` является указателем на объект класса `CPaintDC` и после несложных преобразований
>
> ```
> CPaintDC* pp = (CPaintDC*)pDC;
> CRect rp(pp->m_ps.rcPaint);
> ```
>
> вы получите искомые координаты в объекте `rp` класса `CRect`.

19.7. Дополнительные компоненты MFC приложений

К дополнительным компонентам стандартных MFC приложений можно отнести меню, панель управления и панель состояния. Работа с методами MFC класса **CMenu** не представляет затруднений и, по существу, полностью повторяет те приемы, что мы рассмотрели в курсе API. Поэтому сосредоточим внимание на остальных элементах.

19.7.1. Панель управления

Ни одно современное Windows приложение не обходится без панели управления. Она обычно располагается в верхней части главного окна приложения, под строкой меню, и содержит ряд кнопок, дублирующих функции некоторых строк этого меню.

Панель управления является отдельным ресурсом, таким же как меню или шаблон диалоговой панели. Каждая кнопка панели управления имеет свой идентификатор **ID**, как правило, такой же как и соответствующий пункт меню, который эта кнопка дублирует. На панели управления можно задавать разделители, при помощи которых кнопки объединяются в смысловые группы.

Если приложение имеет сложную систему управления, то создание для каждой строки меню отдельной кнопки на одной панели управления нецелесообразно, так как количество кнопок может быть достаточно велико. В этом случае обычно создают несколько панелей управления, которые пользователь может открывать и закрывать по своему усмотрению.

Для работы с панелями управления в состав библиотеки MFC включены два класса – **CToolBar** и **CDialogBar**. Они оба наследуются от базового класса **CControlBar**, реализующего основные функции панели управления.

Класс **CToolBar** представляет панель управления, состоящую из кнопок. При желании можно в панель управления класса **CToolBar** помимо кнопок включить и другие элементы управления, например, списки или поля редактирования, однако такая возможность требует дополнительного программирования.

Если необходимо создать панель, содержащую произвольные органы управления, а не только кнопки, то удобнее воспользоваться классом **CDialogBar**. Этот класс позволяет создать панель управления на основе шаблона диалоговой панели.

Кнопки панели управления могут работать как кнопки, как переключатели и как переключатели с зависимой фиксацией – радио-кнопки. Тип кно-

пок панели управления устанавливается методами класса **CToolBar**, в частности, методом **SetButtonStyle()**.

Чтобы создать панель управления, необходимо сначала определить объект класса **CToolBar** или наследованного от него, если вы собираетесь расширить возможности класса **CToolBar**. Обычно объект **CToolBar** включают как элемент главного окна приложения.

19.7.2. Дополнительные возможности панели управления

Так как панель управления является не чем иным, как дочерним окном, то можно самостоятельно разместить в нем другие элементы управления, помимо стандартных кнопок.

Дополнительные возможности панели управления рассмотрим на примере решения задачи, которую можно сформулировать следующим образом: разместить на панели управления дополнительный элемент **Static** для вывода текстового сообщения.

Для этого нужно сделать следующие шаги:

1. Определим новый класс, производный от CToolBar

```
class CTextToolBar : public CToolBar {
public:
CStatic  m_szTitl;
         CTextToolBar();
virtual ~CTextToolBar();
};
```

2. Добавим объект класса **CTextToolBar** в объявление класса главного окна приложения

```
class CTestFrame : public CFrameWnd {
public:
CTextToolBar  m_toolBar;
. . .
```

3. Изменим код метода **OnCreate()** класса главного окна приложения

```
int CTextFrame::OnCreate(LPCREATESTRUCT lpCr)
{
   . . .
   if (!m_toolBar.Create(this) ||
       !m_toolBar.LoadToolBar(IDR_MAINFRAME)) {
          return -1; // fail to create
   }

   // делаем наши добавления
   CSize hw = m_toolBar.CalcFixedLayout(TRUE, TRUE);
   CRect rc(0, 0, 4000, hw.cy);
   m_toolBar.MoveWindow(rc);
   CToolBarCtrl& tb = m_toolBar.GetToolBarCtrl();
```

```
// мы добавляем один элемент, если же потребуется
// несколько, то нужно использовать массив
TBBUTTON arTBbut;
arTBbut.fsStyle   = TBSTYLE_SEP;
arTBbut.idCommand = ID_TB_TEXT;

tb.AddButtons(1, &arTBbut);

// изменим размер последнего элемента
int count = tb.GetButtonCount();
CRect rct;
tb.GetItemRect(count-1, rct);
int xsym = 0;
rct.left += 10;

// рассчитаем ширину дополняемого элемента,
// в зависимости от размера шрифта для
// 20-ти символов текста
CClientDC dc(&m_toolBar);
CFont* oldF = dc.SelectObject(m_toolBar.GetFont());
TEXTMETRIC tm;
dc.GetTextMetrics(&tm);
rct.right = rct.left+20*(xsym = tm.tmAveCharWidth);
dc.SelectObject(oldF);
rct.DeflateRect(0, 3);

// создаем дополнительный элемент на toolbar
if ( m_toolBar.m_szTitl.Create(_T(""),
      WS_CHILD | WS_VISIBLE | SS_SUNKEN,
      rct, &m_toolBar, ID_TB_TEXT) )
   m_toolBar.m_szTitl.SetFont(m_toolBar.GetFont());

m_toolBar.SetBarStyle(m_toolBar.GetBarStyle() |
  CBRS_SIZE_FIXED | CBRS_ALIGN_TOP | CBRS_TOOLTIPS);

// продолжаем инициализацию главного окна
. . . .
```

4. Теперь поместить текст в дополнительный элемент на панели управления можно следующим образом:

```
m_toolBar.m_szTitl.SetWindowText(_T("text"));
```

19.7.3. Панель состояния

В нижней части окна большинства Windows приложений находится строка состояния, которая называется панелью состояния. В ней обычно выводится краткая контекстная подсказка для пользователя. В панели состояния также может отображаться описание текущего режима приложения, текущее время и проч.

Для управления панелями состояния в состав библиотеке MFC включен класс **CStatusBar**, который наследуется от базового класса **CControlBar**.

По умолчанию все MFC приложения имеют панель состояния.

Заметим, что специальные ресурсы, предназначенных для разработки панелей состояния, отсутствуют. Панель состояния создается вручную, при этом каждый элемент панели, который именуется как индикатор, должен быть представлен в специальном массиве.

19.7.4. Индикаторы панели состояния

После того, как все индикаторы панели состояния созданы, можно изменить некоторые их характеристики, воспользовавшись методом **SetPaneInfo()**:

```
void SetPaneInfo(int nIndex,
                 UINT nID,
                 UINT Style,
                 int cx)
```

Параметр `nIndex` определяет порядковый номер – индекс – индикатора в панели управления. Характеристики этого индикатора будут меняться. Метод **SetPaneInfo()** позволяет изменить расположение индикаторов на панели или даже заменить существующий индикатор на другой. Для этого можно указать новый идентификатор через параметр **nID**.

Внешний вид и размер индикатора, заданного параметрами **nIndex** и **nID**, определяется параметрами **Style** и **cx**. Так, после создания панели состояния все ее секции, кроме первой, имеют “утопленный” вид. Для того, чтобы и первая секция выглядела аналогично добавьте следующую строку

```
m_wndStatusBar.SetPaneInfo(0, 0, SBPS_STRETCH, 0);
```

19.7.5. Дополнительные возможности панели состояния

Если во время работы приложения выполняется какой-либо длительный процесс, то на панели состояния можно вывести линейный индикатор **ProgressBar**, чтобы показать ход этого процесса.

Методика размещения полосы **ProgressBar** достаточно проста. В тот момент, когда понадобится вывести полосу **ProgressBar**, нужно создать ее, указав в качестве родительского окна панель состояния. Координаты линейного индикатора желательно выбрать так, чтобы он отображался на месте, например, первого индикатора. Предварительно можно убрать рамку с этого индикатора и заблокировать его так, чтобы в нем не отображался текст.

Для осуществления этого создадим новый класс

```
class BProgress : public CProgressCtrl {
      int  hows;
public :
      Bprogress (CStatusBar& StatusBar, int nCount);
      ~BProgress ();

void  SetCount  (int nCount);
void  operator++();
};
```

Реализация методов класса **BProgress** выглядит следующим образом:

```
BProgress :: BProgress (CStatusBar& StatusBar,
                        int nCount)
{
   SetCount(nCount);
   CRect rm, rs, rp;
   CFrameWnd* mf = (CFrameWnd*)AfxGetApp()->m_pMainWnd;
   mf->GetClientRect(rm);
   StatusBar.GetClientRect(rs);
   StatusBar.GetItemRect(0, rp);
   rp.top    += (rm.bottom-rs.bottom);
   rp.bottom += (rm.bottom-rs.bottom);
   Create(0, rp, mf, 0);
   SetRange(0, 100);
   SetPos(10);
   ShowWindow(SW_SHOW);
}

void BProgress :: SetCount (int nCount)
{
   hows = nCount;
   if (hows < 1) hows = 1;
}

BProgress :: ~BProgress ()
{
   DestroyWindow();
}

void BProgress :: operator++()
{
   int prev = OffsetPos(0);
   prev += 90/hows;
   if (prev > 100) prev = 100;
   SetPos(prev);
}
```

Сразу после создания объекта класса **BProgress**, первая секция панели состояния заменяется элементом управления **ProgressBar**. Для продвижения индикатора после каждого шага вычислительного процесса, ис-

пользуйте перегруженную операцию инкремента для объекта класса **BProgress**, например:

```
BProgress prgb(m_wndStatusBar, 12);
. . .
prgb++;
. . .
```

В данном примере полагается, что вычислительный процесс состоит из 12-ти стадий.

Итак, в этом разделе мы рассмотрели вопросы, связанные с использованием MFC архитектуры **Документ-Вид** для SDI и MDI приложений. Были освещены как базовые приемы работы, так и некоторые технологические расширения, направленные на улучшение интерфейсной части MFC приложений.

20. DLL и MFC

Вначале кратко дадим основные понятия из теории DLL, затем укажем способы использования DLL библиотек и, наконец, пути создания собственных DLL.

20.1. Библиотеки динамической компоновки

Операционная система Windows всегда использовала библиотеки динамической компоновки DLL (Dynamic Link Library), в которых содержались реализации наиболее часто применяемых функций. Другими словами, DLL – это просто наборы функций, собранные в библиотеки. Однако в отличие от своих статических родственников – файлов .lib – библиотеки DLL не присоединены непосредственно к выполняемым файлам с помощью редактора связей. В выполняемый файл занесена только информация об их местонахождении. В момент выполнения программы загружается вся библиотека целиком. Благодаря этому разные процессы могут пользоваться совместно одними и теми же библиотеками, находящимися в памяти. Такой подход позволяет сократить объем памяти, необходимый для нескольких приложений, использующих много общих библиотек, а также уменьшить размер исполняемого EXE файла.

> Если библиотека используется только одним приложением, лучше сделать ее обычной, статической. Более того, если функции, входящие в состав библиотеки будут использоваться только в одной программе, просто вставьте их в соответствующий файл с исходным текстом программы.

Чаще всего проект подключается к DLL статически, или неявно, на этапе компоновки. Загрузкой DLL при выполнении программы управляет операционная система. Однако DLL можно загрузить и явно, или динамически, в ходе работы приложения.

20.2. Библиотеки импортирования

При статическом подключении DLL имя .lib-файла определяется среди прочих параметров редактора связей в командной строке или на вкладке “Link” диалогового окна “Project Settings” среды Developer Studio. Однако .lib-файл, используемый при неявном подключении DLL, – это не обычная статическая библиотека. Такие .lib-файлы называются библиотеками импортирования (import libraries). В них содержится не сам код библиотеки, а только ссылки на все функции, экспортируемые из файла DLL, в котором все и хранится.

20.3. Согласование интерфейсов

При использовании собственных библиотек или библиотек независимых разработчиков придется обратить внимание на согласование вызова функции с ее прототипом.

По умолчанию в VC++ интерфейсы функций согласуются по правилам C++, а именно:

1) параметры заносятся в стек справа налево;
2) вызывающая программа отвечает за их удаление из стека при выходе из функции;
3) имя функции расширяется.

Расширение имен (name decoration или name mangling) позволяет редактору связей различать перегруженные функции, т.е. функции с одинаковыми именами, но разными списками аргументов. Однако в старой библиотеке Си функции с расширенными именами отсутствуют.

Хотя все остальные правила вызова функции в Си идентичны правилам вызова функции в C++, в библиотеках Си имена функций не расширяются. К ним только добавляется впереди символ подчеркивания.

Если необходимо подключить библиотеку на Си к приложению на C++, все функции из этой библиотеки придется объявить как внешние в формате Си:

```
extern "C" int OldCFunction (int nParam);
```

Модификатор **`extern "C"`** можно применить и к целому блоку, так что вместо модификации каждой функции в отдельности можно обойтись всего тремя строками:

```
extern "C"
{
      #include "oldCLib.h"
}
```

В программах для старых версий Windows использовались также соглашения о вызове функций языка **`PASCAL`** для функций Windows API. В новых программах следует использовать модификатор **`WINAPI`**, преобразуемый в **`_stdcall`**.

20.4. Загрузка неявно подключаемой DLL

Приложение во время запуска пытается найти все файлы DLL, неявно подключенные к приложению, и поместить их в область оперативной памяти, занимаемую данным процессом. Поиск файлов DLL операционной системой осуществляется в следующей последовательности.

1) каталог, в котором находится EXE-файл;
2) текущий каталог процесса;

3) системный каталог Windows.

Если хотя бы одна библиотека DLL не обнаружена, приложение завершает свою работу с выводом предупреждающего сообщения.

Если нужная библиотека найдена, она помещается в оперативную память процесса, где и остается до его окончания. Теперь приложение может обращаться к функциям, содержащимся в DLL.

20.5. Динамическая загрузка и выгрузка DLL

Второй путь – связать программу с модулем библиотеки DLL во время выполнения программы.

При таком способе в процессе создания приложения (см. далее) не нужно использовать библиотеку импорта.

В частности, можно определить, какая из библиотек DLL доступна пользователю, или разрешить пользователю выбрать, какая из библиотек будет загружаться. Таким образом, можно использовать разные DLL, в которых реализованы одни и те же функции, но выполняющие различные действия. Например, приложение, предназначенное для независимой передачи данных, сможет в ходе выполнения принять решение, загружать ли DLL для протокола TCP/IP или для другого протокола.

20.5.1. Загрузка обычной DLL

Первое, что необходимо сделать при динамической загрузке DLL, – это поместить модуль библиотеки в память процесса. Данная операция выполняется с помощью функции **`LoadLibrary()`**, имеющей единственный аргумент – имя загружаемого модуля. Если модуль обнаружен и библиотека успешно загрузилась, функция **`LoadLibrary()`** возвращает ее дескриптор, который используется для доступа к функциям библиотеки:

```
HINSTANCE hDll;
if ((hDll=LoadLibrary("anyDLL"))==NULL) {
    // не удалось загрузить DLL
}
// пользуемся функциями DLL через hDll
```

Стандартным расширением файла библиотеки Windows считает .dll, если не указано иное расширение. Если в имени файла указан и путь, то только он будет использоваться для поиска файла. В противном случае Windows будет искать файл по той же схеме, что и в случае неявно подключаемой DLL.

Полный путь библиотеки будет сравнен с путем библиотек DLL, уже загруженных данным процессом. Если обнаружится тождество, то приложению возвращается дескриптор уже подключенной библиотеки.

Перед тем, как использовать функции библиотеки, необходимо получить их адрес по следующей схеме:

1) директивой **typedef** нужно определить тип указателя на функцию и определить переменную этого типа;

2) следует получить дескриптор библиотеки, при помощи которого и определить адреса нужных функций.

```
// объявить указатель на функцию.
typedef int (WINAPI *PFN_function)(char *);
. . .
PFN_function pfnFunction =
   (PFN_function)GetProcAddress(hDll,"anyFunction");
. . .
int iCode = pfnFunction("Proba");
```

Адрес функции определяется при помощи функции **GetProcAddress()**, ей следует передать имя библиотеки и имя функции. Последнее должно передаваться в том виде, в котором экспортируется из DLL.

Можно также сослаться на функцию по ее порядковому номеру, по которому она экспортируется. Для создания библиотеки должен использоваться def-файл, о котором будет рассказано далее:

```
pfnFunction=(PFN_Function)::GetProcAddress(hDll,
                                  MAKEINTRESOURCE(1));
```

20.5.2. Выгрузка DLL

После завершения работы с библиотекой динамической компоновки, ее нужно выгрузить из памяти процесса с помощью API функции **FreeLibrary(hDll)**.

Нужно заметить, что процесс загрузки-выгрузки ядром операционной системы любой DLL не так прост, как это может показаться с первого взгляда. Так, для каждой библиотеки ядро ведет свой скрытый счетчик, который увеличивается на единицу, когда какой-либо процесс загружает данную библиотеку, и, соответственно, уменьшается – при обращении к функции **FreeLibrary()**. Следовательно, при обращении счетчика в ноль DLL реально выгружается из памяти. Причем ядро только следит за значением счетчика и не контролирует правильность обращения к функциям загрузки-выгрузки. Другими словами, если два процесса загрузили одну и ту же библиотеку, а первый процесс дважды(**!**) обратился к **FreeLibrary()**, то последствия такого шага будут не предсказуемы для обоих процессов.

20.5.3. Библиотеки ресурсов

Динамическая загрузка из библиотеки DLL применима и к ресурсам. При этом используется следующий порядок. Если ресурс, с заданным идентификатором, присутствует в EXE файле, он загружается из EXE файла, в противном случае – из ресурсов DLL библиотеки.

Чтобы изменить этот порядок, нужно сделать следующее:

1) разместить DLL в памяти, вызвав функцию **`LoadLibrary()`**;

2) сохранить дескриптор ресурсов приложения

```
HINSTANCE hResApp = AfxGetResourceHandle();
```

3) заменить дескриптор ресурсов приложения

```
AfxSetResourceHandle(hResDLL);
```

4) восстановить дескриптор ресурсов приложения

```
AfxSetResourceHandle(hResApp);
```

Такой подход удобен, если нужно использовать различные наборы ресурсов, например для разных языков.

> С помощью функции **`LoadLibrary()`** можно загружать в память все исполняемые модули, *не выполняя их*! Дескриптор модуля может использоваться при обращении к функциям **`FindResource()`** и **`LoadResource()`** для поиска и загрузки ресурсов в этом модуле. Выгружают модули из памяти также при помощи функции **`FreeLibrary()`**.

20.6. Создание DLL

Теперь, познакомившись с принципами работы библиотек DLL в приложениях, рассмотрим способы их создания.

Проще всего создать новый проект DLL с помощью мастера AppWizard, который автоматически выполняет многие операции. При этом имеется возможность создания библиотек с поддержкой MFC и без нее.

Для обычных DLL, которые могут быть подключены к любому приложению, даже созданному в других программных средах, например VB, необходимо выбрать тип проекта *Win32 Dynamic-Link Library*. Новому проекту будут присвоены все необходимые параметры для создания обычной DLL библиотеки. Все остальное, файлы исходных текстов в том числе, придется добавлять к проекту вручную.

Если же планируется в полной мере использовать функциональные возможности MFC, такие как документы и их представления, или вы намерены создать сервер автоматизации OLE, лучше выбрать тип проекта MFC AppWizard (dll). В этом случае, помимо присвоения проекту параметров для подключения динамических библиотек, мастер проделает некоторую допол-

нительную работу. В проект будут добавлены необходимые ссылки на библиотеки MFC и файлы исходных текстов, содержащие описание и реализацию в библиотеке DLL объекта класса приложения, производного от CWinApp.

> При создании DLL проекта на базе MFC вам будет предложен выбор из трех вариантов. Если вы хотите поместить в DLL библиотеку собственные базовые классы, от которых будут наследоваться классы в других приложениях, выберите расширенную MFC DLL библиотеку.

20.6.1. Загрузка MFC расширений динамических библиотек

При загрузке MFC-расширений для DLL вместо функций **`LoadLibrary()`** и **`FreeLibrary()`** используются функции **`AfxLoadLibrary()`** и **`AfxFreeLibrary()`**. Последние почти идентичны функциям Win32 API, они лишь дополнительно гарантируют то, что структуры MFC, инициализированные расширением DLL, не будут запорчены другими потоками вашего приложения.

20.6.2. Функция DllMain

Большинство библиотек DLL – просто коллекции практически независимых друг от друга функций, экспортируемых в вызывающее приложение. Кроме функций, предназначенных для экспортирования, в каждой Win32 DLL библиотеке есть функция **`DllMain()`**. Эта функция предназначена для инициализации и очистки DLL. Структура простейшей функции **`DllMain()`** может выглядеть следующим образом:

```
BOOL WINAPI DllMain (HANDLE hInst,
                     DWORD  dwReason,
                     LPVOID lpReserved)
{
   switch (dwReason) {
   case DLL_PROCESS_ATTACH:
   // библиотека прикрепляется к процессу
   break;

   case DLL_THREAD_ATTACH:
   // библиотека прикрепляется к потоку
   break;

   case DLL_THREAD_DETACH:
   // очистка структур потока
   break;

   case DLL_PROCESS_DETACH:
   // очистка структур процесса
   break;
   }
```

```
    return TRUE;
}
```

Функция **DllMain()** вызывается в нескольких случаях. Причина ее вызова определяется параметром **dwReason**, который может принимать одно из следующих значений.

При первой загрузке библиотеки DLL процессом вызывается функция **DllMain()** с **dwReason**, равным **DLL_PROCESS_ATTACH**. Каждый раз при создании процессом нового потока **DllMain()** вызывается с **dwReason**, равным **DLL_THREAD_ATTACH**, кроме первого потока, потому что в этом случае **dwReason** равен **DLL_PROCESS_ATTACH**.

При завершении потока приложения, кроме первого, **dwReason** будет равен **DLL_THREAD_DETACH**, а по окончании работы процесса с DLL функция **DllMain()** вызывается с параметром **dwReason**, равным **DLL_PROCESS_DETACH**.

Все операции по инициализации и очистке для процессов и потоков, в которых нуждается DLL, необходимо выполнять на основании значения **dwReason**. Инициализация процессов обычно ограничивается выделением ресурсов, совместно используемых потоками, в частности загрузкой разделяемых файлов и инициализацией библиотек. Инициализация потоков применяется для настройки режимов, свойственных только данному потоку, например для инициализации локальной памяти.

Параметр **hInst** представляет собой дескриптор копии библиотеки, аналогично дескриптору копии приложения. Можно сохранить дескриптор **hInst** в глобальной переменной, если он потребуется для функций, работающих, например, с загрузкой ресурсов из DLL.

Параметр **lpReserved** зарезервирован для внутреннего использования Windows. Если библиотека DLL была загружена динамически, **lpReserved** будет равен NULL. При статической загрузке этот указатель будет ненулевым.

В случае успешного завершения функция **DllMain()** должна возвращать TRUE. В случае возникновения ошибки возвращается FALSE, и дальнейшие действия прекращаются.

20.6.3. Экспортирование функций из DLL

Обычная DLL библиотека состоит из функций, которые будут использоваться в других приложениях – тогда говорят об экспортируемых функциях –, и вспомогательных функций, которые используются в качестве внутренних.

Чтобы приложение могло обратиться к функциям DLL библиотеки, каждая из таких функций должна занимать строку в таблице экспортируе-

мых функций DLL. Эта таблица заполняется на этапе компиляции, если некоторые функции имеют модификатор **__declspec(dllexport)**. Заметим, что в состав MFC входит несколько макросов, определяющих этот модификатор, в том числе **AFX_CLASS_EXPORT**, **AFX_DATA_EXPORT** и **AFX_API_EXPORT**.

20.6.4. Файл определения модуля

Файл определения модуля .def должен быть включен в ваш проект только в том случае, если предполагается, что ваша библиотека будет использоваться в приложениях, созданных с помощью языков разработки, отличных от C++. Дело в том, что компилятор C++ выполняет декорирование имен экспортируемых объектов, следовательно, для программ, созданных на "чистом" Си или Visual Basic такие библиотеки будут практически бесполезны.

Синтаксис файлов с расширением .def в Visual C++ достаточно прост, главным образом потому, что сложные параметры, использовавшиеся в ранних версиях Windows, в Win32 более не применяются. Теперь этот файл содержит имя и необязательное описание библиотеки, а также список экспортируемых функций, например:

```
LIBRARY             "MyDLL"
DESCRIPTION         'MyDLL - пример DLL-библиотеки'
EXPORTS
      ProbaFunction       @1
```

В строке экспорта функции нужно указать ее порядковый номер, поставив перед ним символ @. Этот номер может затем использоваться при обращении к **GetProcAddress()**. Если такой номер не указан, компилятор присвоит его самостоятельно, но способом, который непредсказуем.

В строке экспорта можно указать параметр NONAME. Он запрещает компилятору включать имя функции в таблицу экспортирования DLL, порой это позволит сэкономить место в библиотечном файле:

```
ProbaFunction  @1  NONAME
```

> При неявном подключении DLL к приложению, используются не имена функций, а только их порядковые номера. Приложениям же, загружающим библиотеки DLL динамически, в случае NONAME потребуется передавать в **GetProcAddress()** порядковый номер, а не имя функции.

20.6.5. Экспортирование классов

Создание def-файла для экспортирования даже простых классов из динамической библиотеки может оказаться довольно сложным делом. Пона-

добится явно экспортировать каждую функцию, которая может быть использована внешним приложением. Однако есть более простой способ. Если в объявлении класса воспользоваться макро-модификатором **AFX_CLASS_EXPORT**, компилятор сам позаботится об экспортировании необходимых функций, позволяющих приложению полностью использовать класс, содержащийся в DLL.

В def-файл определения классов не включаются и по другой причине – C++ классы не могут быть использованы в приложениях, созданных на других языках программирования.

20.6.6. Статическая память DLL

В Win32 библиотека DLL загружается в адресное пространство вызывающего ее процесса. Следовательно, каждому процессу предоставляется отдельная копия всех статических переменных, определенных в DLL, которые инициализируется каждый раз, когда библиотеку загружает новый процесс. Так что статические переменные динамической библиотеки не могут использоваться совместно всеми вызывающими процессами, как это было в Winl6.

И все же, выполнив ряд замысловатых манипуляций над сегментом данных DLL, можно создать общую область памяти для всех процессов, использующих данную библиотеку.

Допустим, имеется глобальный массив целых чисел, который должен использоваться *всеми процессами*, загружающими данную библиотеку. Это можно реализовать, используя директиву препроцессора `pragma`:

```
#pragma data_seg(".myseg")
AFX_DATA_EXPORT int sharedArr[10] = {0};
// другие переменные общего пользования
#pragma data_seg()

#pragma comment(linker, "/SECTION:.myseg,rws");
```

Все переменные, объявленные между директивами `#pragma data_seg()`, размещаются в отдельном сегменте `.myseg` и к ним можно иметь доступ, также как к библиотечным функциям, но только при выполнении условий:

1) ОБЪЕКТЫ ИМЕЮТ МОДИФИКАТОР ЭКСПОРТА;

2) объекты инициализированы;

3) имена объектов помещены в .def файл.

Директива `#pragma comment` – не совсем обычный комментарий. Она дает указание редактору связей системы пометить новый раздел, как разрешенный для чтения, записи и совместного доступа одновременно. За-

метим, что этот прием работает под всеми Windows платформами без исключений.

Итак, если проект динамической библиотеки создан с помощью AppWizard и .def-файл модифицирован соответствующим образом – этого достаточно для успешной компиляции DLL. Если же файлы проекта создаются вручную или другими способами без помощи AppWizard, в командную строку редактора связей следует включить параметр /DLL. В результате вместо EXE файла будет создана библиотека DLL.

Для MFC предусмотрен ряд особых режимов, касающихся использования динамической библиотекой библиотек MFC. Этому вопросу посвящен следующий раздел.

20.7. Использование MFC в DLL

Как уже отмечалось, имеется два уровня использования каркаса MFC в DLL. Первый из них – это обычная динамическая библиотека на основе MFC. Она может использовать MFC, но не может передавать указатели на объекты MFC между DLL и приложениями. Второй уровень – реализован в динамических расширениях MFC. Использование этого вида динамических библиотек требует некоторых дополнительных усилий по настройке, но позволяет свободно обмениваться указателями на объекты MFC между DLL и приложением.

20.7.1. Обычные MFC DLL

Обычные MFC DLL позволяют применять MFC в динамических библиотеках. При этом приложения, обращающиеся к таким библиотекам, не обязательно должны быть построены на основе MFC. В обычных DLL можно использовать MFC любым способом, в том числе путем создания в DLL новых классов на базе классов MFC, но только для “внутреннего” использования. Обычные DLL не могут обмениваться с приложениями указателями на классы, производные от MFC.

> Архитектура обычных DLL рассчитана на ее использование другими средами программирования, такими как Visual Basic и PowerBuilder.

20.7.2. Управление информацией о состоянии MFC

В каждом модуле процесса MFC содержится информация о его состоянии. Таким образом, информация о состоянии DLL отлична от информации о состоянии вызвавшего ее приложения. Поэтому любые экспортируемые из библиотеки функции, обращение к которым исходит непосредственно из приложений, должны сообщать MFC, какую информацию состояния использовать. В обычной MFC DLL, использующей динамические библиотеки

MFC, перед вызовом любой подпрограммы MFC в начале экспортируемой функции нужно поместить следующую строку:

```
AFX_MANAGE_STATE(AfxGetStaticModuleState());
```

Данный оператор определяет использование соответствующей информации о состоянии во время выполнения функции, обратившейся к данной подпрограмме.

20.7.3. Динамические расширения MFC

MFC позволяет создавать такие библиотеки DLL, которые воспринимаются приложениями не как набор отдельных функций, а как расширения MFC. С помощью данного вида DLL можно создавать новые классы, производные от классов MFC, и использовать их в своих приложениях.

Чтобы обеспечить возможность свободного обмена указателями на объекты MFC между приложением и DLL, нужно создать динамическое расширение MFC. DLL этого типа подключаются к динамическим библиотекам MFC так же, как и любые приложения, использующие динамическое расширение MFC.

> Следует обратить внимание, что динамические библиотеки данного типа *не должны содержать* объектов, производных от CWinApp.

20.7.4. Инициализация динамических расширений

Чтобы удовлетворить требованиям структуры MFC, динамические расширения MFC требуют дополнительной начальной настройки. Соответствующие операции выполняются функцией **`DllMain()`**. Рассмотрим пример этой функции, созданный мастером AppWizard.

```
static AFX_EXTENSION_MODULE ExtDLL = { NULL, NULL };

extern "C"
int APIENTRY DllMain(HINSTANCE hInst,
                     DWORD dwReason,
                     LPVOID lpReserved)
{
   if (dwReason == DLL_PROCESS_ATTACH) {
       TRACE("ExtDLL.DLL Initializing!\n");
       AfxInitExtensionModule(ExtDLL, hinstance);
       new CDynLinkLibrary(ExtDLL);
   }
   else if (dwReason == DLL_PROCESS_DETACH) {
       TRACE("ExtDLL.DLL Terminating!\n");
       AfxTermExtensionModule(ExtDLL);
   }
   return TRUE;
}
```

Самой важной частью этой функции является вызов глобальной MFC функции **AfxInitExtensionModule()**. Это инициализация динамической библиотеки, позволяющая ей корректно работать в составе каркаса MFC. Аргументами данной функции являются передаваемый в **DllMain()** дескриптор библиотеки DLL и структура AFX_EXTENSION_MODULE, содержащая информацию о подключаемой к MFC динамической библиотеке.

Нет необходимости инициализировать структуру AFX_EXTENSION_MODULE явно. Однако объявить ее нужно обязательно. Инициализацией же займется конструктор класса **CDynLinkLibrary**. Этот конструктор не только инициализирует структуру AFX_EXTENSION_MODULE, но и добавляет новую библиотеку в список DLL, с которыми в данный момент может работать MFC.

20.7.5. Загрузка динамических расширений MFC

MFC позволяет динамически загружать и выгружать DLL, в том числе и расширения. Для корректного выполнения этих операций над DLL библиотекой в ее функцию **DllMain()** в момент отключения от процесса необходимо добавить вызов функции **AfxTermExtensionModule()**, в качестве параметра которой передается та же структура AFX_EXTENSION_MODULE.

Напомним, что данная библиотека DLL является динамическим расширением MFC и должна загружаться и выгружаться динамически, с помощью функций **AfxLoadLibrary()** и **AfxFreeLibrary()**.

20.7.6. Экспортирование классов из динамических расширений

Для обеспечения экспортирования в приложение классов и функций из динамического расширения, необходимо использовать специальные модификаторы при объявлении экспортируемых классов и функций. Так, например, если предполагается, что в приложениях будет использоваться объект класса **CInfoPanel**, его объявление класса в заголовочном файле должно выглядеть следующим образом:

```
class AFX_EXT_CLASS CInfoPanel : public CDialog {
//. . . .

};
```

Если в этой же библиотеке присутствует некая функция **IsDuplicApp()**, которая тоже будет использоваться в приложениях, ее объявление должно быть:

```
AFX_API_EXPORT  BOOL  IsDuplicApp ();
```

Итак, мы детально рассмотрели все вопросы, касающиеся создания и использования как обычных, так и расширенных MFC библиотек динамической загрузки. Можно ли обойтись без создания собственных DLL? Скорее нет, чем да. Поскольку помимо удобства использования, такие библиотеки просто необходимы при решении целого класса задач, например, реализации системных “hooks”.

Вместо заключения – API, MFC, что же дальше?

Вот мы и закончили изучение курса, посвященного программированию под Windows. Но, конечно же, не закончилось ваше самостоятельное исследование этого вопроса. Ведь мы задели только “вершину айсберга” – рассмотрели базовые понятия, и освоили довольно простые технологии. Надеюсь, те знания, что вы приобрели, будут твердой основой для дальнейшего, углубленного изучения предмета как в классических направлениях, перечисленных в настоящем пособии, так и в принципиально новых, возникших в самое последнее время.

Дальнейшее развитие Windows корпорация Microsoft видит в использовании dot-NET Framework технологии, основные идеи которой заключаются в переходе на новый формат исполняемых файлов с применением “сборок” (assemblies). В качестве первой .NET платформы Microsoft анонсировала beta-версию Windows 2003 Server, декларируя, что пользователи, помимо всего прочего, получат существенное повышение эффективности выполнения приложений, особенно ориентированных на использовании Internet технологий, в сравнении, например, с Windows 2000 Server [1].

Конечно же, новая платформа требует и новых инструментальных средств разработки. Здесь приемником C/C++ назван C# (читается как Си Шарп) – язык, который можно назвать конгломератом C++, Java и, пожалуй, Visual Basic. Примеры простейших C# программ выглядят изящно, компилируются намного быстрее, чем MFC проекты, да и занимают многократно меньше места на жестком диске в сравнении с тем же MFC. Однако к приверженцам Си придет разочарование, например, в C# практически исчезали указатели! Хотя их еще можно использовать в так называемом unsafe режиме, но нужно иметь ввиду, что .NET Framework реализует garbage collection – так называемый “сборщик мусора в памяти”. А это означает, что при выделении требуемого количества памяти какому-либо процессу, ваши объекты могут быть передвинуты в любой момент времени, после чего, конечно же, указатели окажутся недействительными! Частое же использование fixed оператора для фиксации объектов в памяти может снизить эффективность работы системы в целом.

И все же за C# будущее, и программирование на нем продолжает цепочку API – MFC, и никакие трудности не должны вас останавливать на пути дальнейшего освоения технологии разработки программных систем.

Литература к первой части

1. Гоулд И.Г., Тутилл Дж.С. Терминологическая работа IFIP (Международная федерация по обработке информации) и ICC (Международный вычислительный центр) // Журн. вычисл. матем. и матем. физ., 1965, №2. - С. 377-386.
2. Майерс Г. Надежность программного обеспечения. - М.: Мир, 1980.
3. Турский В. Методология программирования. - М.: Мир, 1981.
4. Дейкстра Э. Дисциплина программирования. - М.: Мир, 1978.
5. Жоголев Е.А. Система программирования с использованием библиотеки подпрограмм / Система автоматизация программирования. - М.: Физматгиз, 1961. - С. 15-52.
6. Кауфман В.Ш. Языки программирования. Концепции и принципы. - М.: Радио и связь, 1993.
7. Требования и спецификации в разработке программ. - М.: Мир, 1984.
8. Буч Г. Объектно-ориентированное проектирование с примерами применения. - М.: Конкорд, 1992.
9. Sommerville I. Software Engineering. - Addison-Wesley Publishing Company, 1992.
10. Зиглер К. Методы проектирования программных систем. - М.: Мир, 1985.
11. Жоголев Е.А. Введение в технологию программирования. - М.: “ДИАЛОГ-МГУ”, 1994.
12. Липаев В.В. Качество программного обеспечения. - М.: Финансы и статистика, 1983.

Литература ко второй части

1. Рихтер Дж. Windows для профессионалов // М.: Русская Редакция, 1996.
2. Петзолд Ч. Программирование для Windows 95, Т.1-2. // СПб.: BHV, 1997.
3. Шилдт Г. Программирование на C и C++ для Windows 95. // СПб.: BHV, 1996.
4. Фролов А.В., Фролов Г.В. Библиотека системного программиста, Т.11-14, 17. // М.: "ДИАЛОГ-МИФИ", 1994-1995.

Литература к третьей части

1. MSDN.
2. Мюллер Дж. Visual C++ 5. Руководство для профессиональной работы. Спб: BHV, 1998.

3. Петзолд Ч. Программирование для Windows 95, Т.1-2. // СПб.: BHV, 1997.
4. Шилдт Г. MFC: основы программирования. - С.-П.: “BHV”, 1997.
5. Биллинг В.А, Мусикаев И.Х.. Visual C++. Книга для программистов. М.: “Русская редакция”, 1996.
6. Боровской И.Г. Технология разработки программных систем. Учебное пособие // Томск, ТУСУР, 2005. – 300с.